Instruction

Z-Wave ZW0102/ZW0201/ZW0301 Appl. Prg. Guide v5.00 (Beta1)

Document No.:	**INS10247**
Version:	**11**
Description:	**Guideline for developing ZW0102, ZW0201 and ZW0301 based applications using the application programming interface (API) based on Developer's Kit v5.00**
Written By:	**ABR;EFH;HEH;JFR;JRM;JSI;PSH;SSE**
Date:	**2007-03-20**
Reviewed By:	**HEH;JFR;JRM;JSI;PSH;SSE**
Restrictions:	**None**

Approved by:

Date	CET	Initials	Name	Justification
2007-03-20	10:26:53	NTJ	Niels Thybo Johansen	

This document is the property of Zensys A/S. The data contained herein, in whole or in part, may not be duplicated, used or disclosed outside the recipient for any purpose other than to conduct technical evaluation. This restriction does not limit the recipient's right to use information contained in the data if it is obtained from another source without restriction.

CONFIDENTIAL

REVISION RECORD

Doc. Rev	Date	By	Pages affected	Brief description of changes
1	20050223	JFR	4.8.9	ZW_RequestNodeNeighborUpdate callback function parameters changed
1	20050223	JFR	8.7	External EEPROM image for the development controller have changed name form zerofilled16kb_extern_eep.hex to Extern_eep.hex. Description of downloading to ZW0201 based module included.
1	20050223	JFR	8.4	Hardware development components for ZW0201 added
1	20050224	JFR	3	Updated with respect to directories, files and descriptions
1	20050228	JFR	5.3.1	Corrected the address of a node in test mode
2	20050329	JSI	5.3.3 5.4.1 5.6	Added note about Bridge Controller Multicast handling Added note about Primary Controller created with Create New Primary nodeID assignment Updated note about Bridge Controller functionality
2	20050411	JSI	4.1, 4.2	Refreshed to follow library flow
2	20050412	JSI	5.6.5 5.4.19	Updated ZW_GetVirtualNodes description Updated ZW_GetRoutingInfo description
2	20050414	JFR	5.1.1.4	Updated library memory usage
3	20050622	JFR JSI	7.5	Updated description of Serial API
3	20050706	JFR	5.3.1	The ApplicationCommandHandler extended to also return the type of frame received (single cast, mulitcast or broadcast frame).
3	20050803	JSI	4.8.9	The ZW_AssignReturnRoute, ZW_DeleteReturnRoute, ZW_AssignSUCReturnRoute and ZW_DeleteSUCReturnRoute now returns a BOOL to indicate if the operation was started or not.
3	20050804	JFR	4.8.9	Documented ZW_SendSUCID
4	20050921	JFR	5.3.3	Number of timers available in the Timer API etc.
4	20050922	JSI	Chapter 5	Updated description of Serial API calls to be consistent with the Serial API sample code. Documented ZW_StoreHomeID
5	20051005	SSE	5.3.2	Update the description of ZW_SetSleepMode API.
5	20051010	HEH	5.3.3 5.3.11	Added TRANSMITOPTION_NO_ROUTE. Added ZW_AreNodesNeighbours. Added ZW_SendDataMeta Added ZW_RequestNewRouteDestinations Added ZW_IsNodeWithinDirectRange
5	20051010	PSH	5.3.11	Updated description of ZW_GetControllerCapability()
5	20051013	JSI	5.3.1 5.8, 5.3 7.6.2.3 5.3.4	Added ApplicationSlaveUpdate Updated ApplicationControllerUpdate with UPDATE_STATE_NODE_INFO_RECEIVED status Moved TRIAC API section to Common API section as all libraries now contain the TRIAC API Updated SerialAPI error handling description Added FUNC_ID_SERIAL_API_SET_TIMEOUTS Serial API function description. Added SerialAPI description for ZW_SendDataAbort
6	20051027	PSH	5.3.7	Allowed NULL pointer as pHomeID parameter to MemoryGetID()
6	20051103	JSI	7.6.2.3	FUNC_ID_SERIAL_API_SET_TIMEOUTS only returns the old timeouts
6	20051104	JSI	5.3.2	Added missing serialAPI description for ZW_SetRFReceiveMode
6	20051117	HEH	5.3.1 5.3.7 4.1, 7.3	Added missing state to ApplicationControllerUpdate UPDATE_SUC_NODE_ID Added Description of NULL pointer usage to ZW_MEM_PUT_BUFFER Added description of Uninitialized RAM for the ZW_0201
6	20051118	HEH	5.7.2	Updated SerialAPI description for ZW_StoreNodeInfo

CONFIDENTIAL

REVISION RECORD

Doc. Rev	Date	By	Pages affected	Brief description of changes
6	20051207	JSI	5.3.1 5.3.2 4.8.9 7.9.2.5 7.9.2.3 7.9.2.6	Updated ApplicationControllerUpdate Added SerialAPI description for ZW_RFPowerLevelSet Added SerialAPI description for ZW_SendDataMeta Added section "Serial API capabilities" describing the Serial API function FUNC_ID_SERIAL_API_GET_CAPABILITIES Added description of the usage of the CAN frame Added section "Serial API softreset" describing the Serial API function FUNC_ID_SERIAL_API_SOFT_RESET
7	20051221	JFR	5.1.1.2	Updated library functionality table
8	20060113	MVO	All	New 1st page/header/footer contents. New Doc. No.
8	20060127	JSI	5.3.2	Updated ZW_SetRFReceiveMode with new return parameter. SerialAPI description also updated accordingly.
8	20060215	SSE	5.12.1 5.12.2	Added the default values of RF options in the description of "App_RFSetup.a51"
8	20060227	SSE	5.3.8	Improved the description of the ADC_Init API
8	20060320	JFR		Removed bSUCNode parameter in ZW_ASSIGN_SUC_RETURN_ROUTE API call and corresponding swerail API call.
8	20060322	HEH	5.8.3 5.7.2 5.3.1	Added description of ZW_SUPPORT9600_ONLY call Inserted Return value description for ZW_StoreNodeInfo Rewrote ApplicationNodeInformation description and updated with defines for deviceOptionsMask.
8	20060327	JSI	5.2	Updated RECEIVE_STATUS_ROUTED_BUSY and RECEIVE_STATUS_LOW_POWER rx status bit description
8	20060403	TKR	Several	Added definitions for several SerialAPI calls
8	20060410	JFR	5.3.1	Updated ApplicationPoll frequency measurements for ZW0102/ZW0201
8	20060420	JFR	5.1.1.4	Refer to the Software Release Note with respect to library sizes.
9	20060426	JFR	Front page	Fixed formatting problem
10	20060519	JFR	4.8.9	Added a dummy parameter to serial API call ZW_AssignSUCReturnRoute to be backward compatible. Have previously only been removed in v4.10.
10	20060807	JSI	5.4.1	Update ZW_RequestNodeInfo description
10	20060817	SSE	5.3.6	Changed the description of the API parameters
11	20060929	PSH	5.3.2	Fixed upper/lower case mismatch in WatchDog functions
11	20061013	JFR	5.3.2 5.9.3	ZW_RFPowerlevelRediscoverySet added ZW_RequestNodeInfo now available for routing slaves too.
11	20061025	JFR	5.3.2	ZW_SLEEP macro call removed for the 200 Series
11	20061027	EFH		Memory - Discontinue ZW_IsSUCActive
11	20061031	SSE		Edited the ZWAVE_MEM_FLUSH api text
11	20061110	ABR	5.3.3	Added the use of TRANSMIT_OPTION_RETURN_ROUTE to ZW_SendDataMeta. (Already implemented. Doc bug discovered by customer...)
11	20061214	SSE	5.3.1	Added the RECEIVE_STATUS_FOREIGN_FRAME rx status bit to ApplicationCommandHandler
11	20061214	SSE	5.3.1	Updated the description of ApplicationRfNotify function.
11	20070103	SSE	5.7	Updated the description of the ZW_SetPromiscuousMode API
11	20070108	HEH	5.5	Added ZW_RediscoveryNeeded description
11	20070108	JFR	3.4 3.2.8, 7.7 and 7.8	Eeprom Loader, Equinox and Zetup RF description removed Production test sample code added
11	20070109	EFH	4.8.6 5.1.1 5.3.3	Multicast capability added for routing slave Multicast capability added for routing- and enhanced slave in table of library functionality. ZW_SendDataMulti parameters changed for optimized multicast handling.
11	20070110	ABR	Several	Added descriptions of functionality related to Zensor Nets in various relevant sections (Many) MVO corrections included
11	20070125	EFH	4.8.9	Added description of ADD_NODE_OPTION_HIGH_POWER for ZW_AddNodeToNetwork
11	20070207	EFH	7.7, 7.8	Added description of sample applications Prod_Test_DUT and Prod_Test_Gen
11	20070207	PSH	7.5, 7.10	Added descriptions of Smokesensor and DoorBell sample applications
11	20070208	PSH	5.8	Added Zensor Net API to API description
11	20070212	EFH	7.7	Added description of flow in Prod_test_DUT sample application program

CONFIDENTIAL

REVISION RECORD

Doc. Rev	Date	By	Pages affected	Brief description of changes
11	20070220	EFH	7.8	Corrected module numbers in block diagram
11	20070228	JFR	5.3.1	ZW0201/ZW0301 ApplicationInitHW parameter added
11	20070309	EFH	5.3.1, 5.12.2	Added description of RF parameter for PA
11	20070312	JFR	Chapter 5	API calls updated and added to table of contents

CONFIDENTIAL

Table of Contents

1 ABBREVIATIONS....1

2 INTRODUCTION....1

2.1 Purpose....1
2.2 Audience and Prerequisites....1

3 SOFTWARE COMPONENTS....2

3.1 Directory Structure....2
3.2 Z-Wave....4
3.2.1 Include....5
3.2.2 Portable Controller....5
3.2.3 Bridge Controller....5
3.2.4 Installer Controller....5
3.2.5 Static Controller....6
3.2.6 Slave....6
3.2.7 Enhanced Slave....6
3.2.8 Production Test DUT....7
3.2.9 Production Test Generator....7
3.2.10 Routing Slave....7
3.2.11 Zensor Net Routing Slave....7
3.3 Product....7
3.3.1 Bin....8
3.3.1.1 Bin_Sensor....8
3.3.1.2 Bin_Sensor_Battery....8
3.3.1.3 Dev_Ctrl....9
3.3.1.4 Doorbell....9
3.3.1.5 LED_Dimmer....10
3.3.1.6 MyProduct....10
3.3.1.7 Prod_Test_DUT....11
3.3.1.8 Prod_Test_Gen....11
3.3.1.9 SerialAPI_Controller_Bridge....12
3.3.1.10 SerialAPI_Controller_Installer....12
3.3.1.11 SerialAPI_Controller_Portable....13
3.3.1.12 SerialAPI_Controller_Static....13
3.3.1.13 SerialAPI_Slave....14
3.3.1.14 Smoke_Sensor....14
3.3.2 Binary Sensor....15
3.3.3 Common....15
3.3.4 Development Controller....15
3.3.5 Door Bell....16
3.3.6 IO_defines....16
3.3.7 LED Dimmer....17
3.3.8 MyProduct....17
3.3.9 Production Test DUT....18
3.3.10 Production Test Generator....18
3.3.11 Serial API....19
3.3.12 Smoke Detector....19
3.3.13 Utilities....20
3.4 Tools....22
3.4.1 Eeprom Loader....22
3.4.2 Equinox....23
3.4.3 ERTT....24
3.4.4 IncDep....24
3.4.5 Intel UPnP....24
3.4.6 Make....24

CONFIDENTIAL

3.4.7 Mergehex....25
3.4.8 Programmer....25
3.4.9 PVT and RF Regulatory....26
3.4.10 Python....27
3.4.11 Zniffer....27
3.5 PC....27
3.5.1 Bin....27
3.5.2 InstallerTool....28
3.5.3 Serial API....28
3.5.4 Source....28
3.5.4.1 Libraries....28
3.5.4.2 Sample Application....29
3.6 Graphics....30

4 Z-WAVE SOFTWARE ARCHITECTURE....31

4.1 Z-Wave System Startup Code....32
4.2 Z-Wave Main Loop....32
4.3 Z-Wave Protocol Layers....32
4.4 Z-Wave Application Layer....32
4.5 Z-Wave Software Timers....35
4.6 Z-Wave Hardware Timers....36
4.7 Z-Wave Hardware Interrupts....36
4.8 Z-Wave Nodes....37
4.8.1 Z-Wave Portable Controller Node....37
4.8.2 Z-Wave Static Controller Node....39
4.8.3 Z-Wave Installer Controller Node....39
4.8.4 Z-Wave Bridge Controller Node....39
4.8.5 Z-Wave Slave Node....40
4.8.6 Z-Wave Routing Slave Node....41
4.8.7 Z-Wave Enhanced Slave Node....42
4.8.8 Z-Wave Zensor Net Routing Slave Node....43
4.8.9 Adding and Removing Nodes to/from the network....45
4.8.10 The Automatic Network Update....47

5 Z-WAVE APPLICATION INTERFACES....48

5.1 Z-Wave Libraries....49
5.1.1 Library Functionality....49
5.1.1.1 Library Functionality without a SUC/SIS....50
5.1.1.2 Library Functionality with a SUC....51
5.1.1.3 Library Functionality with a SIS....52
5.1.1.4 Library Memory Usage....53
5.2 Z-Wave Header Files....54
5.3 Z-Wave Common API....56
5.3.1 Required Application Functions....56
5.3.1.1 ApplicationInitHW....56
5.3.1.2 ApplicationInitSW....57
5.3.1.3 ApplicationTestPoll....58
5.3.1.4 ApplicationPoll....59
5.3.1.5 ApplicationCommandHandler....60
5.3.1.6 ApplicationNodeInformation....61
5.3.1.7 ApplicationSlaveUpdate (All slave libraries)....64
5.3.1.8 ApplicationControllerUpdate (All controller libraries)....65
5.3.1.9 ApplicationSlaveCommandHandler (Bridge Controller library only)....66
5.3.1.10 ApplicationSlaveNodeInformation (Bridge Controller library only)....67
5.3.1.11 ApplicationRfNotify (ZW0301 only)....68
5.3.2 Z-Wave Basis API....69
5.3.2.1 ZW_Poll....69
5.3.2.2 ZW_Random....69
5.3.2.3 ZW_RFPowerLevelSet....70

CONFIDENTIAL

5.3.2.4 ZW_RFPowerLevelGet 71
5.3.2.5 ZW_RediscoveryNeeded (Not Slave and Bridge Controller Library) 72
5.3.2.6 ZW_RequestNetWorkUpdate (Not Slave Library) 73
5.3.2.7 ZW_RFPowerlevelRediscoverySet 75
5.3.2.8 ZW_SetPromiscuousMode (Not Bridge Controller library) 76
5.3.2.9 ZW_SetRFReceiveMode 77
5.3.2.10 ZW_SetSleepMode 78
5.3.2.11 ZW_SendNodeInformation 81
5.3.2.12 ZW_SendTestFrame 82
5.3.2.13 ZW_Type_Library 84
5.3.2.14 ZW_Version 85
5.3.2.15 ZW_VERSION_MAJOR / ZW_VERSION_MINOR / ZW_VERSION_BETA 86
5.3.2.16 ZW_GetProtocolStatus 87
5.3.2.17 ZW_WatchDogEnable 88
5.3.2.18 ZW_WatchDogDisable 88
5.3.2.19 ZW_WatchDogKick 89
5.3.2.20 ZW_SetExtIntLevel (ZW0201/ZW0301 only) 90
5.3.3 Z-Wave Transport API 91
5.3.3.1 ZW_SendData 91
5.3.3.2 ZW_SendDataMeta (ZW0201/ZW0301 only) 95
5.3.3.3 ZW_SendDataMulti 97
5.3.3.4 ZW_SendDataAbort 99
5.3.3.5 ZW_LockRoute 99
5.3.3.6 ZW_SendConst 100
5.3.4 Z-Wave TRIAC API 101
5.3.4.1 TRIAC_Init 101
5.3.4.2 TRIAC_SetDimLevel 103
5.3.4.3 TRIAC_Off 103
5.3.5 Z-Wave Timer API 104
5.3.5.1 TimerStart 104
5.3.5.2 TimerRestart 105
5.3.5.3 TimerCancel 105
5.3.6 Z-Wave PWM API 106
5.3.6.1 ZW_PWMSetup 106
5.3.6.2 ZW_PWMPrescale 108
5.3.6.3 ZW_PWMClearInterrupt 109
5.3.6.4 ZW_PWMEnable 109
5.3.7 Z-Wave Memory API 110
5.3.7.1 MemoryGetID 110
5.3.7.2 MemoryGetByte 111
5.3.7.3 MemoryPutByte 112
5.3.7.4 MemoryGetBuffer 113
5.3.7.5 MemoryPutBuffer 114
5.3.7.6 ZW_EepromInit 115
5.3.7.7 ZW_MemoryFlush 115
5.3.8 Z-Wave ADC API 116
5.3.8.1 ADC_On (ZW0102 only) 116
5.3.8.2 ADC_Off 116
5.3.8.3 ADC_Start 116
5.3.8.4 ADC_Stop 117
5.3.8.5 ADC_Init 118
5.3.8.6 ADC_SelectPin 122
5.3.8.7 ADC_Buf (ZW0201/ZW0301 only) 123
5.3.8.8 ADC_SetAZPL (ZW0201/ZW0301 only) 124
5.3.8.9 ADC_SetResolution (ZW0201/ZW0301 only) 124
5.3.8.10 ADC_SetThresMode (ZW0201/ZW0301 only) 125
5.3.8.11 ADC_SetThres 126
5.3.8.12 ADC_Int 127
5.3.8.13 ADC_IntFlagClr 127
5.3.8.14 ADC_SetSamplingRate (ZW0102 only) 128
5.3.8.15 ADC_GetRes 129
5.3.9 Z-Wave Power API 130
5.3.9.1 PWR_SetStopMode (ZW0102 only) 130

CONFIDENTIAL

5.3.9.2 PWR_Clk_PD (ZW0102 only) 131
5.3.9.3 PWR_Clk_Pup (ZW0102 only) 131
5.3.9.4 PWR_Select_Clk (ZW0102 only) 132
5.3.9.5 ZW_SetWutTimeout (ZW0201/ZW0301 only) 132
5.3.9.6 ZW_IsWutKicked (ZW0201/ZW0301 only) 133
5.3.10 UART interface API 133
5.3.10.1 UART_Init 133
5.3.10.2 UART_RecStatus 134
5.3.10.3 UART_RecByte 134
5.3.10.4 UART_SendStatus 135
5.3.10.5 UART_SendByte 135
5.3.10.6 UART_SendNum 136
5.3.10.7 UART_SendStr 136
5.3.10.8 UART_SendNL 136
5.3.10.9 UART_Enable (ZW0201/ZW0301 only) 137
5.3.10.10 UART_Disable (ZW0201/ZW0301 only) 137
5.3.10.11 UART_ClearTx (ZW0201/ZW0301 only) 137
5.3.10.12 UART_ClearRx (ZW0201/ZW0301 only) 138
5.3.10.13 UART_Write (ZW0201/ZW0301 only) 138
5.3.10.14 UART_Read (ZW0201/ZW0301 only) 139
5.3.10.15 Serial debug output. 139
5.3.11 Z-Wave Real Time Clock API (ZW0102 only) 140
5.3.11.1 ClockSet 141
5.3.11.2 ClockGet 142
5.3.11.3 ClockCmp 142
5.3.11.4 RTCTimerCreate 143
5.3.11.5 RTCTimerRead 144
5.3.11.6 RTCTimerDelete 145
5.3.12 Z-Wave Node Mask API 146
5.3.12.1 ZW_NodeMaskSetBit 146
5.3.12.2 ZW_NodeMaskClearBit 146
5.3.12.3 ZW_NodeMaskClear 147
5.3.12.4 ZW_NodeMaskBitsIn 147
5.3.12.5 ZW_NodeMaskNodeIn 148
5.4 Z-Wave Controller API 149
5.4.1 ZW_AddNodeToNetwork 149
5.4.2 ZW_RemoveNodeFromNetwork 151
5.4.3 ZW_ControllerChange 153
5.4.4 ZW_SetLearnMode 155
5.4.5 ZW_GetControllerCapabilities 157
5.4.6 ZW_GetNodeProtocolInfo 158
5.4.7 ZW_SetDefault 159
5.4.8 ZW_ReplicationSend 160
5.4.9 ZW_ReplicationReceiveComplete 161
5.4.10 ZW_AssignReturnRoute 162
5.4.11 ZW_DeleteReturnRoute 163
5.4.12 ZW_RemoveFailedNodeID 164
5.4.13 ZW_isFailedNode 165
5.4.14 ZW_IsPrimaryCtrl 166
5.4.15 ZW_ReplaceFailedNode 166
5.4.16 ZW_GetNeighborCount 169
5.4.17 ZW_AreNodesNeighbours 170
5.4.18 ZW_RequestNodeNeighborUpdate 171
5.4.19 ZW_GetRoutingInfo 172
5.4.20 ZW_GetSUCNodeID 173
5.4.21 ZW_SetSUCNodeID 173
5.4.22 ZW_SendSUCID 176
5.4.23 ZW_AssignSUCReturnRoute 177
5.4.24 ZW_DeleteSUCReturnRoute 178
5.4.25 ZW_RequestNodeInfo 179

CONFIDENTIAL

5.5 Z-Wave Static Controller API 180
5.5.1 ZW_EnableSUC 180
5.5.2 ZW_CreateNewPrimaryCtrl 181
5.6 Z-Wave Bridge Controller API 183
5.6.1 ZW_SendSlaveData 183
5.6.2 ZW_SendSlaveNodeInformation 185
5.6.3 ZW_SetSlaveLearnMode 186
5.6.4 ZW_IsVirtualNode 189
5.6.5 ZW_GetVirtualNodes 190
5.7 Z-Wave Installer Controller API 191
5.7.1 zwTransmitCount 191
5.7.2 ZW_StoreNodeInfo 192
5.7.3 ZW_StoreHomeID 193
5.8 Z-Wave Slave API 194
5.8.1 ZW_SetLearnMode 194
5.8.2 ZW_SetDefault 195
5.8.3 ZW_Support9600Only (ZW0201/ZW0301 only) 196
5.9 Z-Wave Routing and Enhanced Slave API 197
5.9.1 ZW_RequestNewRouteDestinations 197
5.9.2 ZW_IsNodeWithinDirectRange 198
5.9.3 ZW_RequestNodeInfo 199
5.10 Z-Wave Zensor Net Routing Slave API 200
5.10.1 ZW_ZensorNetBind 200
5.10.2 ZW_ZensorSendDataFlood 201
5.10.3 ZW_ZensorGetID 202
5.10.4 ZW_ZensorSetDefault 202
5.11 Serial Command Line Debugger 203
5.11.1 ZW_DebugInit 205
5.11.2 ZW_DebugPoll 205
5.12 RF Settings in App_RFSetup.a51 file 206
5.12.1 ZW0102 RF parameters 206
5.12.2 ZW0201/ZW0301 RF parameters 207

6 HARDWARE SUPPORT DRIVERS 208

6.1 Hardware Pin Definitions 208

7 APPLICATION SAMPLE CODE 211

7.1 Building ZW0x0x Sample Code 211
7.1.1 MK.BAT 212
7.1.2 Makefiles 215
7.2 Binary Sensor Sample Code 217
7.2.1 Bin_Sensor Files 218
7.3 Binary Sensor Battery Sample Code 221
7.3.1 Bin_Sensor_Battery Files 222
7.4 Development Controller Sample Code 225
7.4.1 Dev_Ctrl Files. 225
7.5 Door Bell Sample Code 227
7.5.1 User interface 227
7.5.2 Door Bell Files 227
7.6 LED Dimmer Sample Code 229
7.6.1 Build Targets 229
7.6.2 LED_Dimmer Files 230
7.7 Production Test DUT 232
7.7.1 Production Test DUT Files 233
7.8 Production Test Generator 234
7.8.1 Production Test Generator Files 236
7.9 Serial API Embedded Sample Code 237

CONFIDENTIAL

7.9.1 Supported API Calls 237
7.9.2 Implementation 237
7.9.2.1 Frame Layout 237
7.9.2.2 Frame Flow 240
7.9.2.3 Error handling 242
7.9.2.4 Restrictions on functions using buffers 243
7.9.2.5 Serial API capabilities 243
7.9.2.6 Serial API Softreset 244
7.9.2.7 Build Targets 245
7.9.2.8 Serial API Files 246
7.10 Smoke Detector Sample Code 248
7.10.1 User interface 248
7.10.2 Smoke Detector Files 248
7.10.3 PC based Controller Sample Application 250
7.10.4 PC based Installer Tool Sample Application 250
7.10.5 PC based Z-Wave Bridge Sample Application 251
8 REQUIRED DEVELOPMENT COMPONENTS 252
8.1 Software development components 252
8.2 ZW0102/ZW0201/ZW0301 single chip programmer 252
8.3 Hardware development components for ZW0102 253
8.4 Hardware development components for ZW0201 254
8.5 Hardware development components for ZW0301 254
8.6 ZW0102/ZW0201/ZW0301 lock bit settings 255
8.7 External EEPROM initialization 256
9 APPLICATION NOTE: SUC/SIS IMPLEMENTATION 257
9.1 Implementing SUC support In All Nodes 257
9.2 Static Controllers 257
9.2.1 Request For Becoming SUC 257
9.2.1.1 Request For Becoming a SUC Node ID Server (SIS) 257
9.2.2 Updates From The Primary Controller 258
9.2.3 Assigning SUC Routes To Routing Slaves 258
9.2.4 Receiving Requests for Network Updates 258
9.2.5 Receiving Requests for new Node ID (SIS only) 258
9.3 The Primary Controller 258
9.4 Secondary Controllers 259
9.4.1 Knowing The SUC 259
9.4.2 Asking For And Receiving Updates 259
9.5 Inclusion Controllers 260
9.6 Routing Slaves 260
10 APPLICATION NOTE: INCLUSION/EXCLUSION IMPLEMENTATION 262
10.1 Including new nodes to the network 262
10.2 Excluding nodes from the network 267
11 APPLICATION NOTE: CONTROLLER SHIFT IMPLEMENTATION 270
12 APPLICATION NOTE: ZENSOR NET BINDING AND FLOODING 271
13 REFERENCES 272
INDEX 273

List of Figures

Figure 1 Software architecture 31

Figure 2 Multiple copies of the same Set frame 33
Figure 3 Multiple copies of the same Get/Report frame 34
Figure 4 Simultaneous communication to a number of nodes 35
Figure 5 Slave node architecture 40
Figure 6 Enhanced slave node architecture 42
Figure 7 Node Information frame format 63
Figure 8 Node Information frame without command classes format 158
Figure 9 Configuring environment variables 212
Figure 10 Building sample applications 213
Figure 11 Possible sample application targets 214
Figure 12, Prod_Test_DUT test program flow 232
Figure 13, Z-Wave test generator 234
Figure 14 ZW0102 Controller/Slave Unit 253
Figure 15 ZW0102 Development Controller Unit 253
Figure 16 Inclusion of a node having a SUC in the network 258
Figure 17 Requesting network updates from a SUC in the network 259
Figure 18 Inclusion of a node having a SIS in the network 260
Figure 19 Lost routing slave frame flow 261
Figure 20 Node inclusion frame flow 263
Figure 21 Node exclusion frame flow 268
Figure 22 Controller shift frame flow 270

List of Tables

Table 1, ZW0102/ZW0201/ZW0301 hardware timer allocation 36
Table 2, ZW0102/ZW0201/ZW0301 Application ISR avialability 36
Table 3. Controller functionality 46
Table 4. Library functionality 49
Table 5. Library functionality without a SUC/SIS 50
Table 6. Library functionality with a SUC 51
Table 7. Library functionality with a SIS 52
Table 8. ApplicationPoll frequency 59
Table 9. App_RFSetup.a51 module definitions for ZW0102 206
Table 10. App_RFSetup.a51 module definitions for ZW0201/ZW0301 207
Table 11. Lock bits settings during development 255
Table 12. Lock bits settings in end products 255

CONFIDENTIAL

1 ABBREVIATIONS

Abbreviation	Explanation
ACK	Acknowledge
ANZ	Australia/New Zealand
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
DLL	Dynamic Link Library
DUT	Device Under Test
ERTT	Enhanced Reliability Test tool
EU	Europe
GNU	An organization devoted to the creation and support of Open Source software
HK	Hong Kong
HW	Hardware
ISR	Interrupt Service Routines
LRC	Longitudinal Redundancy Check
NAK	Not Acknowledged
PWM	Pulse Width Modulator
R&D	Research and Development
RF	Radio Frequency
RTC	Real Time Clock
SFF	Small Form Factor
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 ZW0102/ZW0201/ZW0301 Z-Wave module. The document is also API reference guide for programmers.

2.2 Audience and Prerequisites

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

CONFIDENTIAL

3 SOFTWARE COMPONENTS

The Z-Wave development software packet consists of a protocol part, sample applications and a number of tools used for developing and building the sample code.

3.1 Directory Structure

The development software is organized in the following directory structure:

```
/
        - Graphics
        - PC
                - Bin
                        - InstallerTool
                        - ZWavePCController
                        - ZWaveUPnPBridge
                - InstallerTool
                - SerialAPI
                - Source
                        - Libraries
                                - ITransportLayer
                                - ZWave
                                - ZWaveCmdClass
                                - ZWaveRS232
                        - SampleApplications
                                - Lib
                                - ZensysClasses
                                - ZWavePCController
                                - ZWaveUPnPBridge
```

CONFIDENTIAL

```
- Product
        - Bin
                - Bin_Sensor
                - Bin_Sensor_Battery
                - Dev_Ctrl
                - Doorbell
                - LED_Dimmer
                - Prod_Test_DUT
                - Prod_Test_Gen
                - SerialAPI_Controller_Bridge
                - SerialAPI_Controller_Installer
                - SerialAPI_Controller_Portable
                - SerialAPI_Controller_Static
                - SerialAPI_Slave
                - Smoke_Sensor
        - Bin_Sensor
        - Bin_Sensor_Battery
        - Common
        - Dev_Ctrl
        - Doorbell
        - IO_Defines
        - LED_Dimmer
        - MyProduct
        - Prod_Test_DUT
        - Prod_Test_Gen
        - SerialAPI
        - Smoke_Sensor
        - Util_Func

- Tools
        - ERTT
                - PC
                - Z-Wave_Firmware
        - Equinox
        - IncDep
        - Intel_UPnP
        - Make
        - Mergehex
        - Programmer
                - PC
                - ZDP02A_Firmware
        - PVT_and_RF_regulatory
        - Python
        - Zniffer
                - PC
                - Z-Wave_Firmware
```

CONFIDENTIAL

```
- Z-Wave
        - include
        - lib
                - controller_bridge_ZW010x
                - controller_bridge_ZW020x
                - controller_bridge_ZW030x
                - controller_installer_ZW010x
                - controller_installer_ZW020x
                - controller_installer_ZW030x
                - controller_portable_ZW010x
                - controller_portable_ZW020x
                - controller_portable_ZW030x
                - controller_static_ZW010x
                - controller_static_ZW020x
                - controller_static_ZW030x
                - slave_enhanced_ZW010x
                - slave_enhanced_ZW020x
                - slave_enhanced_ZW030x
                - slave_prodtest_dut_ZW010x
                - slave_prodtest_dut_ZW020x
                - slave_prodtest_dut_ZW030x
                - slave_prodtest_gen_ZW010x
                - slave_prodtest_gen_ZW020x
                - slave_prodtest_gen_ZW030x
                - slave_routing_ZW010x
                - slave_routing_ZW020x
                - slave_routing_ZW030x
                - slave_sensor_ZW020x
                - slave_sensor_ZW030x
                - slave_ZW010x
                - slave_ZW020x
                - slave_ZW030x
```

This directory structure contains all the tools and sample applications needed, except the recommended Keil software which must be purchased separately. More information about where and how to buy the Keil software development components are described in paragraph 8.1.

Note! It is recommended to leave the directory structure as is due to compiler and linker issues.

The majority of the above mentioned Z-Wave specific tools and sample application are briefly described in the following sections.

3.2 Z-Wave

The Z-Wave header files and libraries are the software files needed for building a Z-Wave enabled product. The files are organized in directories used for building Z-Wave controllers and slaves respectively.

CONFIDENTIAL

3.2.1 Include

The include directory contain all the header files ZW_xxx_api.h with declarations of API calls etc. The header files are the same for ZW0102, ZW0201 and ZW0301. Refer to chapter 5.2 regarding a detailed description.

3.2.2 Portable Controller

The lib\controller_ZW0x0x directory contains all files needed for building a Z-Wave controller application. The directory contains the following files:

ZW_controller_portable_zw010xs.lib **ZW_controller_portable_zw020xs.lib** **ZW_controller_portable_zw030xs.lib**	These files are the compiled Z-Wave protocol and API library for the ZW0102/ZW0201/ZW0301 based modules that a Z-Wave portable controller application should be linked together with. The library requires a ZM12xxRE/ZM21xxE/ZM31xxC-E module.
extern_eep.hex	This file contains the external EEPROM data on the ZM12xxRE/ZM21xxE/ZM31xxC-E module. The external EEPROM must only be initialized once.

3.2.3 Bridge Controller

The lib\controller_bridge_ZW0x0x directory contains all files needed for building a Z-Wave bridge controller application. The directory contains the following files:

ZW_controller_bridge_zw010xs.lib **ZW_controller_bridge_zw020xs.lib** **ZW_controller_bridge_zw030xs.lib**	These files are the compiled Z-Wave protocol and API library for the ZW0102/ZW0201/ZW0301 based modules that a Z-Wave controller bridge application should be linked together with. The library requires a ZM12xxRE/ZM21xxE/ZM31xxC-E module.
extern_eep.hex	This file contains the external EEPROM data on the ZM12xxRE/ZM21xxE/ZM31xxC-E module. The external EEPROM must only be initialized once.

3.2.4 Installer Controller

The lib\controller_installer_ZW0x0x directory contains all files needed for building a Z-Wave installer controller application. The directory contains the following files:

ZW_controller_installer_ZW010xs.lib **ZW_controller_installer_ZW020xs.lib** **ZW_controller_installer_ZW030xs.lib**	These files are the compiled Z-Wave protocol and API library for the ZW0102/ZW0201/ZW0301 based modules that a Z-Wave installer application should be linked together with. The library requires a ZM12xxRE/ZM21xxE/ZM31xxC-E module.
extern_eep.hex	This file contains the external EEPROM data on the ZM12xxRE/ZM21xxE/ZM31xxC-E module. The external

CONFIDENTIAL

EEPROM must only be initialized once.

3.2.5 Static Controller

The lib\controller_static_ZW0x0x directory contains all files needed for building a Z-Wave static controller application. The directory contains the following files:

ZW_controller_static_zw010xs.lib **ZW_controller_static_zw020xs.lib** **ZW_controller_static_zw030xs.lib**	These files are the compiled Z-Wave protocol and API library for the ZW0102/ZW0201/ZW0301 based modules that a Z-Wave static controller application should be linked together with. The library requires a ZM12xxRE/ZM21xxE/ZM31xxC-E module.
Extern_eep.hex	This file contains the external EEPROM data on the ZM12xxRE/ZM21xxE/ZM31xxC-E module. The external EEPROM must only be initialized once.

3.2.6 Slave

The lib\slave_ZW0x0x directory contains all files needed for building a Z-Wave routing slave node application. The directory contains the following files:

ZW_slave_zw010xs.lib **ZW_slave_zw020xs.lib** **ZW_slave_zw030xs.lib**	These files are the compiled Z-Wave protocol and API library for the ZW0102/ZW0201/ZW0301 based modules that a Z-Wave slave application should be linked together with.

3.2.7 Enhanced Slave

The lib\slave_enhanced_ZW0x0x directory contains all files needed for building a Z-Wave enhanced slave node application. The directory contains the following files:

ZW_slave_enhanced_ZW010xs.lib **ZW_slave_enhanced_ZW020xs.lib** **ZW_slave_enhanced_ZW030xs.lib**	These files are the compiled Z-Wave protocol and API library for the ZW0102/ZW0201/ZW0301 based modules that a Z-Wave enhanced slave application should be linked together with. The library requires a ZM12xxRE/ZM21xxE/ZM31xxC-E module.
Extern_eep.hex	This file contains the external EEPROM data without home ID on the ZM12xxRE/ZM21xxE/ZM31xxC-E module. The external EEPROM must only be initialized once.

CONFIDENTIAL

3.2.8 Production Test DUT

The lib\slave_prodtest_dut_ZW0x0x directory contains all files needed for building a production test DUT application on a Z-Wave module. The directory contains the following files:

ZW_slave_prodtest_dut_ZW010xs.lib **ZW_slave_prodtest_dut_ZW020xs.lib** **ZW_slave_prodtest_dut_ZW030xs.lib**	These files are the compiled Z-Wave protocol and API library for the ZW0102/ZW0201/ZW0301 based modules that a Z-Wave production test DUT application should be linked together with.

3.2.9 Production Test Generator

The lib\slave_prodtest_ZW0x0x directory contains all files needed for building a production test generator application on a Z-Wave module. The directory contains the following files:

ZW_slave_prodtest_gen_ZW010xs.lib **ZW_slave_prodtest_gen_ZW020xs.lib** **ZW_slave_prodtest_gen_ZW030xs.lib**	These files are the compiled Z-Wave protocol and API library for the ZW0102/ZW0201/ZW0301 based modules that a Z-Wave production test generator application should be linked together with.

3.2.10 Routing Slave

The lib\slave_routing_ZW0x0x directory contains all files needed for building a Z-Wave routing slave node application on a Z-Wave module. The directory contains the following files:

ZW_slave_routing_ZW010xs.lib **ZW_slave_routing_ZW020xs.lib** **ZW_slave_routing_ZW030xs.lib**	These files are the compiled Z-Wave protocol and API library for the ZW0102/ZW0201/ZW0301 based modules that a Z-Wave routing slave application should be linked together with.

3.2.11 Zensor Net Routing Slave

The lib\slave_sensor_ZW0x0x directory contains all files needed for building a Z-Wave Zensor Net routing slave node application on a Z-Wave module. The directory contains the following files:

ZW_slave_sensor_ZW020xs.lib **ZW_slave_sensor_ZW030xs.lib**	These files are the compiled Z-Wave protocol and API library for the ZW0201/ZW0301 based modules that a Z-Wave Zensor Net routing slave application should be linked together with.

3.3 Product

The Product directory contains Z-Wave sample applications for a number of different product examples. Both source code and precompiled files ready for download are supplied.

CONFIDENTIAL

Each directory contains the necessary files for creating EU (868.42 MHz), US (908.42 MHz), ANZ (921.42 MHz), and HK (919.82 MHz) products.

3.3.1 Bin

The Product\Bin directory structure contains the precompiled code of the Z-Wave sample applications and the hex files needed to download to the Z-Wave ZW0x0x single chip via the Z-Wave Programmer.

3.3.1.1 Bin_Sensor

The Product\Bin\Bin_Sensor directory contains all files needed for running a binary sensor sample application on a Z-Wave module. The directory contains the following files:

binsensor_ZW010x_EU.hex **binsensor_ZW010x_US.hex**	The compiled and linked binary sensor sample application for the EU and US versions of the ZW0102 based module.
binsensor_ZW020x_EU.hex **binsensor_ZW020x_US.hex** **binsensor_ZW020x_ANZ.hex** **binsensor_ZW020x_HK.hex**	The compiled and linked binary sensor sample application for the ANZ, EU, HK and US versions of the ZW0201 based module.
binsensor_ZW030x_EU.hex **binsensor_ZW030x_US.hex** **binsensor_ZW030x_ANZ.hex** **binsensor_ZW030x_HK.hex**	The compiled and linked binary sensor sample application for the ANZ, EU, HK and US versions of the ZW0301 based module.

3.3.1.2 Bin_Sensor_Battery

The Product\Bin\Bin_Sensor_Battery directory contains all files needed for running a battery operated binary sensor sample application on a Z-Wave module. The directory contains the following files:

binsensor_Battery_ZW010x_EU.hex **binsensor_Battery_ZW010x_US.hex**	The compiled and linked battery operated binary sensor sample application for the EU and US versions of the ZW0102 based module.
binsensor_Battery_ZW020x_EU.hex **binsensor_Battery_ZW020x_US.hex** **binsensor_Battery_ZW020x_ANZ.hex** **binsensor_Battery_ZW020x_HK.hex**	The compiled and linked battery operated binary sensor sample application for the ANZ, EU, HK and US versions of the ZW0201 based module.
binsensor_Battery_ZW030x_EU.hex **binsensor_Battery_ZW030x_US.hex** **binsensor_Battery_ZW030x_ANZ.hex** **binsensor_Battery_ZW030x_HK.hex**	The compiled and linked battery operated binary sensor sample application for the ANZ, EU, HK and US versions of the ZW0301 based module.

CONFIDENTIAL

3.3.1.3 Dev_Ctrl

The Product\Bin\Dev_Ctrl directory contains all files needed for running a development controller sample application on a Z-Wave module. The directory contains the following files:

extern_eep.hex	This file contains the external EEPROM data with home ID on the ZM12xxRE/ZM21xxE/ZM31xxC-E module. The external EEPROM must only be initialized once.
dev_ctrl_ZW010x_EU.hex **dev_ctrl_ZW010x_US.hex**	The compiled and linked development controller sample application for the EU and US versions of the ZW0102 based module.
dev_ctrl_ZW020x_EU.hex **dev_ctrl_ZW020x_US.hex** **dev_ctrl_ZW020x_ANZ.hex** **dev_ctrl_ZW020x_HK.hex**	The compiled and linked development controller sample application for the ANZ, EU, HK and US versions of the ZW0201 based module.
dev_ctrl_ZW030x_EU.hex **dev_ctrl_ZW030x_US.hex** **dev_ctrl_ZW030x_ANZ.hex** **dev_ctrl_ZW030x_HK.hex**	The compiled and linked development controller sample application for the ANZ, EU, HK and US versions of the ZW0301 based module.

3.3.1.4 Doorbell

The Product\Bin\Doorbell directory contains all files needed for running a bell sample application on a Z-Wave module. The development controller application is used as button in the doorbell application. The directory contains the following files:

doorbell_bell_ZW020x_EU.hex **doorbell_bell_ZW020x_US.hex** **doorbell_bell_ZW020x_ANZ.hex** **doorbell_bell_ZW020x_HK.hex**	The compiled and linked bell sample application for the ANZ, EU, HK and US versions of the ZW0201 based module.
doorbell_bell_ZW030x_EU.hex **doorbell_bell_ZW030x_US.hex** **doorbell_bell_ZW030x_ANZ.hex** **doorbell_bell_ZW030x_HK.hex**	The compiled and linked bell sample application for the ANZ, EU, HK and US versions of the ZW0301 based module.

CONFIDENTIAL

3.3.1.5 LED_Dimmer

The Product\Bin\LED_Dimmer directory contains all files needed for running a LED dimmer sample application on a Z-Wave module. The directory contains the following files:

leddimmer_ZW010x_EU.hex **leddimmer_ZW010x_US.hex**	The compiled and linked LED dimmer sample application for the EU and US versions of the ZM1220 module.
leddimmer_ZW010x_sff_EU.hex **leddimmer_ZW010x_sff_US.hex**	The compiled and linked LED dimmer sample application for the EU and US versions of the ZM1206 module. The hex file is different because the ZM1206 use the IO10 pin to detect the production test mode and the ZM1220 module use the ZEROX pin.
leddimmer_ZW020x_EU.hex **leddimmer_ZW020x_US.hex** **leddimmer_ZW020x_ANZ.hex** **leddimmer_ZW020x_HK.hex**	The compiled and linked LED dimmer sample application for the ANZ, EU, HK and US versions of the ZW0201 based module.
leddimmer_ZW030x_EU.hex **leddimmer_ZW030x_US.hex** **leddimmer_ZW030x_ANZ.hex** **leddimmer_ZW030x_HK.hex**	The compiled and linked LED dimmer sample application for the ANZ, EU, HK and US versions of the ZW0301 based module.

3.3.1.6 MyProduct

No hex files available.

CONFIDENTIAL

3.3.1.7 Prod_Test_DUT

The Product\Bin\Prod_Test_DUT directory contains all files needed for running a production test DUT sample application on a Z-Wave module. The directory contains the following files:

prod_test_dut_ZW010x_EU.hex **prod_test_dut_ZW010x_US.hex**	The compiled and linked development controller sample application for the EU and US versions of the ZW0102 based module.
prod_test_dut_ZW010x_sff_EU.hex **prod_test_dut_ZW010x_sff_US.hex**	The compiled and linked LED dimmer sample application for the EU and US versions of the ZM1206 module. The hex file is different because the ZM1206 use the IO10 pin to detect the production test mode and the ZM1220 module use the ZEROX pin.
prod_test_dut_ZW020x_ANZ.hex **prod_test_dut_ZW020x_EU.hex** **prod_test_dut_ZW020x_HK.hex** **prod_test_dut_ZW020x_US.hex**	The compiled and linked development controller sample application for the ANZ, EU, HK and US versions of the ZW0201 based module.
prod_test_dut_ZW030x_ANZ.hex **prod_test_dut_ZW030x_EU.hex** **prod_test_dut_ZW030x_HK.hex** **prod_test_dut_ZW030x_US.hex**	The compiled and linked development controller sample application for the ANZ, EU, HK and US versions of the ZW0301 based module.

3.3.1.8 Prod_Test_Gen

The Product\Bin\Prod_Test_Gen directory contains all files needed for running a production test generator sample application on a Z-Wave module. The directory contains the following files:

prod_test_gen_ZW010x_EU.hex **prod_test_gen_ZW010x_US.hex**	The compiled and linked production test generator sample application for the EU and US versions of the ZW0102 based module.
prod_test_gen_ZW020x_ANZ.hex **prod_test_gen_ZW020x_EU.hex** **prod_test_gen_ZW020x_HK.hex** **prod_test_gen_ZW020x_US.hex**	The compiled and linked production test generator sample application for the ANZ, EU, HK and US versions of the ZW0201 based module.
prod_test_gen_ZW030x_ANZ.hex **prod_test_gen_ZW030x_EU.hex** **prod_test_gen_ZW030x_HK.hex** **prod_test_gen_ZW030x_US.hex**	The compiled and linked production test generator sample application for the ANZ, EU, HK and US versions of the ZW0301 based module.

CONFIDENTIAL

3.3.1.9 SerialAPI_Controller_Bridge

The Product\Bin\SerialAPI_Controller_Bridge directory contains all files needed for running a serial API based bridge controller sample application on a Z-Wave module. The directory contains the following files:

extern_eep.hex	This file contains the external EEPROM data with home ID on the ZM12xxRE/ZM21xxE/ZM31xxC-E module. The external EEPROM must only be initialized once.
serialapi_controller_bridge_ZW010x_EU.hex **serialapi_controller_bridge_ZW010x_US.hex**	The compiled and linked serial API based static controller sample application for the EU and US versions of the ZW0102 based module.
serialapi_controller_bridge_ZW020x_EU.hex **serialapi_controller_bridge_ZW020x_US.hex** **serialapi_controller_bridge_ZW020x_ANZ.hex** **serialapi_controller_bridge_ZW020x_HK.hex**	The compiled and linked serial API based static controller sample application for the ANZ, EU, HK and US versions of the ZW0201 based module.
serialapi_controller_bridge_ZW030x_EU.hex **serialapi_controller_bridge_ZW030x_US.hex** **serialapi_controller_bridge_ZW030x_ANZ.hex** **serialapi_controller_bridge_ZW030x_HK.hex**	The compiled and linked serial API based static controller sample application for the ANZ, EU, HK and US versions of the ZW0301 based module.

3.3.1.10 SerialAPI_Controller_Installer

The Product\Bin\SerialAPI_Controller_Installer directory contains all files needed for running a serial API based installer controller sample application on a Z-Wave module. The directory contains the following files:

extern_eep.hex	This file contains the external EEPROM data with home ID on the ZM12xxRE/ZM21xxE/ZM31xxC-E module. The external EEPROM must only be initialized once.
serialapi_controller_installer_ZW010x_EU.hex **serialapi_controller_installer_ZW010x_US.hex**	The compiled and linked serial API based static controller sample application for the EU and US versions of the ZW0102 based module.
serialapi_controller_installer_ZW020x_EU.hex **serialapi_controller_installer_ZW020x_US.hex** **serialapi_controller_installer_ZW020x_ANZ.hex** **serialapi_controller_installer_ZW020x_HK.hex**	The compiled and linked serial API based static controller sample application for the ANZ, EU, HK and US versions of the ZW0201 based module.
serialapi_controller_installer_ZW030x_EU.hex **serialapi_controller_installer_ZW030x_US.hex** **serialapi_controller_installer_ZW030x_ANZ.hex** **serialapi_controller_installer_ZW030x_HK.hex**	The compiled and linked serial API based static controller sample application for the ANZ, EU, HK and US versions of the ZW0301 based module.

CONFIDENTIAL

3.3.1.11 SerialAPI_Controller_Portable

The Product\Bin\SerialAPI_Controller_Portable directory contains all files needed for running a serial API based portable controller sample application on a Z-Wave module. The directory contains the following files:

extern_eep.hex	This file contains the external EEPROM data with home ID on the ZM12xxRE/ZM21xxE/ZM31xxC-E module. The external EEPROM must only be initialized once.
serialapi_controller_portable_ZW010x_EU.hex **serialapi_controller_portable_ZW010x_US.hex**	The compiled and linked serial API based static controller sample application for the EU and US versions of the ZW0102 based module.
serialapi_controller_portable_ZW020x_EU.hex **serialapi_controller_portable_ZW020x_US.hex** **serialapi_controller_portable_ZW020x_ANZ.hex** **serialapi_controller_portable_ZW020x_HK.hex**	The compiled and linked serial API based static controller sample application for the ANZ, EU, HK and US versions of the ZW0201 based module.
serialapi_controller_portable_ZW030x_EU.hex **serialapi_controller_portable_ZW030x_US.hex** **serialapi_controller_portable_ZW030x_ANZ.hex** **serialapi_controller_portable_ZW030x_HK.hex**	The compiled and linked serial API based static controller sample application for the ANZ, EU, HK and US versions of the ZW0301 based module.

3.3.1.12 SerialAPI_Controller_Static

The Product\Bin\SerialAPI_Controller_Static directory contains all files needed for running a serial API based static controller sample application on a Z-Wave module. The directory contains the following files:

extern_eep.hex	This file contains the external EEPROM data with home ID on the ZM12xxRE/ZM21xxE/ZM31xxC-E module. The external EEPROM must only be initialized once.
serialapi_controller_static_ZW010x_EU.hex **serialapi_controller_static_ZW010x_US.hex**	The compiled and linked serial API based static controller sample application for the EU and US versions of the ZW0102 based module.
serialapi_controller_static_ZW020x_EU.hex **serialapi_controller_static_ZW020x_US.hex** **serialapi_controller_static_ZW020x_ANZ.hex** **serialapi_controller_static_ZW020x_HK.hex**	The compiled and linked serial API based static controller sample application for the ANZ, EU, HK and US versions of the ZW0201 based module.
serialapi_controller_static_ZW030x_EU.hex **serialapi_controller_static_ZW030x_US.hex** **serialapi_controller_static_ZW030x_ANZ.hex** **serialapi_controller_static_ZW030x_HK.hex**	The compiled and linked serial API based static controller sample application for the ANZ, EU, HK and US versions of the ZW0301 based module.

CONFIDENTIAL

3.3.1.13 SerialAPI_Slave

The Product\Bin\SerialAPI_Slave directory contains all files needed for running a serial API based slave sample application on a Z-Wave module. The directory contains the following files:

serialapi_slave_ZW010x_EU.hex **serialapi_slave_ZW010x_US.hex**	The compiled and linked serial API based slave sample application for the EU and US versions of the ZW0102 based module.
serialapi_slave_ZW020x_EU.hex **serialapi_slave_ZW020x_US.hex** **serialapi_slave_ZW020x_ANZ.hex** **serialapi_slave_ZW020x_HK.hex**	The compiled and linked serial API based slave sample application for the ANZ, EU, HK and US versions of the ZW0201 based module.
serialapi_slave_ZW030x_EU.hex **serialapi_slave_ZW030x_US.hex** **serialapi_slave_ZW030x_ANZ.hex** **serialapi_slave_ZW030x_HK.hex**	The compiled and linked serial API based slave sample application for the ANZ, EU, HK and US versions of the ZW0301 based module.

3.3.1.14 Smoke_Sensor

The Product\Bin\Smoke_Sensor directory contains all files needed for running a smoke sensor sample application on a Z-Wave module. The directory contains the following files:

smokesensor_ZW020x_EU.hex **smokesensor_ZW020x_US.hex** **smokesensor_ZW020x_ANZ.hex** **smokesensor_ZW020x_HK.hex**	The compiled and linked smoke sensor sample application for the ANZ, EU, HK and US versions of the ZW0201 based module.
smokesensor_ZW030x_EU.hex **smokesensor_ZW030x_US.hex** **smokesensor_ZW030x_ANZ.hex** **smokesensor_ZW030x_HK.hex**	The compiled and linked smoke sensor sample application for the ANZ, EU, HK and US versions of the ZW0301 based module.

CONFIDENTIAL

3.3.2 Binary Sensor

The Product\Bin_Sensor directory contains sample source code for a binary sensor and battery operated binary sensor application, which uses the 4 LEDs and the button on the interface module (see section 7.2.1 and 7.3.1).

Bin_Sensor.c **Bin_Sensor.h**	These files contain the source code for the binary sensor's application state machine. The common API functions such as **ApplicationInitHW**, **ApplicationInitSW**, **ApplicationNodeInformation**, **ApplicationPoll**, **ApplicationSlaveUpdate**, **ApplicationRfNotify** and **ApplicationCommandHandler** are defined here.
eeprom.h	This header file defines the addresses where application data are stored in the external EEPROM.
MK.BAT	Batch file used to build ZW0102/ZW0201/ZW0301 based sample applications in ANZ, EU, HK and US versions respectively. Refer to chapter 7.1 regarding details about the build procedure.

3.3.3 Common

The Product\Common directory contains a set of standard make files needed for building the sample applications. The make files define the compiler options, linker options and defines for the different library types.

3.3.4 Development Controller

The Product\Dev_Ctrl directory contains sample source code for the development controller application used on the ZM12xxRE Module mounted on the Z-Wave Development module. For further information refer to section 7.4 and reference [15].

dev_ctrl.c **dev_ctrl_if.h**	These files contain the source code for the development controller's application state machine. The common API functions such as **ApplicationInitHW**, **ApplicationInitSW**, **ApplicationNodeInformation**, **ApplicationPoll**, **ApplicationRfNotify** and **ApplicationCommandHandler** are defined here.
p_button.c **p_button.h**	Collection of functions and macros used to access the pushbuttons on the development module
eeprom.c **eeprom.h**	These files define the functions for accessing the application data in the external EEPROM and associated addresses.
MK.BAT	Batch file used to build ZW0102/ZW0201/ZW0301 based sample applications in ANZ, EU, HK and US versions respectively. Refer to chapter 7.1 regarding details about the build procedure.

CONFIDENTIAL

3.3.5 Door Bell

The Product\Doorbell directory contains sample source code for the door bell application used on the Z-Wave Interface module. The Development Controller application can be used as push button to control the door bell application. For further information refer to section 7.5.

bell.c **bell.h**	These files contain the source code for the door bell's application state machine. The common API functions such as **ApplicationInitHW**, **ApplicationInitSW**, **ApplicationNodeInformation**, **ApplicationPoll**, **ApplicationRfNotify** and **ApplicationCommandHandler** are defined here.
eeprom_slave.h	This file contains various defines related to the external EEPROM.
MK.BAT	Batch file used to build ZW0201/ZW0301 based sample applications in ANZ, EU, HK and US versions respectively. Refer to chapter 7.1 regarding details about the build procedure.

3.3.6 IO_defines

The Product\IO_defines directory contains hardware definition files needed for building an application e.g. the development controller sample application.

ZW_evaldefs.h	This file contains definitions of the connector pins on the controller board.
ZW_pindefs.h	This file contains definitions of the connector pins on the ZM12xxRE/ZM21xxE/ZM31xxC-E module, and macros for accessing the I/O pins. Refer to paragraph 6.1 regarding a detail description.

CONFIDENTIAL

3.3.7 LED Dimmer

The Product\LED_Dimmer directory contains sample source code for a slave application on a Z-Wave module, which uses the 4 LEDs on the interface module to simulate a light switch with a built in dimmer (see section 7.6.2).

LEDdim.c **LEDdim.h**	These files contain the source code for the LED dimmer's application state machine. The common API functions such as **ApplicationInitHW**, **ApplicationInitSW**, **ApplicationNodeInformation**, **ApplicationPoll,** **ApplicationSlaveUpdate**, **ApplicationRfNotify** and **ApplicationCommandHandler** are defined here.
eeprom.h	This header file defines the addresses where application data are stored in the external EEPROM.
MK.BAT	Batch file used to build ZW0102/ZW0201/ZW0301 based sample applications in ANZ, EU, HK and US versions respectively. Refer to chapter 7.1 regarding details about the build procedure.

3.3.8 MyProduct

The Product\MyProduct directory contains sample source code for a slave application on a Z-Wave module.

MyProduct.c **MyProduct.h**	These files contain the source code for the My Product's application state machine. The common API functions such as **ApplicationInitHW**, **ApplicationInitSW**, **ApplicationNodeInformation**, **ApplicationPoll**, **ApplicationRfNotify** and **ApplicationCommandHandler** are defined here.
MK.BAT	Batch file used to build ZW0102/ZW0201/ZW0301 based sample applications in ANZ, EU, HK and US versions respectively. Refer to chapter 7.1 regarding details about the build procedure.

CONFIDENTIAL

3.3.9 Production Test DUT

The Product\ProdTestDUT directory contains sample source code for a production test DUT application on a Z-Wave module.

prodtestdut.c **prodtestdut.h**	These files contain the source code for the production test DUT's application state machine. The common API functions such as **ApplicationInitHW**, **ApplicationInitSW**, **ApplicationNodeInformation**, **ApplicationPoll**, **ApplicationRfNotify** and **ApplicationCommandHandler** are defined here.
MK.BAT	Batch file used to build ZW0102/ZW0201/ZW0301 based sample applications in ANZ, EU, HK and US versions respectively. Refer to chapter 7.1 regarding details about the build procedure.

3.3.10 Production Test Generator

The Product\ProdTestGen directory contains sample source code for a production test generator application on a Z-Wave module.

Prod_test_gen.c	These files contain the source code for the production test generator's application state machine. The common API functions such as **ApplicationInitHW**, **ApplicationInitSW**, **ApplicationNodeInformation**, **ApplicationPoll**, **ApplicationRfNotify** and **ApplicationCommandHandler** are defined here.
MK.BAT	Batch file used to build ZW0102/ZW0201/ZW0301 based sample applications in ANZ, EU, HK and US versions respectively. Refer to chapter 7.1 regarding details about the build procedure.

CONFIDENTIAL

3.3.11 Serial API

The Product\SerialAPI directory contains sample source code for the Serial API for the ZM12xxRE module. The Serial API consists of this code together with the PC source code in the PC\SerialAPI directory. For further information about the Serial API refer to section 7.5.

serialappl.c **serialappl.h**	This module implements Serial API protocol handling. The module parse the frames, calls the appropriate Z-Wave API library functions and returns results etc. to the PC.
conhandle.c **conhandle.h**	Routines for handling Serial API protocol between PC and Z-Wave module.
UART_buf_io.c **UART_buf_io.h**	Low-level routines for handling buffered transmit/receive of data through the UART.
MK.BAT	Batch file used to build ZW0102/ZW0201/ZW0301 based sample applications in ANZ, EU, HK and US versions respectively. Refer to chapter 7.1 regarding details about the build procedure.

3.3.12 Smoke Detector

The Product\Smoke_Sensor directory contains sample source code for a smoke sensor application on a Z-Wave module.

smokesensor.c **smokesensor.h**	These files contain the source code for the smoke sensor's application state machine. The common API functions such as **ApplicationInitHW**, **ApplicationInitSW**, **ApplicationNodeInformation**, **ApplicationPoll**, **ApplicationRfNotify** and **ApplicationCommandHandler** are defined here.
eeprom_slave.h	This file contains various defines related to the external EEPROM.
MK.BAT	Batch file used to build ZW0102/ZW0201/ZW0301 based sample applications in ANZ, EU, HK and US versions respectively. Refer to chapter 7.1 regarding details about the build procedure.

CONFIDENTIAL

3.3.13 Utilities

The Product\util_func directory contains some helpful functions that are used by several of the sample applications.

App_RFSetup.a51	This file is an assembler file that should be used to define which RF setup is to be used (ANZ/EU/HK/US) in the transmissions. Also if special values for capacity match array RX/TX or Normal/Low power transmission levels are needed then these can be defined here.
association.c **association.h**	This file contains sample code that shows how association between nodes could be implemented on a Z-Wave module. This sample code holds all associations in RAM and the number of nodes/groupings possible using this implementation is limited. Applications using this collection of functions must implement three functions (ApplicationStoreAll, ApplicationInitAll, ApplicationClearAll). These should handle the storage in nonvolatile memory if this is desired.
battery.c **battery.h**	This file contains sample code that shows how battery operated devices may implement power down, wake up notification and network update requests. Applications using this collection must call the following functions at their appropriate location: *UpdateWakeupCount* – call from ApplicationInitSW to update the wakeup counter which determines the wakeup interval on application level (200-series) – Only called when Wakeupreason is WUT-Kicked. *InitRTCActionTimer* – call from ApplicationInitSW, to activate the RTC timer. (100-Series) *HandleWakeupFrame* – call from ApplicationCommandHandler to handle incoming COMMAND_CLASS_WAKE_UP is received. Handles WAKE_UP_INTERVAL_GET/SET/NO_MORE_INFORMATION. *SetDefaultBatteryConfiguration* – is called from ApplicationInitHW when node is reset, and from SetDefaultConfiguration. Sets the default values for powerdown timeout, sleep time and networkupdate. *LoadBatteryConfiguration* – call from LoadConfiguration. Loads the battery related information from EEPROM and make them available for the running application. *SaveBatteryConfiguration* – call from SaveConfiguration. Saves the battery related information to EEPROM. *StartPowerDownTimer* – call from ApplicationInitSW and set as callback function ZW_SEND_DATA methods after which the node should enter sleep mode. Please refer to the BatterySensor sample application for an example on how this can be implemented.
one_button.c **one_button.h**	Enables easy use of a button. The functions detect whether a button has been pressed shortly or is being held. To initialise the button detection, run OneButtonInit() from ApplicationInitSW. And call OneButtonLastAction when button information is needed (e.g. in ApplicationPoll()).
self_heal.c **self_heal.h**	Support functions to implement Lost / Self Heal functionality. This file is mandatory if the battery helper functions are used and ZW_SELF_HEAL is defined. See the battery.c and bin_sensor.c source files for help on using the functions.

CONFIDENTIAL

slave_learn.c **slave_learn.h**	The file contains sample code for how to handle learn mode on slave nodes. These two files are used by all slave based sample code in the development kit. The sample application should just call StartLearnModeNow() to enter learnmode and transmit nodeinformation. The sample application should then wait for the BOOL learnState to go FALSE before doing transmissions.

CONFIDENTIAL

3.4 Tools

The Tools directory contains various tools needed for building and debugging the sample applications. All tools in this directory can freely be used for building Z-Wave applications.

3.4.1 Eeprom Loader

This directory contains tools used for downloading hex files to the external EEPROM on the ZM1206 module through the serial port. The external EEPROM on all the other Z-Wave modules can be updated by the Z-Wave Programmer. Additional files are also added in case the external EEPROM is updated using the Equinox Epsilon5 programmer.

eeploader_ZW0102.hex	This program must be downloaded to the ZW0102 module before downloading data to the external EEPROM. The program receives the external EEPROM data via the serial port and copies it to the external EEPROM. Only necessary to use in conjunction with the Epsilon5 programmer or when downloading hex files to the external EEPROM on the ZM1206 module using the Z-Wave Programmer.
eeploader_ZW0201.hex	This program must be downloaded to the ZW0201 based module before downloading data to the external EEPROM. The program receives the external EEPROM data via the serial port and copies it to the external EEPROM. Only necessary to use in conjunction with the Epsilon5 programmer
eeploader_ZW0301.hex	This program must be downloaded to the ZW0301 based module before downloading data to the external EEPROM. The program receives the external EEPROM data via the serial port and copies it to the external EEPROM. Only necessary to use in conjunction with the Epsilon5 programmer
eeploader.exe	Used to download the **extern_eep.hex** file to external EEPROM via the PC’s serial port. Refer to paragraph 8.7 for further details. Only necessary to use in conjunction with the Epsilon5 programmer or when downloading hex files to the external EEPROM on the ZM1206 module using the Z-Wave Programmer.
doload.bat	Used to download the **extern_eep.hex** file to external EEPROM via the PC’s serial port. Only necessary to use in conjunction with the Epsilon5 programmer

The procedure to doanload hex files to the external EEPROM on the Z-Wave module using the Equinox Epsilon5 programmer with the Developer’s Kit is as follows:

1. Create the relevant PPC file from eeploader_ZW0x0x.hex file by the EQTools application.
2. Use the Epsilon5 programmer to download PPC file to the Z-Wave module.
3. Connect a RS-232 serial cable directly from the PC the Z-Wave interface module. Notice that download is not done via the Epsilon5 programmer.
4. In case a ZW0201 based Development Controller Unit is used then remember to remove the short circuit between RXD (J9) and LED1 (J1) and reinstall jumper J9 to enable the UART.

CONFIDENTIAL

5. Open a DOS prompt on your PC and go to the directory C:\DevKit_5_00\Product\Bin\... containing the appropriate image file Extern_eep.hex. The doload.bat file must also be available in this directory and is used to download the image file via the COM port. Notice: It is necessary to reset the ZW0201/ZW0301 before downloading the image file to the external EEPROM.
6. Run **doload.bat COMx** (x = the number of the COM port the cable is attached to).
7. The PC application displays download progress and finally status of the download.

3.4.2 Equinox

This directory contains the documentation and software needed by the Equinox Epsilon5 programmer when programming the ZW0102/ZW0201/ZW0301 based modules. Remember that the Equinox Epsilon5 programmer is replaced by the new Z-Wave programmer in the Developer's Kit. This solution is included because many customers have already establish production lines using the Equinox Epsilon5 programmer.

ConfigIt.exe	The ConfigIt v6.04 program is used to update the Equinox Epsilon5 programmer with new firmware v2.48. The generated ZW0102/ZW0201/ZW0301 Equinox PPC files uses the new format. Therefore is it necessary to upgrade the Equinox Epsilon5 programmer delivered in Developer's Kit v3.40 or older before the PPC files using the new format can be downloaded.
EQTools_v210_b576i.exe	The EQTools program is used to download the PPC files to the Equinox Epsilon5 programmer. Via the buttons on the Equinox Epsilon5 programmer is the hex file transferred to the ZW0102/ZW0201/ZW0301 based module. PPC files provided in Developer's Kit v3.40 or older cannot be used by this version of EQTools because the PPC format has changed.
ZW0102-BASIC.PPM	Project default settings for the ZW0102 when embedding the hex file in the PPC file using EQTools.
ZW0201-BASIC.PPM	Project default settings for the ZW0201 when embedding the hex file in the PPC file using EQTools.
ZW0301-BASIC.PPM	Project default settings for the ZW0301 when embedding the hex file in the PPC file using EQTools.
INS10309-4 - Using the Epsilon5 for programming Z-Wave Single Chips.pdf	Users guide located in the directory: X:\data\SW_DOC\ on the Developer's Kit CD.

CONFIDENTIAL

3.4.3 ERTT

This directory contains the PC software and the embedded code for the Enhanced Reliability Test Tool. (ERTT) For further details refer to [11].

The ERTT directory contains the following files:

PC\setup.exe	PC application.
Z-Wave_Firmware\extern_eep.hex	This file contains the external EEPROM data on the ZM12xxRE/ZM21xxE/ZM31xxC-E module. The external EEPROM must only be initialized once.
Z-Wave_Firmware\serialapi_ctrl_single_ZW010x_EU.hex **Z-Wave_Firmware\serialapi_ctrl_single_ZW010x_US.hex**	The compiled and linked ERTT application for the EU and US versions of the ZW0102 based module.
Z-Wave_Firmware\serialapi_ctrl_single_ZW020x_ANZ.hex **Z-Wave_Firmware\serialapi_ctrl_single_ZW020x_EU.hex** **Z-Wave_Firmware\serialapi_ctrl_single_ZW020x_HK.hex** **Z-Wave_Firmware\serialapi_ctrl_single_ZW020x_US.hex**	The compiled and linked ERTT application for the ANZ, EU, HK and US versions of the ZW0201 based module.
Z-Wave_Firmware\serialapi_ctrl_single_ZW030x_ANZ.hex **Z-Wave_Firmware\serialapi_ctrl_single_ZW030x_EU.hex** **Z-Wave_Firmware\serialapi_ctrl_single_ZW030x_HK.hex** **Z-Wave_Firmware\serialapi_ctrl_single_ZW030x_US.hex**	The compiled and linked ERTT application for the ANZ, EU, HK and US versions of the ZW0301 based module.

3.4.4 IncDep

This directory contains a python script that is used for making dependency files when building the sample applications.

3.4.5 Intel UPnP

This directory contains Intel’s tools for UPnP technology helping software developers during development, testing, and deployment of UPnP-compliant devices. These tools are used in conjunction with the Z-Wave to UPnP bridge sample applications [8].

3.4.6 Make

This directory contains a DOS/Windows version of the GNU make utility. The make utility is used for building the sample applications.

CONFIDENTIAL

3.4.7 Mergehex

This directory contains a tool used for merging two files in Intel hex format. The tool is used for building external EEPROM files in the sample code.

3.4.8 Programmer

This Programmer directory contains the PC software and ATMega128 firmware for the Programmer. For further details refer to [14].

The Programmer directory contains the following files:

PC\setup.exe	PC application.
PC\CP210x_VCP_Win2K_XP.exe	USB driver.
ZDP02A_Firmware\ATMega128_Firmware.hex	The compiled and linked Z-Wave Programmer application for the ATMega128 situated on the ZDP02A Development Platform.

CONFIDENTIAL

3.4.9 PVT and RF Regulatory

This directory contains software used in connection with PVT and RF regulatory measurements on the hardware. For a guideline how to carry out the measurements refer to [9] and [19].

The PVT_and_RF_regulatory directory contains the following files:

ZW0102_Fmax.hex	Used for measuring the lower frequency of the VCO (only on ZW0102).
ZW0102_Fmin.hex	Used for measuring the higher frequency of the VCO (only on ZW0102).
ZW0102_rx_eu.hex	Puts the ZW0102 in receive mode. European products.
ZW0201_rx_eu.hex	Puts the ZW0201 in receive mode. European products.
ZW0301_rx_eu.hex	Puts the ZW0301 in receive mode. European products.
ZW0102_rx_us.hex	Puts the ZW0102 in receive mode. US products.
ZW0201_rx_us.hex	Puts the ZW0201 in receive mode. US products.
ZW0301_rx_us.hex	Puts the ZW0301 in receive mode. US products.
ZW0102_TXcar_60_us.hex	ZW0102 constantly transmits a carrier (908.42MHz).
ZW0201_TXcar_us.hex	ZW0201 constantly transmits a carrier (908.42MHz).
ZW0301_TXcar_us.hex	ZW0301 constantly transmits a carrier (908.42MHz).
ZW0102_TXcar_f0_eu.hex	ZW0102 constantly transmits a carrier (868.42MHz).
ZW0201_TXcar_eu.hex	ZW0201 constantly transmits a carrier (868.42MHz).
ZW0301_TXcar_eu.hex	ZW0301 constantly transmits a carrier (868.42MHz).
ZW0102_TXgen.hex	SW loaded to a ZM1220 generating a digital Z-Wave Frame bit stream. A FSK/FM signal generator is used to modulate the signal.
ZW0102_TXmod_01_eu.hex	ZW0102 constantly transmits a modulated signal - 868.42MHz +/-20kHz (output power setting: “01”).
ZW0102_TXmod_01_us.hex	ZW0102 constantly transmits a modulated signal - 908.42MHz +/-20kHz (output power setting: “01”).
ZW0102_TXmod_f0_eu.hex	ZW0102 constantly transmits a modulated signal - 868.42MHz +/-20kHz (output power setting: “F0”).
ZW0201_TXmod_eu.hex	ZW0201 constantly transmits a modulated signal - 868.42MHz +/-25kHz.
ZW0301_TXmod_eu.hex	ZW0301 constantly transmits a modulated signal - 868.42MHz +/-25kHz.
ZW0102_TXmod_60_us.hex	ZW0102 constantly transmits a modulated signal - 908.42MHz +/-20kHz.
ZW0201_TXmod_us.hex	ZW0201 constantly transmits a modulated signal - 908.42MHz +/-25kHz.
ZW0301_TXmod_us.hex	ZW0301 constantly transmits a modulated signal - 908.42MHz +/-25kHz.

Notice: Hex files for ZW0301 and ZW0201 using ANZ and HK frequencies are currently not available. Contact Technical Services support@zen-sys.com to obtain these files.

CONFIDENTIAL

3.4.10 Python

This directory contains a python scripting language interpreter. Python is used for various purposes in the sample code build process.

3.4.11 Zniffer

The Zniffer directory contains the Z-Wave protocol Zniffer program; the program can be used for viewing the application commands send out on the RF media from a Z-Wave module. The tool consists of two parts, an embedded part that should be downloaded to a Z-Wave module and a PC application that should run on a PC attached to the Z-Wave module via the serial interface. For further details refer to [12].

The Zniffer directory contains the following files:

PC\setup.exe	PC application.
Z-Wave_Firmware\sniffer_ZW010x_EU.hex **Z-Wave_Firmware\sniffer_ZW010x_US.hex**	The compiled and linked Zniffer application for the EU and US versions of the ZW0102 based module.
Z-Wave_Firmware\sniffer_ZW020x_EU.hex **Z-Wave_Firmware\sniffer_ZW020x_US.hex** **Z-Wave_Firmware\sniffer_ZW020x_ANZ.hex** **Z-Wave_Firmware\sniffer_ZW020x_HK.hex**	The compiled and linked Zniffer application for the ANZ, EU, HK and US versions of the ZW0201 based module.
Z-Wave_Firmware\sniffer_ZW030x_EU.hex **Z-Wave_Firmware\sniffer_ZW030x_US.hex** **Z-Wave_Firmware\sniffer_ZW030x_ANZ.hex** **Z-Wave_Firmware\sniffer_ZW030x_HK.hex**	The compiled and linked Zniffer application for the ANZ, EU, HK and US versions of the ZW0301 based module.

3.5 PC

The PC directory contains three PC sample applications demonstrating the use of the Z-Wave DLL and Serial API.

3.5.1 Bin

The PC\Bin directory contains the program or installation executables of the PC sample applications.

InstallerTool\InstallerTool.exe	The InstallerTool application.
ZWavePCController\Setup.Exe	The PC Controller application.
ZWaveUPnPBridge\Setup.Exe	The Z-WaveBridge application

CONFIDENTIAL

3.5.2 InstallerTool

The PC\InstallerTool directory contains sample source code for a PC based Installer Tool using the Serial API. Further readings on how to use the InstallerTool see [7].

InstallerTool.dsp	Microsoft Development Environment 2003 project file containing information at the project level and is used to build the project. Application is implemented in C++.

3.5.3 Serial API

The PC\SerialAPI directory contains sample source code used by the PC application to communicate with the ZW0102/ZW0201/ZW0301 based module via the serial API interface. Currently only used by the PC based Installer Tool. Notice that future PC applications should be based on the Z-Wave DLL as interface to the Z-Wave network. Refer to paragraph 7.5 regarding details about the embedded Serial API interface.

3.5.4 Source

The PC\Source directory contains the PC applications for the Microsoft Visual Studio 2005 environment. All applications are implemented in C#.

3.5.4.1 Libraries

The PC\Source\Libraries directory contains various libraries used by the PC applications.

3.5.4.1.1 Transport Layer

The PC\Source\Libraries\ITransportLayer directory contains source code of the interface between the transport layer, e.g. ZWRS232 and the DLL.

ITransportLayer.sln	Microsoft Visual Studio 2005 solution file containing information at the project level and is used to build the project. Application is implemented in C#.

3.5.4.1.2 Z-Wave DLL

The PC\Source\Libraries\ZWave directory contains source code of the dynamic link library used by the PC application to communicate with the ZW0102/ZW0201/ZW0301 based module via the serial API interface. Refer to [13] for further details.

ZWave.sln	Microsoft Visual Studio 2005 solution file containing information at the project level and is used to build the project. Application is implemented in C#.

CONFIDENTIAL

3.5.4.1.3 Z-Wave Command Class

The PC\Source\Libraries\ZWaveCmdClass directory contains source code for the XML parser, which enables parsing of Z-Wave frames by the Zniffer and generating frames by the PC Controller.

ZWaveCmdClass.sln	Microsoft Visual Studio 2005 solution file containing information at the project level and is used to build the project. Application is implemented in C#.

3.5.4.1.4 Z-Wave RS232

The PC\Source\Libraries\ZwaveRS232 directory contains source code for the serial port transport layer which is interface by the ITransportLayer.

ZwaveRS232.sln	Microsoft Visual Studio 2005 solution file containing information at the project level and is used to build the project. Application is implemented in C#.

3.5.4.2 Sample Application

The PC\Source\SampleApplication contains the various the PC applications

3.5.4.2.1 Zensys Classes

The PC\Source\SampleApplication\ZensysClasses directory contains various classes used by the PC applications.

ZCmdClass.cs	This class provides a number of methods for device and command class types and definitions. Application is implemented in C#.

3.5.4.2.2 Z-Wave PC Controller

The PC\Source\SampleApplication\ZWavePCController directory contains sample source code for a PC based controller using the Z-Wave DLL etc. Further reading on how to use the PC based Controller see [6].

ZWavePCController.sln	Microsoft Visual Studio 2005 solution file containing information at the project level and is used to build the project. Application is implemented in C#.

3.5.4.2.3 Z-Wave UPnP Bridge

The PC\Source\SampleApplication\ZWaveUPnPBridge directory contains sample source code for a PC based Z-Wave Bridge using the Z-Wave DLL etc. Further readings on how to use the Z-Wave UPnP Bridge see [8].

ZWaveUPnPBridge.sln	Microsoft Visual Studio 2005 solution file containing information at the project level and is used to build the project. Application is implemented in C#.

CONFIDENTIAL

3.6 Graphics

The Graphics directory contains an icon file showing the Z-Wave logo used by PC applications.

Z_wave.ico	Z-Wave icon file.

CONFIDENTIAL

4 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.

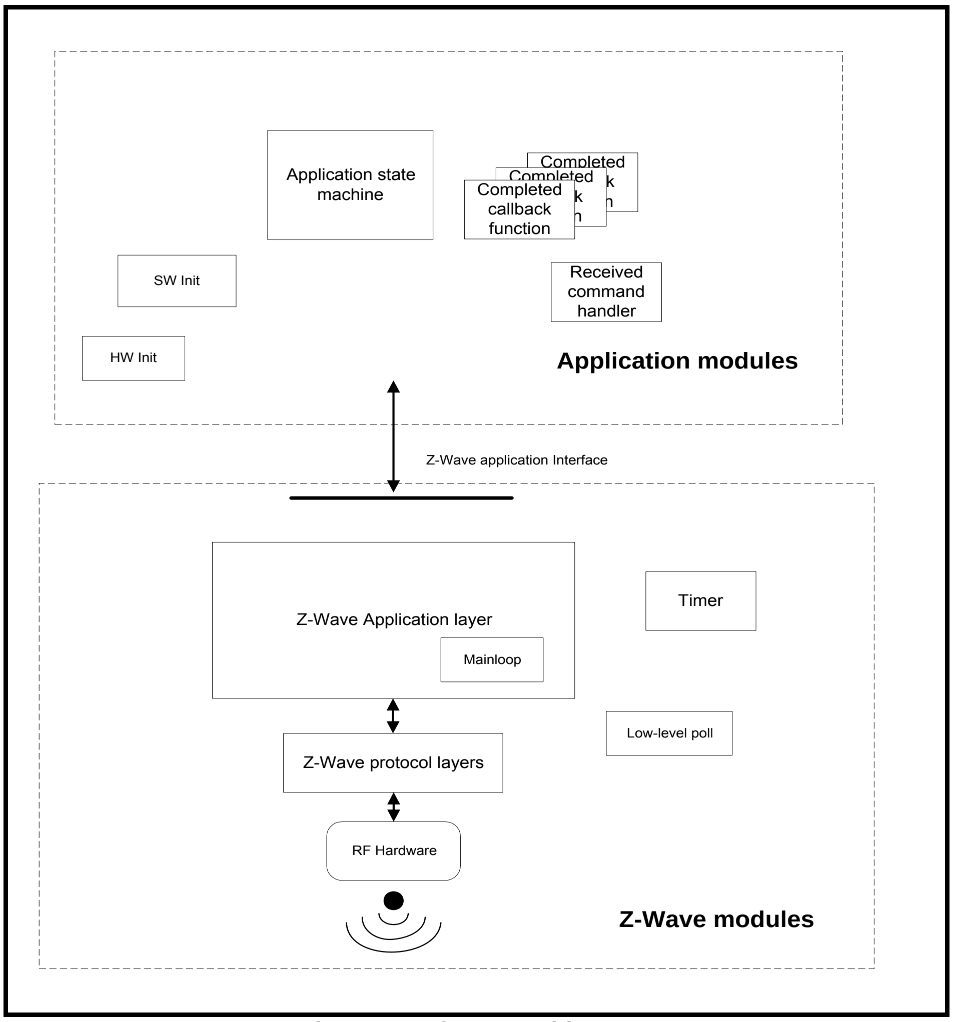


Figure 1 Software architecture

CONFIDENTIAL

4.1 Z-Wave System Startup Code

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

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

4.2 Z-Wave Main Loop

The Z-Wave main loop will call the list of Z-Wave protocol functions, including the **ApplicationPoll** function, the **ApplicationCommandHandler** function (if a frame was received) and the RTC timer function in round robin order. The functions must therefore be designed to return to the caller as fast as possible to allow the CPU to do other tasks. Busy loops are not allowed. This will make it possible to receive Z-Wave data, transfer data via the UART and check user-activated buttons "simultaneously".

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

4.3 Z-Wave Protocol Layers

When the application 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 protocol layer also provides support for operation in Zensor Nets, i.e. environments of entirely battery-operated nodes. Seen from the application layer, the only difference from a normal Z-Wave Network is a significantly higher latency as it may take some time to get in contact with a battery-operated node.
The Zensor Net Routing Slave is an extension to the normal Routing Slave library.

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.

4.4 Z-Wave Application Layer

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

CONFIDENTIAL

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

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

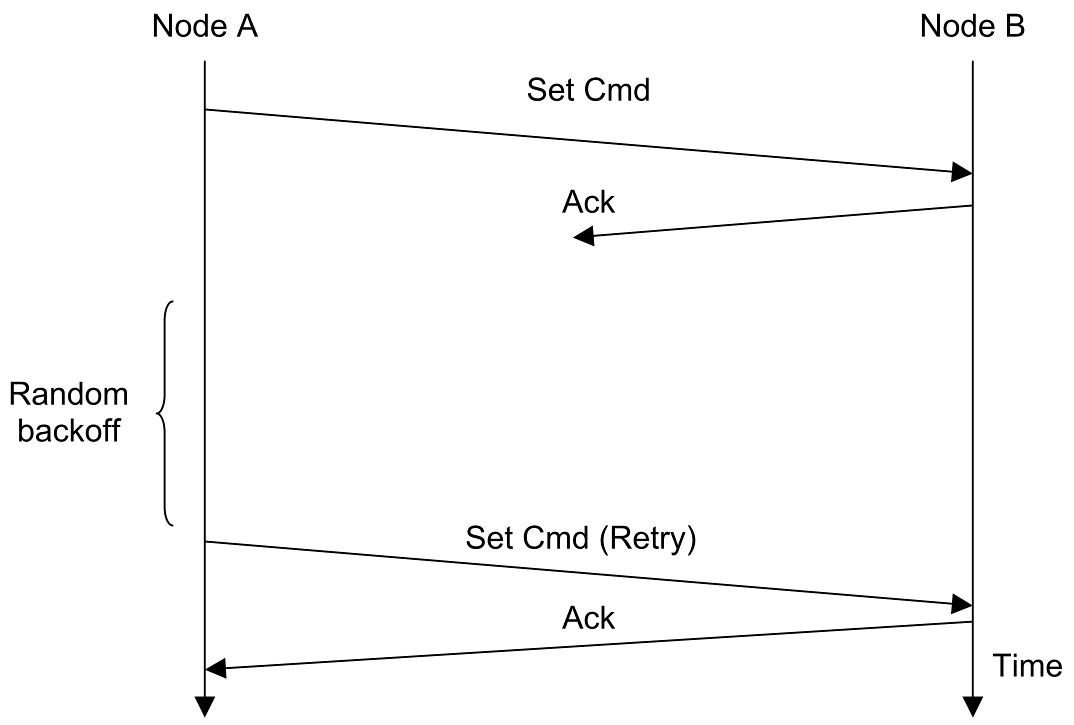


Figure 2 Multiple copies of the same Set frame

CONFIDENTIAL

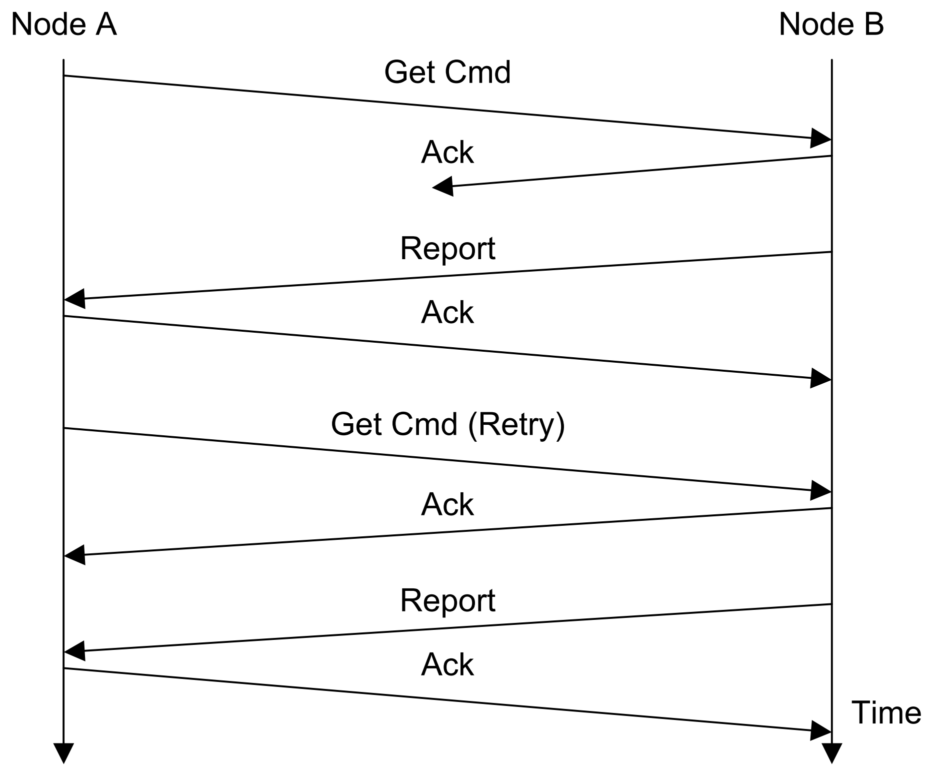


Figure 3 Multiple copies of the same Get/Report frame

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.

CONFIDENTIAL

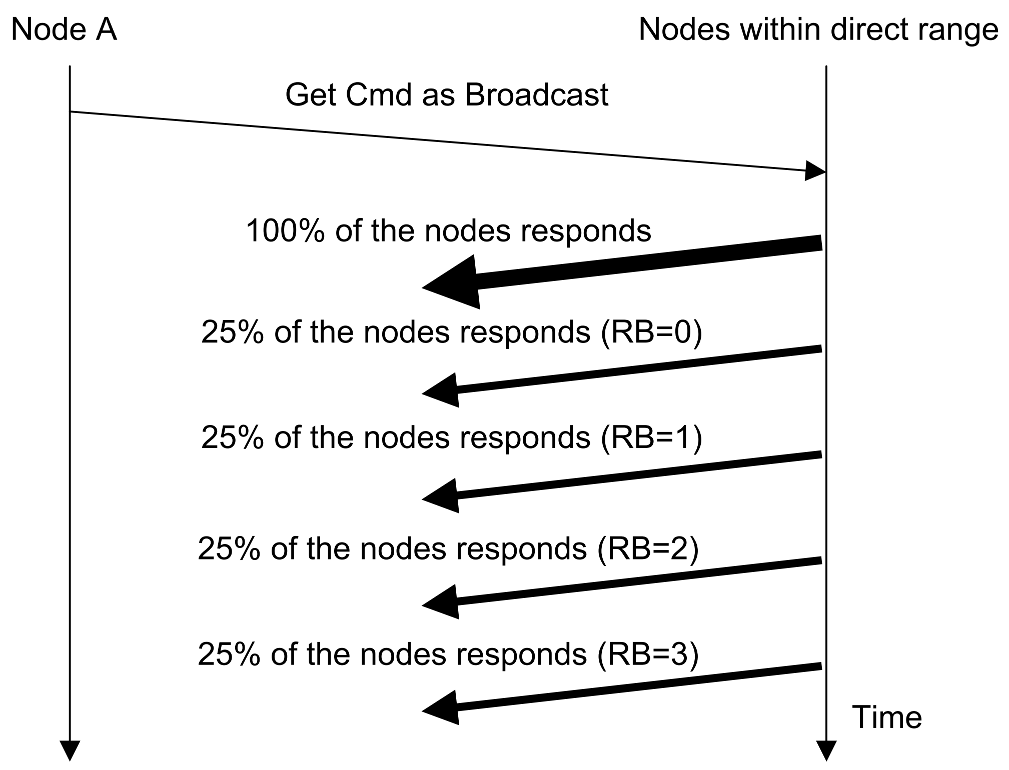

Node A
Nodes within direct range
Get Cmd as Broadcast
100% of the nodes responds
25% of the nodes responds (RB=0)
25% of the nodes responds (RB=1)
25% of the nodes responds (RB=2)
25% of the nodes responds (RB=3)
Time

Figure 4 Simultaneous communication to a number of nodes

Simultaneous communication to nodes in a Z-Wave network which requires a response from the nodes in question must therefore be avoided in the application.

4.5 Z-Wave Software Timers

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

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

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

CONFIDENTIAL

4.6 Z-Wave Hardware Timers

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

Table 1, ZW0102/ZW0201/ZW0301 hardware timer allocation

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

The TIMER0 and TIMER1 are standard 8051 timers/counters.

4.7 Z-Wave Hardware Interrupts

Application interrupt functions must use 8051 register set 0. The Z-Wave protocol uses 8051 register set 1 for protocol ISR's, see table below regarding application ISR availability:

Table 2, ZW0102/ZW0201/ZW0301 Application ISR avialability

ZW0102	ZW0201	ZW0301
INUM_INT0 INUM_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

Refer to ZW010x,.h ZW020x, and ZW030x.h header files with respect to ISR definitions. For examples refer to the sample applications.

CONFIDENTIAL

4.8 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, Slave 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.

4.8.1 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 physical application memory offset is defined in the header file “ZW_eep_addr.h”.

CONFIDENTIAL

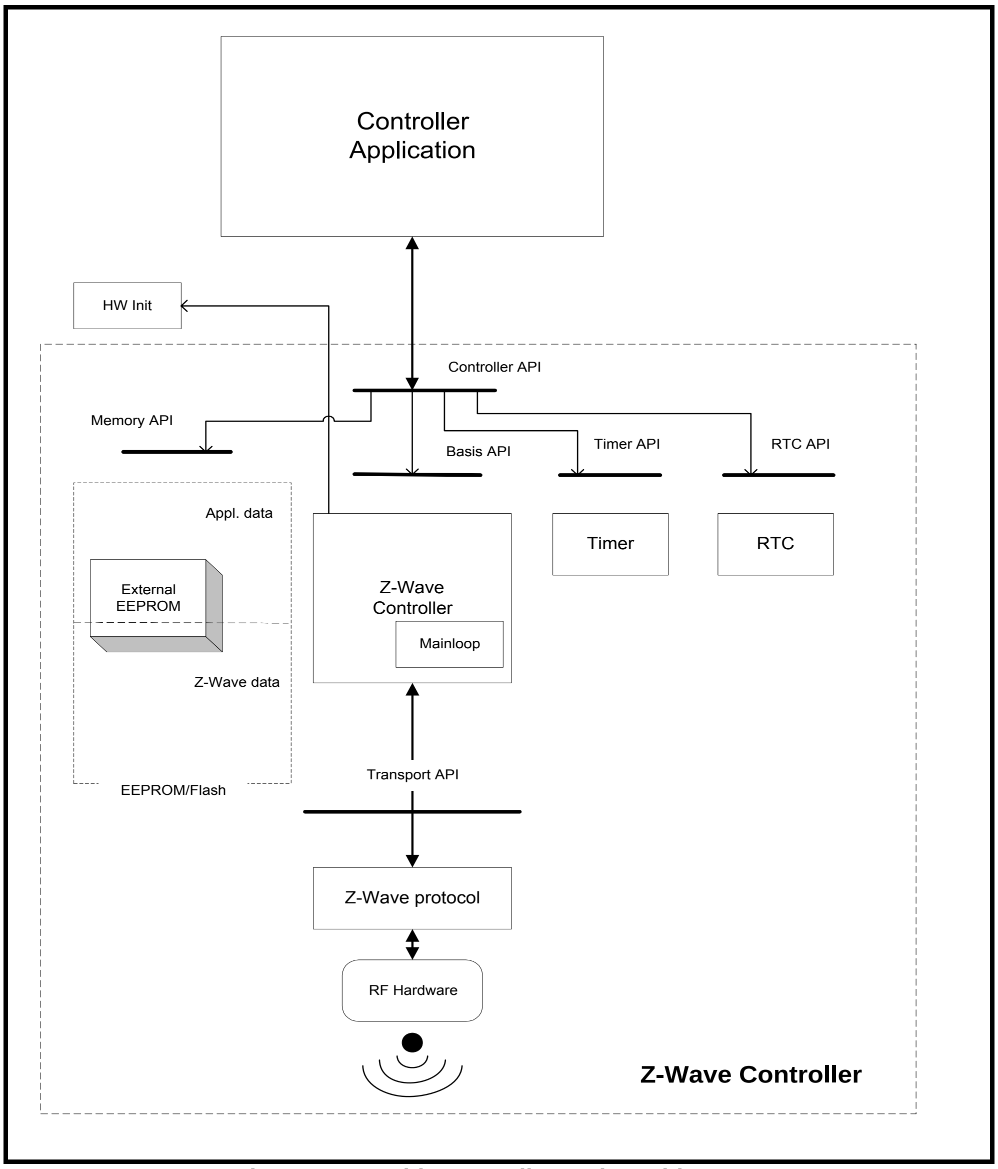


Figure 4 Portable controller node architecture

The Portable Controller node has a unique home ID number assigned, which is stored in the Z-Wave basis area of the external EEPROM. Care must be taken, when reprogramming the external EEPROM, that different controller nodes do not get the same home ID number. Refer to paragraph 8.7 regarding a description of external EEPROM programming.

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

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

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.

CONFIDENTIAL

The application must be linked with ZW_CONTROLLER_PORTABLE_ZW∗S.LIB
(∗ = 010X for ZW0102 modules, etc).

4.8.2 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 4.8.1 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 slave static controller not functioning as SUC can also request network Topology updates.

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
(∗ = 010X for ZW0102 modules, etc).

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

The application must be linked with ZW_CONTROLLER_INSTALLER_ZW∗S.LIB
(∗ = 010X for ZW0102 modules, etc).

4.8.4 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. A Virtual Slave node is a slave node, 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.

CONFIDENTIAL

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
(∗ = 010X for ZW0102 modules, etc).

4.8.5 Z-Wave Slave Node

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

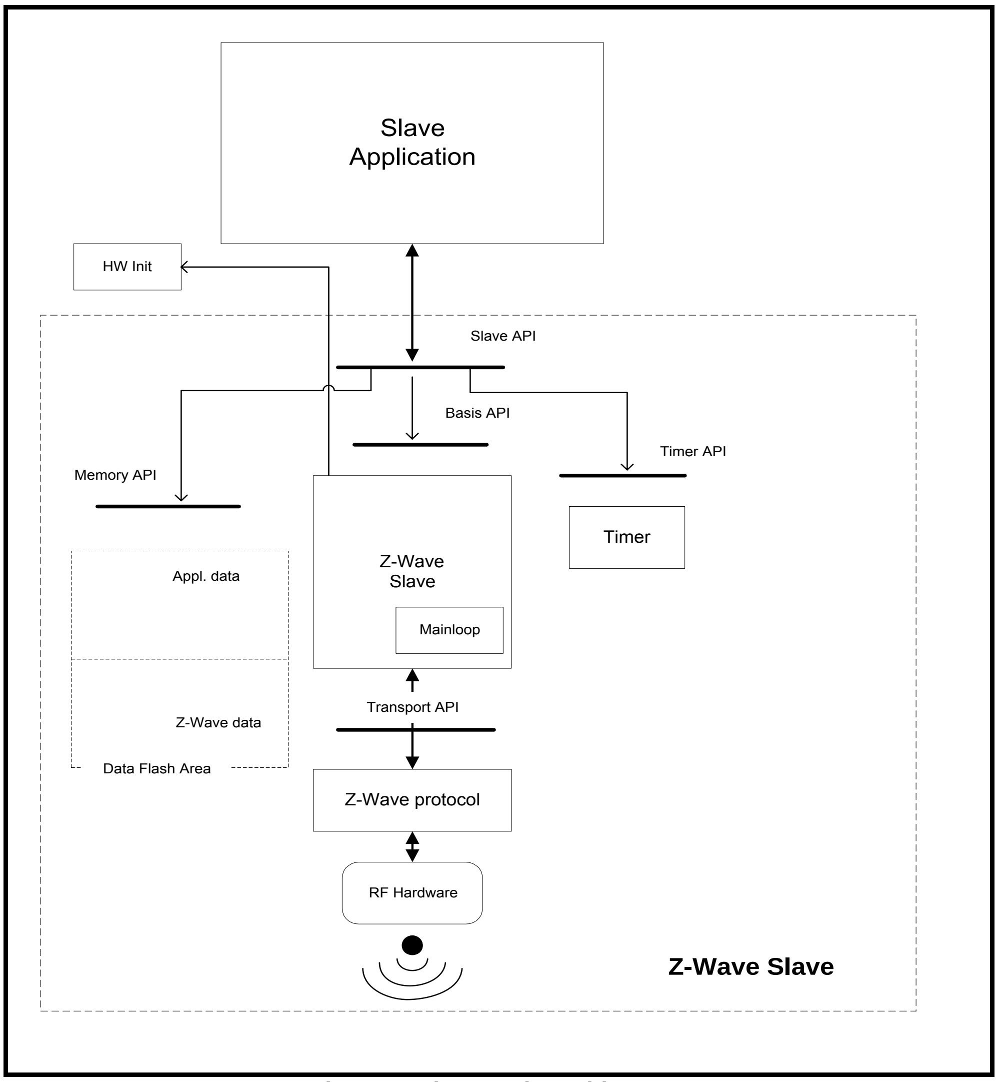


Figure 5 Slave node architecture

Slave nodes have an area of approximately 4 Kbytes in the flash reserved for storing data, but only 125 bytes can be used directly. The Z-Wave basis software reserve the first part of this area, and the last part of the area are reserved for the application data. The physical application memory offset is defined in the header file “ZW_eep_addr.h”.

CONFIDENTIAL

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

The slave can send unsolicited broadcasts and non-routed singlecasts. 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.

When developing application software the header file "ZW_slave_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_SLAVE.

The application must be linked with ZW_SLAVE_ZW∗S.LIB
(∗ = 010X for ZW0102 modules, etc).

4.8.6 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. Refer to Figure 5.

The routing slave is an extension to the basic slave node. From a functionality point of view, the routing slave is somewhere between a controller and a slave. While a slave can only react to incoming communication, a routing slave is also capable of initiating communication. Examples of a routing slave could be a wall control or temperature sensor. If a user activates the wall control, the routing slave sends an "on" command to a lamp (slave).
The routing slave does not have a complete routing table. Frames are sent to destinations configured during association. The association is performed via a controller. If routing is needed for reaching the destinations, it is also up to the controller to calculate the routes.

Routing slave nodes have an area of 4352 bytes in the flash reserved for storing data, but only 125 bytes can be used directly. The Z-Wave basis software reserve the first part of this area, and the last part of the area are reserved for the application data. The physical application memory offset is defined in the header file "ZW_eep_addr.h".

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

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.
A special case of a battery-operated routing slave is the Frequently Listening Routing Slave (FLiRS). This is a normal Routing Slave configured to listen for traffic in every wake-up interval. One usage for a FLiRS could be as the chime node in a wireless doorbell system.
In general, battery-operated nodes are not used as repeaters for routing.

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.

CONFIDENTIAL

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
(∗ = 010X for ZW0102 modules, etc).

4.8.7 Z-Wave Enhanced Slave Node

The Z-Wave enhanced slave has the same basic functionality as a Z-Wave routing slave node, but because of more features on the hardware more software components are available.

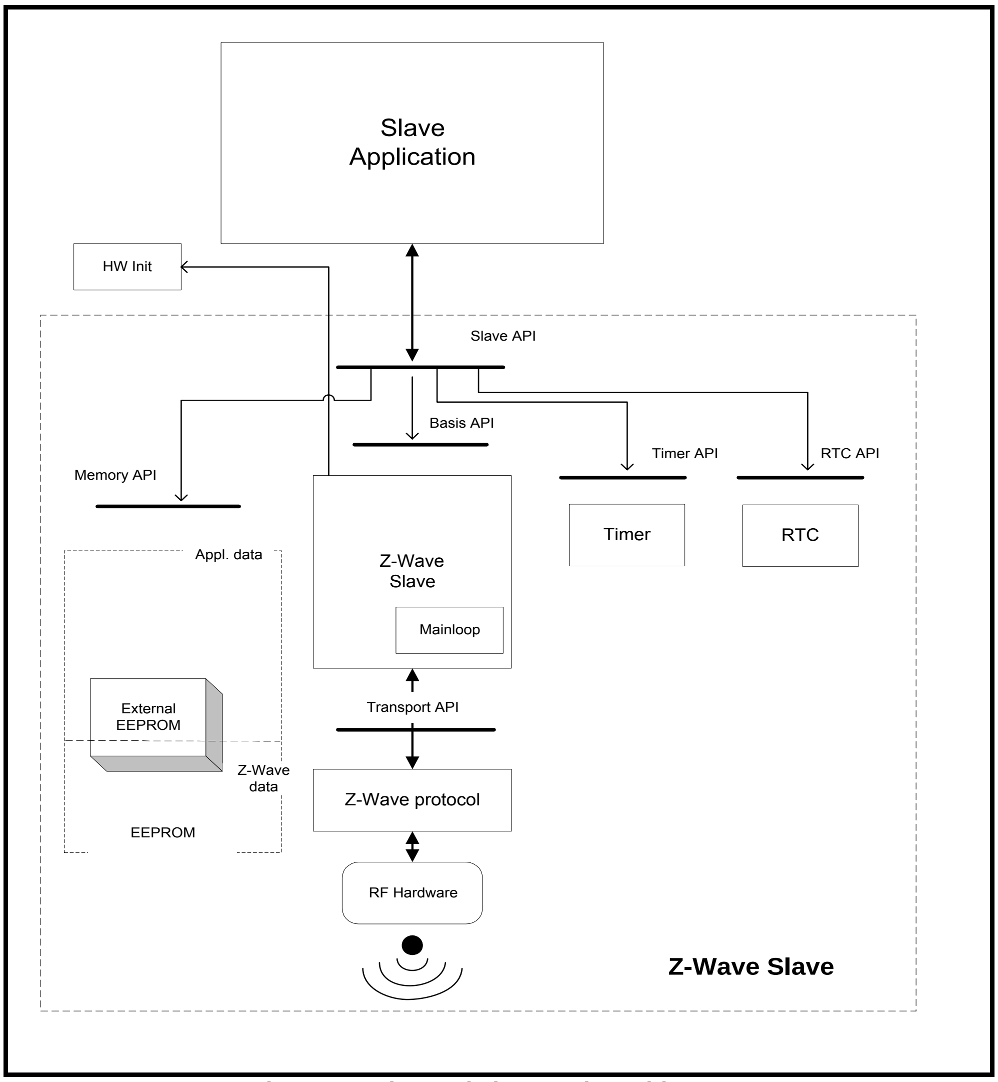


Figure 6 Enhanced slave node architecture

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

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.

CONFIDENTIAL

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
(∗ = 010X for ZW0102 modules, etc).

4.8.8 Z-Wave Zensor Net Routing Slave Node

The Zensor Net Routing Slave node is basically a routing slave node configured as Frequently Listening Routing Slave. The extra features are

- Zensor Net binding
- Zensor Net flooding

Binding is an alternative to classic Z-Wave inclusion. Where a controller is needed to perform inclusion, any Zensor Net node can bind all other Zensor Net nodes. Binding has a strong resemblance to classic Z-Wave inclusion, but it does not replace inclusion. Zensor Nets use a reserved range of Z-Wave Home ID's. A Zensor Net Routing Slave node can be part of a Z-Wave Network and a Zensor Net at the same time. As a consequence, the Zensor Net Routing Slave node must be able to hold the following informations internally:

• Z-Wave HomeID	(32 bits)
• Z-Wave nodeID	(8 bits)
• Zensor Net ID	(16 bits)
• Zensor Net nodeID	(8 bits)

Flooding is a special property only intended for emergency applications such as networked smoke detectors. Zensor Net Routing Slave nodes are capable of forwarding broadcast frames without generating loops. At the same time, special care is taken to ensure that collisions are kept at an acceptable level.

Zensor Net Routing slave nodes have an area of 4352 bytes in the flash reserved for storing data, but only 125 bytes can be used directly. The Z-Wave basis software reserve the first part of this area, and the last part of the area are reserved for the application data. The physical application memory offset is defined in the header file "ZW_eep_addr.h".

The home ID and node ID of a new node is zero. When registering a Zensor Net Routing Slave node to a Z-Wave network the Zensor Net Routing 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 Zensor Net 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.

CONFIDENTIAL

Normally, a Zensor Net Routing Slave is battery operated. To improve battery lifetime, the application will bring the node into sleep mode most of the time. Using the wake-up timer (WUT), the application may wake up once per second or even less often. At every wake-up, the Zensor Net Routing Slave node may do some measurements, listen for traffic and go back to sleep.
An example of Zensor Net Routing Slave nodes is in networked smoke detectors. The binding allows an installer to associate a number of devices without the help of a controller. The flooding allows the installer to place nodes wherever he likes without having to perform any new network discovery. At the same time, the flooding makes the detector network insensitive to failing nodes; be that failing hardware or drained battery.

When developing application software the header file “ZW_slave_sensor_api.h” also include the other Z-Wave API header files e.g. ZW_slave_routing_api_api.h.

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

The application must be linked with ZW_SLAVE_SENSOR_ZW∗S.LIB
(∗ = 010X for ZW0102 modules, etc).

CONFIDENTIAL

4.8.9 Adding and Removing Nodes to/from the network

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

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

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

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

Adding a node normally.

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

Adding a new controller and make it the primary controller

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

Create a new primary controller

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

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

CONFIDENTIAL

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

Table 3. Controller functionality

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

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

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

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

SUC ID Server

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

CONFIDENTIAL

4.8.10 The Automatic Network Update

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

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

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

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

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

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

CONFIDENTIAL

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.

CONFIDENTIAL

5.1 Z-Wave Libraries

5.1.1 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

	Slave	Routing Slave	Sensor Slave	Enhanced Slave	Portable Controller	Static Controller	Installer Controller	Bridge Controller
Basic Functionality								
Singlecast	X	X	X	X	X	X	X	X
Multicast	-	X	X	X	X	X	X	X[1]
Broadcast	X	X	X	X	X	X	X	X
UART support	X	X	X	X	X	X	X	X
SPI support	-	-	-	-	-	-	-	-
ADC support	X	X	X	X	X	X	X	X
TRIAC control	X	X	X	X	X	X	X	X
PWM/HW timer support	X	X	X	X	X	X	X	X
Power management	X	X	X	X	X	-	X	-
Real time clock (only ZW0102)	-	-	-	X	X	X	X	-
SW timer support	X	X	X	X	X	X	X	X
Controller replication	-	-	-	-	X	X	X	X
Promiscuous mode	-	-	-	-	X	-	X	-
Routing table information					X	X	X	X
Memory Location								
Non-volatile RAM in flash	X	X	X	-	-	-	-	-
Non-volatile RAM in EEPROM	-	-	-	X	X	X	X	X
Network Management								
Network router (repeater)	X	X	X	X	-	X	-	-
Assign routes to routing slave	-	-	-	-	X	X	X	X
Routing slave functionality	-	X	X	X	-	-	-	-
Access to routing table	-	-	-	-	X	-	X	-
Maintain virtual slave nodes	-	-	-	-	-	-	-	X[2]
Zensor Net Support								
Zensor Binding	-	-	X	-	-	-	-	-
Zensor Flooding	-	-	X	-	-	-	-	-

[1] Do not apply for virtual slaves

[2] Only when secondary controller

CONFIDENTIAL

5.1.1.1 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

	Slave	Routing Slave	Sensor Slave	Enhanced Slave	Portable Controller	Static Controller	Installer Controller	Bridge Controller
Network Management								
Controller replication	-	-	-	-	X	X	X	X
Controller shift	-	-	-	-	X[1]	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	X	X	X	X	X
"I'm lost" – provide help	-	-	-	-	-	-	-	-
Provide routing table info	-	-	-	-	X	X	X	X

[1] Only when primary controller

CONFIDENTIAL

5.1.1.2 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

	Slave	**Routing Slave**	**Sensor Slave**	**Enhanced Slave**	**Portable Controller**	**Static Controller**	**Installer Controller**	**Bridge Controller**
Network Management								
Controller replication	-	-	-	-	X	X	X	X
Controller shift	-	-	-	-	X[1]	X[2]	X[1]	X[2]
Create new primary controller	-	-	-	-	-	X[3]	-	X[3]
Request network updates	-	X	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	X	X	X	X	
"I'm lost" – provide help	-	X[4]	X[5]	X	-	X	X[5]	X
Provide routing table info	-	-	-	-	X	X	X	X

[2] Only when primary controller and not SUC

[3] Only when SUC and not primary controller

[4] Only if "always listening"

CONFIDENTIAL

5.1.1.3 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

	Slave	Routing Slave	Sensor Slave	Enhanced Slave	Portable Controller	Static Controller	Installer Controller	Bridge Controller
Network Management								
Controller replication	-	-	-	-	X	X	X	X
Controller shift	-	-	-	-	-	-	-	-
Create new primary controller	-	-	-	-	-	-	-	-
Request network updates	-	X	X	X	X	X	X	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	X	X	X	X	
"I'm lost" – provide help	-	X[5]	X[5]	X	-	X	X[5]	X
Provide routing table info	-	-	-	-	X	X	X	X

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

[1] Only when primary/inclusion controller

[2] Only when primary controller

[5] Only if "always listening"

CONFIDENTIAL

5.1.1.4 Library Memory Usage

Each API library uses some of the 32KB flash and 2KB RAM available in the ZW0102/ZW0201. Refer to the software release note [20] regarding the minimum amount of flash and RAM that is available for an application build on the library in question. Using the debug functionality of the API will use up to 4K of additional flash and 60 bytes of RAM.

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

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

CONFIDENTIAL

5.2 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 Controller bridge 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 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_slave_api.h	Slave interface. Macro defines. Includes all other necessary header files.
ZW_slave_routing_api.h	Routing slave node interface. Macro definitions. 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_basis_api.h	Z-Wave ⇔ Application general software interface. Interface to common Z-Wave functions.
ZW_transport_api.h	Transfer of data via Z-Wave protocol.
ZW_classcmd.h	Defines for device and command classes used to obtain interoperability between Z-Wave enabled products from different vendors, for a detailed description refer to [1].

CONFIDENTIAL

Various header files	Description
ZW010x.h	Synopsys DW8051 SFR and ISR defines for the Z-Wave ZW010x RF transceiver.
ZW020x.h	Inventra m8051w SFR and ISR defines for the Z-Wave ZW020x RF transceiver.
ZW030x.h	Inventra m8051w SFR and ISR defines for the Z-Wave ZW030x RF transceiver.
ZW_adcdriv_api.h	ADC functionality.
ZW_appltimer.h	PWM/Timer function
ZW_debug_api.h	Debugging functionality via serial port.
ZW_eep_addr.h	Defines application EEPROM start address.
ZW_mem_api.h	EEPROM interface.
ZW_nodemask_api.h	Routines for manipulation of node ID lists organized as bit masks.
ZW_non_zero.h	Define none zero area of the SRAM (ZW0201/ZW0301 only). See also section 4.1and 7.3
ZW_power_api.h	ASIC power management functionality.
ZW_RF010x.h	Flash ROM RF table offset for the Z-Wave ZW010x
ZW_RF020x.h	Flash ROM RF table offset for the Z-Wave ZW020x
ZW_RF030x.h	Flash ROM RF table offset for the Z-Wave ZW030x
ZW_rtc_api.h	Real Time Clock functionality (ZW0102 only).
ZW_SerialAPI.h	Serial API interface with function ID defines etc.
ZW_sysdefs.h	CPU and clock defines.
ZW_timer_api.h	Timer functionality.
ZW_triac_api.h	TRIAC controller functionality.
ZW_typedefs.h	Common used defines (BYTE, WORD…).
ZW_uart_api.h	UART functionality.

CONFIDENTIAL

5.3 Z-Wave Common API

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

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

5.3.1 Required Application Functions

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

5.3.1.1 ApplicationInitHW

ZW0102 version:

BYTE ApplicationInitHW(void)

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

Defined in: ZW_basis_api.h

Return value:

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

Serial API (Not supported)

CONFIDENTIAL

ZW0201 and ZW0301 version:

BYTE ApplicationInitHW(BYTE bWakeupReason)

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

Defined in: ZW_basis_api.h

Return value:

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

Parameters:

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

Serial API (Not supported)

5.3.1.2 ApplicationInitSW

BYTE ApplicationInitSW(void)

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

Defined in: ZW_basis_api.h

Return value:

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

Serial API (Not supported)

CONFIDENTIAL

5.3.1.3 ApplicationTestPoll

void ApplicationTestPoll(void)

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

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

The following API calls are only available in production test mode:

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

Defined in: ZW_basis_api.h

Serial API (Not supported)

CONFIDENTIAL

5.3.1.4 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
Minimum	58 us	7.2 us	7.2 us
Maximum	3.8 ms	2.4 ms	2.4 ms

Defined in: ZW_basis_api.h

Serial API (Not supported)

CONFIDENTIAL

5.3.1.5 ApplicationCommandHandler

void ApplicationCommandHandler(BYTE rxStatus,
BYTE sourceNode,
ZW_APPLICATION_TX_BUFFER *pCmd,
BYTE cmdLength)

The Z-Wave protocol will call the **ApplicationCommandHandler** function when an application command or request has been received from another node. The receive buffer is released when returning from this function. The type of frame used by the request can be determined (single cast, mulitcast or broadcast frame). This is used to avoid flooding the network by responding on a multicast or broadcast.

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)
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[ ]

CONFIDENTIAL

5.3.1.6 ApplicationNodeInformation

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

The Z-Wave application layer use **ApplicationNodeInformation** to generate the Node Information frame and to save information about node capabilities. All the Z-Wave application related fields of the Node Information structure must be initialized in this function. The Generic Device Class, Specific Device Class, and Command Classes are described in [1].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) should only be set if the node is powered continuously and reside on a fixed position in the installation. This is because always listening nodes are included in the routing table to assist as repeaters in the network. The routing table is static during normal operation and this is the reason always listening nodes should not be moved around in the network.

In case the Listening bit is cleared (APPLICATION_NODEINFO_NOT_LISTENING) the node should be non-listening. This option will typically be selected for battery operated nodes that are power off RF reception when idle (prolongs battery lifetime). Even though a node is battery operated it should NOT normally be moved around in the network. The only exception is the Controller library which is targeted specifically to node types that move around in the network, such as a portable remote.

The Optional Functionality bit (APPLICATION_NODEINFO_OPTIONAL_FUNCTIONALITY) is used to indicate 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.

CONFIDENTIAL

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
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[ ]

CONFIDENTIAL

The figure below lists the Node Information Frame format. The frame is created by the Z-Wave Protocol using ApplicationNodeInformation.

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 7 Node Information frame format

CONFIDENTIAL

5.3.1.7 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 Application layer.

Defined in: ZW_slave_api.h

Parameters:

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

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

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

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

bLen IN The length of the pCmd parameter.

Serial API:

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

CONFIDENTIAL

5.3.1.8 ApplicationControllerUpdate (All controller libraries)

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

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

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

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

Defined in: ZW_controller_api.h

Parameters:

bStatus IN	The status of the update process, value could be one of the following:	
	UPDATE_STATE_ADD_DONE	A new node has been added to the controller and the network
	UPDATE_STATE_DELETE_DONE	A node has been deleted from the controller and 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

CONFIDENTIAL

ZW_RequestNodeInfo call.

5.3.1.9 ApplicationSlaveCommandHandler (Bridge Controller library only)

void ApplicationSlaveCommandHandler(BYTE rxStatus,
BYTE destNode,
BYTE sourceNode,
ZW_APPLICATION_TX_BUFFER *pCmd,
BYTE cmdLength)

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

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

Defined in: ZW_controller_bridge_api.h

Parameters:

rxStatus IN	Frame header info:	
	RECEIVE_STATUS_LOW_POWER	Received at low output power level.
	RECEIVE_STATUS_ROUTED_BUSY	A response route is locked by the application.
destNode IN	Command receiving virtual slave Node ID.	
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 | 0xA1 | rxStatus | destNode | sourceNode | cmdLength | pCmd[ ]

CONFIDENTIAL

5.3.1.10 ApplicationSlaveNodeInformation (Bridge Controller library only)

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

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

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

Defined in: ZW_controller_bridge_api.h

Parameters:

destNode IN	Which Virtual Node do we want the node information from.	
listening OUT	TRUE if this node is always listening and not moving.	
nodeType OUT	Pointer to structure with the Device Class:	
	(*nodeType).generic	The Generic Device Class [1]. Do not enter zero in this field.
	(*nodeType).specific	The Specific Device Class [1].
nodeParm OUT	Command Class buffer pointer.	Command Classes [1] supported by the device itself and optional Command Classes the device can control in other devices.
parmLength OUT	Number of Command Class bytes.	

Serial API:

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

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

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

CONFIDENTIAL

5.3.1.11 ApplicationRfNotify (ZW0301 only)

void ApplicationRfNotify (BYTE rfState)

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

- From Tx to Rx
- From Rx to Tx
- From powere down to Rx
- From power down to Tx
- When PA is powered up
- When PA is powered down

This enables the application to control an external PA using the appropriate number of I/O pins. It takes approximately 250 us to shift from Tx->Rx or Rx->Tx, so the settling time of the PA hardware must be faster.

When using an external PA, remember to set the field at FLASH_APPL_PLL_STEPUP_OFFS in App_RFSetup.a51 to 0 (zero) for adjustment of the signal quality.

This is necessary to be able to pass a FCC compliance test.

Defined in: ZW_basis_api.h

Parameters:

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

Serial API:

Not implemented

CONFIDENTIAL

5.3.2 Z-Wave Basis API

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

5.3.2.1 ZW_Poll

void ZW_Poll(void)

Macro: ZW_POLL

This Z-Wave low-level poll function handles the transfer of bytes from the RF input, and transfer of bytes to the RF output media. This function must therefore be called while waiting on HW ready states and when executing other time consuming functions.

In order not to disrupt the radio communication and the protocol no application function must execute code for more than 10ms without returning or calling the ZW_Poll function. Interrupt must not be disabled more than 8 bit receive time, which is around 0.8 ms.

Defined in: ZW_basis_api.h

Serial API (Not supported)

5.3.2.2 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 (Not supported)

CONFIDENTIAL

5.3.2.3 ZW_RFPowerLevelSet

BYTE ZW_RFPowerLevelSet(BYTE powerLevel)

Macro: ZW_RF_POWERLEVEL_SET(POWERLEVEL)

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

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

Defined in: ZW_basis_api.h

Parameters:

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

normalPower	Max power possible
minus1dB	Normal power - 1dB On the ZW0201 will this value be mapped into the next even value e.g. minus2dB
minus2dB	Normal power - 2dB
minus3dB	Normal power - 3dB On the ZW0201 will this value be mapped into the next even value e.g. minus4dB
minus4dB	Normal power - 4dB
minus5dB	Normal power - 5dB On the ZW0201 will this value be mapped into the next even value e.g. minus6dB
minus6dB	Normal power - 6dB
minus7dB	Normal power - 7dB On the ZW0201 will this value be mapped into the next even value e.g. minus8dB
minus8dB	Normal power - 8dB
minus9dB	Normal power - 9dB On the ZW0201 will this value be mapped into the next even value e.g. minus10dB

CONFIDENTIAL

Return value:

BYTE	The powerlevel set.

Serial API (Serial API protocol version 4):

HOST->ZW: REQ | 0x17 | powerLevel

ZW->HOST: RES | 0x17 | retVal

5.3.2.4 ZW_RFPowerLevelGet

BYTE ZW_RFPowerLevelGet(void)

Macro: ZW_RF_POWERLEVEL_GET()

Get the current power level used in RF transmitting.

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

Defined in: ZW_basis_api.h

Return value:

BYTE	The power level currently in effect during RF transmissions.

Serial API

HOST->ZW: REQ | 0xBA

ZW->HOST: RES | 0xBA | powerlevel

CONFIDENTIAL

5.3.2.5 ZW_RediscoveryNeeded (Not Slave and Bridge Controller Library)

BYTE ZW_RediscoveryNeeded (BYTE bNodeID,
VOID_CALLBACKFUNC (completedFunc)(BYTE txStatus))

Macro: ZW_REDISCOVERY_NEEDED(nodeid, func)

This function can be used to request a controller to update the nodes neighbors. The function will try to request a neighbor rediscovery from a controller in the network. In order to reach a controller it uses other nodes (bNodeID) in the network. It is left to the application to decide the algorithm that should be used for trying different bNodeID's.

If bNodeID supports this functionality (routing slave, enhanced slave and controller libraries), bNodeID will try to contact a 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. If ZW_ROUTE_UPDATE_DONE is returned bNodeID was able to contact a controller and the routing information have been updated.

If ZW_ROUTE_LOST_FAILED is returned the bNodeID was unable to help and the application can try next node if it decides so.

Defined in: ZW_controller_static_api.h and 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

CONFIDENTIAL

Callback function parameters:

ZW_ROUTE_LOST_ACCEPT	The node bNodeID was able to contact a controller, which then will initiate the re-discovery of the lost node.
ZW_ROUTE_LOST_FAILED	The node bNodeID could not help, try another node
ZW_ROUTE_UPDATE_ABORT	The node bNodeID do not respond, try another node
ZW_ROUTE_UPDATE_DONE	The rediscovery ended successfully.

Serial API (not supported)

5.3.2.6 ZW_RequestNetWorkUpdate (Not Slave Library)

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.

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.

CONFIDENTIAL

Callback function Parameters:

txStatus IN	Status of command:	
	ZW_SUC_UPDATE_DONE	The update process succeeded.
	ZW_SUC_UPDATE_ABORT	The update process aborted because of an error.
	ZW_SUC_UPDATE_WAIT	The SUC node is busy.
	ZW_SUC_UPDATE_DISABLED	The SUC functionality is disabled.
	ZW_SUC_UPDATE_OVERFLOW	The controller requested an update after more than 64 changes have occurred in the network. The update information is then out of date in respect to that controller. In this situation the controller have to make a replication before trying to request any new network updates.

Serial API:

HOST->ZW: REQ | 0x53 | funcID

ZW->HOST: RES | 0x53 | retVal

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

CONFIDENTIAL

5.3.2.7 ZW_RFPowerlevelRediscoverySet

void ZW_RFPowerlevelRediscoverySet(BYTE bNewPower)

Macro: ZW_RF_POWERLEVEL_REDISCOVERY_SET(bNewPower)

Set the reduced power level locally when finding neighbors. The protocol is automatically initialized with the default power levels used when finding neighbors as before this API call was introduced. The default power levels are -1dB for the 100 Series and -2dB for the 200 Series. It is only necessary to call ZW_RFPowerlevelRediscoverySet in case a value different from the default power levels is needed. Furthermore is it only necessary to set a new reduced power level once then the new level will be used every time a neighbour discovery is performed. The API call can be called from ApplicationInit or during runtime from ApplicationPoll or ApplicationCommandHandler.

NOTE: Be aware of that weak RF links can be included in the routing table in case the reduce power level is set to 0dB (normalPower). Weak RF links can increase latency in the network due to retries to get through. Finally will a large reduction in power level result in a reduced range between the nodes in the network resulting 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 On the ZW0201 will this value be mapped into the next even value e.g. minus2dB
	minus2dB	Normal power - 2dB
	minus3dB	Normal power - 3dB On the ZW0201 will this value be mapped into the next even value e.g. minus4dB
	minus4dB	Normal power - 4dB
	minus5dB	Normal power - 5dB On the ZW0201 will this value be mapped into the next even value e.g. minus6dB
	minus6dB	Normal power - 6dB
	minus7dB	Normal power - 7dB On the ZW0201 will this value be mapped into the next even value e.g. minus8dB

CONFIDENTIAL

minus8dB	Normal power - 8dB
minus9dB	Normal power - 9dB On the ZW0201 will this value be mapped into the next even value e.g. minus10dB

Serial API (Not supported)

5.3.2.8 ZW_SetPromiscuousMode (Not Bridge Controller library)

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

ZW_SetPromiscuousMode Enable / disable the promiscuous mode.

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

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

CONFIDENTIAL

5.3.2.9 ZW_SetRFReceiveMode

BYTE ZW_SetRFReceiveMode(BYTE mode)

Macro: ZW_SET_RX_MODE(mode)

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

Defined in: ZW_basis_api.h

Return value:

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

Parameters:

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

Serial API

HOST->ZW: REQ | 0x10 | mode

ZW->HOST: RES | 0x10 | retVal

CONFIDENTIAL

5.3.2.10 ZW_SetSleepMode

ZW0102 version:

void ZW_SetSleepMode(void)

Macro: ZW_SLEEP()

Set the CPU in sleep mode until woken by an interrupt. This function is used by battery operated devices in order to save power when idle. The CPU will first enter sleep mode when all internal protocol tasks is finished. The RF transceiver is turned off so nothing can be received while in sleep mode. RF transceiver will not be turned off for a device configured as listening. The ADC is not disabled during sleep mode. The Z-Wave main poll loop is stopped until the CPU is awake again. Any interrupt will wake up the CPU from sleep mode.

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

Be aware of the additional power consumption in case the internal power on reset (POR) circuit is enabled.

Defined in: ZW_basis_api.h

Serial API

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

The parameters mode and intEnable are ignored by the 100 Series but are added to enable support of the 200 Series.

CONFIDENTIAL

ZW0201 and ZW0301 version:

void ZW_SetSleepMode(BYTE mode, BYTE intEnable)

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

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

Be aware of the additional power consumption in case the internal power on reset (POR) circuit is enabled.

For more information on the best way to use this API call see section 5.3.9.

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

CONFIDENTIAL

Defined in: ZW_power_api.h

Parameters:

mode IN	Specify the type of power save mode:	
	ZW_STOP_MODE	The whole ASIC is turned down. The ASIC can be wakened up again by Hardware reset or by the external interrupt INT1.
	ZW_WUT_MODE	The ASIC is powered down, and it can only be waked by the timer timeout or by the external interrupt INT1. The time out value of the WUT can be set by the API call **ZW_SetWutTimeout**. When the ASIC is waked from the WUT_MODE, the API call **ZW_IsWutFired** can be used to test if the ASIC is waked up by timeout or INT1. The ASIC wake up from WUT mode from the reset state. The timer resolution in this mode is one second. The maximum timeout value is 256 secs.
	ZW_WUT_FAST_MODE	This mode has the same functionality as ZW_WUT_MODE, except that the timer resolution is 1/128 sec. The maximum timeout value is 2 secs. This mode is only available in ZW0301.
intEnable IN	Interrupt enable bit mask. If a bit mask is 1, the corresponding interrupt is enabled and this interrupt will wakeup the ASIC from power down. Valid bit masks are:	
	ZW_INT_MASK_EXT1	External interrupt 1 (PIN P1_7) is enabled as interrupt source
	0x00	No external Interrupts will wakeup. Usefull in WUT mode

Serial API

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

CONFIDENTIAL

5.3.2.11 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: REQ | 0x12 | funcID | txStatus

CONFIDENTIAL

5.3.2.12 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.

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 On the ZW0201 will this value be mapped into the next even value e.g. minus2dB
minus2dB	Normal power - 2dB
minus3dB	Normal power - 3dB On the ZW0201 will this value be mapped into the next even value e.g. minus4dB
minus4dB	Normal power - 4dB
minus5dB	Normal power - 5dB On the ZW0201 will this value be mapped into the next even value e.g. minus6dB
minus6dB	Normal power - 6dB
minus7dB	Normal power - 7dB On the ZW0201 will this value be mapped into the next even value e.g. minus8dB
minus8dB	Normal power - 8dB

CONFIDENTIAL

	minus9dB	Normal power - 9dB On the ZW0201 will this value be mapped into the next even value e.g. 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

CONFIDENTIAL

5.3.2.13 ZW_Type_Library

BYTE ZW_Type_Library(void)

Macro: ZW_TYPE_LIBRARY()

Get the Z-Wave library type.

Defined in: ZW_basis_api.h

Return value:

BYTE	Returns the library type as one of the following:	
	ZW_LIB_CONTROLLER_STATIC	Static controller library
	ZW_LIB_CONTROLLER_BRIDGE	Bridge controller library
	ZW_LIB_CONTROLLER	Portable controller library
	ZW_LIB_SLAVE_ENHANCED	Enhanced slave library
	ZW_LIB_SLAVE_ROUTING	Routing slave library
	ZW_LIB_SLAVE	Slave library
	ZW_LIB_INSTALLER	Installer library

Serial API

HOST->ZW: REQ | 0xBD

ZW->HOST: RES | 0xBD | retVal

CONFIDENTIAL

5.3.2.14 ZW_Version

BYTE ZW_Version(BYTE *buffer)

Macro: ZW_VERSION(buffer)

Get the Z-Wave basis API library version.

Defined in: ZW_basis_api.h

Parameters:

buffer OUT	Returns the API library version in text using the format: Z-Wave x.yy where x.yy is the library version.

Return value:

BYTE Returns the library type as one of the following:

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

Serial API:

HOST->ZW: REQ | 0x15

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

CONFIDENTIAL

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

HOST->ZW: REQ | 0x02

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

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

Nodes[29] is a node bitmask.

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

The chip used can be determined as follows:

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

Timer functions are: TimerStart, TimerRestart and TimerCancel.

5.3.2.15 ZW_VERSION_MAJOR / ZW_VERSION_MINOR / ZW_VERSION_BETA

Macro: ZW_VERSION_MAJOR/ZW_VERSION_MINOR/ ZW_VERSION_BETA

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

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

Defined in: ZW_basis_api.h

Serial API (Not supported)

CONFIDENTIAL

5.3.2.16 ZW_GetProtocolStatus

BYTE ZW_GetProtocolStatus(void)

Macro: ZW_GET_PROTOCOL_STATUS()

Report the status of the protocol.

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

Defined in: ZW_basis_api.h

Return value:

BYTE	Returns the protocol status as one of the following:	
	Zero	Protocol is idle.
	ZW_PROTOCOL_STATUS_ROUTING	Protocol is analyzing the routing table.
	ZW_PROTOCOL_STATUS_SUC	SUC sends pending updates.

Serial API

HOST->ZW: REQ | 0xBF

ZW->HOST: RES | 0xBF | retVal

CONFIDENTIAL

5.3.2.17 ZW_WatchDogEnable

void ZW_WatchDogEnable(void)

Macro: ZW_WATCHDOG_ENABLE()

Enables the ASIC's built in watchdog. By default, the watchdog is enabled.
The watchdog timeout interval depends on the hardware platform.

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

The watchdog must be kicked at least one time per interval. Failing to do so will cause the ASIC to be reset.
To avoid unintentional reset of the application during initialization, the watchdog may be kicked one or more times in the function **ApplicationInitSW**.

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

Defined in: ZW_basis_api.h

Serial API

HOST->ZW: REQ | 0xB6

5.3.2.18 ZW_WatchDogDisable

void ZW_WatchDogDisable(void)

Macro: ZW_WATCHDOG_DISABLE ()

Disable the ASIC's built in watchdog.

Defined in: ZW_basis_api.h

Serial API

HOST->ZW: REQ | 0xB7

CONFIDENTIAL

5.3.2.19 ZW_WatchDogKick

void ZW_WatchDogKick(void)

Macro: ZW_WATCHDOG_KICK ()

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

Defined in: ZW_basis_api.h

Serial API

HOST->ZW: REQ | 0xB8

CONFIDENTIAL

5.3.2.20 ZW_SetExtIntLevel (ZW0201/ZW0301 only)

void ZW_SetExtIntLevel(BYTE intSrc, BOOL triggerLevel)

Macro: ZW_SET_EXT_INT_LEVEL(SRC, TRIGGER_LEVEL)

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

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

Defined in: ZW_basis_api.h

Parameters:

intSrc IN	The external interrupt valid values:	
	ZW_INT0	External interrupt 0 (PIN P1_6)
	ZW_INT1	External interrupt 1 (PIN P1_7)
triggerLevel IN	The external interrupt trigger level:	
	TRUE	Set the interrupt trigger to high level /Rising edge
	FALSE	Set the the interrupt trigger to low level /Faling edge

Serial API

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

CONFIDENTIAL

5.3.3 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.

5.3.3.1 ZW_SendData

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

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.

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

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

The Routing Slave and Enhanced Slave nodes can add the TRANSMIT_OPTION_RETURN_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.

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.

CONFIDENTIAL

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's only the pointer there is passed to the transmit queue.

Defined in: ZW_transport_api.h

Return value:

BYTE	FALSE	If transmit queue overflow

CONFIDENTIAL

Parameters:

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 single cast the source routing info takes up to 6 bytes depending on the number of hops leaving minimum 48 bytes for the payload.
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 **(Controller API only)**	Request retransmission via repeater nodes (at normal output power level).
	TRANSMIT_OPTION_RETURN_ROUTE **(Routing Slave and Enhanced Slave API only)**	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.
completedFunc	Transmit completed call back function	

CONFIDENTIAL

Callback function Parameters:

txStatus	Transmit completion status:	
	TRANSMIT_COMPLETE_OK	Successfully
	TRANSMIT_COMPLETE_NO_ACK	No acknowledge is received before timeout from the destination node. Acknowledge is discarded in case it is received after the timeout.
	TRANSMIT_COMPLETE_FAIL	Not possible to transmit data because the Z-Wave network is busy (jammed).

Serial API:

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

ZW->HOST: RES | 0x13 | RetVal

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

CONFIDENTIAL

5.3.3.2 ZW_SendDataMeta (ZW0201/ZW0301 only)

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

Macro: ZW_SEND_DATA_META(nodeid, data, length, options, func)

NOTE: This function is only available in Z-Wave 0201 libraries.

Transmit streaming or bulk data in the Z-Wave network. The application must implement a delay of minimum 35ms after each ZW_SendDataMeta call to ensure that streaming data traffic don't prevent control data from getting through in the network. The API call ZW_SendDataMeta can be used by both slaves and controllers supporting 40kbps. The call also checks that the destination supports 40kbps except if it is a source based on a slave. 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).

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

NOTE: Allways use the completeFunc callback to determine when the next frame can be send. Calling the ZW_SendDataMeta 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 or if destination node is not 40kbit/s compatible

CONFIDENTIAL

Parameters:

destNodeID	IN Destination node ID	Node to send Meta data to. Should be 40kbit/s capable
pData	IN Data buffer pointer	Pointer to data buffer.
dataLength	IN Data buffer length	Length of buffer
txOptions	IN Transmit option flags:	The maximum 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.
	TRANSMIT_OPTION_LOW_POWER	Transmit at low output power level (1/3 of normal RF range).
	TRANSMIT_OPTION_ACK	Request the destination node to acknowledge the frame
	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 (Serial API protocol version 4):

HOST->ZW: REQ | 0x18 | destNodeID | dataLength | pData[ ] | txOptions | funcID

ZW->HOST: RES | 0x18 | RetVal

ZW->HOST: REQ | 0x18 | funcID | txStatus

CONFIDENTIAL

5.3.3.3 ZW_SendDataMulti

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

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

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

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.

NOTE: For Bridge Controller based applications only the Controller node personality (not the local virtual slave nodes) will receive the actual multicast frame, but all personalities (Controller node and all virtual slave nodes) will receive its designated singlecast frame (if any) if the multicast frame was transmitted with the transmit optionflag TRANSMIT_OPTION_ACK set.

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's only the pointer there is passed to the transmit queue.

Defined in: ZW_transport_api.h

Return value:

BYTE	FALSE	If transmit queue overflow

CONFIDENTIAL

Parameters:

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_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

CONFIDENTIAL

5.3.3.4 ZW_SendDataAbort

void ZW_SendDataAbort( )

Macro: ZW_SEND_DATA_ABORT

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

Defined in:	ZW_transport_api.h

Serial API:

HOST->ZW: REQ | 0x16

5.3.3.5 ZW_LockRoute

void ZW_LockRoute(BYTE bNodeID)

Macro: ZW_LOCK_RESPONSE_ROUTE(node)

The function is used to assure that the response route don’t disappear when transmitting a sequence of frames to the same node ID. This is especially important when trying to route to a node that normally moves around in the network, i.e. a battery operated controller.

This function lock and unlock only one response route for a given node ID. A buffer is allocated with space for two response routes. The buffer is overwritten every time the node gets a routed frame from another node.

Receiving a frame within direct range from a node that already exists as a response route destination will erase this response route in the buffer. The application has full control over when to lock and unlock the response route.

If the TRANSMIT_OPTION_RETURN_ROUTE flag is set the node use the return routes (see ZW_Assign_Return_Routes), but if the flag is not set it first checks if a response route exist for the destination node otherwise it try direct.

If a routed frame is received in the **ApplicationCommandHandler** while a response route is locked RECEIVE_STATUS_ROUTED_BUSY bit will be set in rxStatus. This flag can be used by the application to put another node on hold until the lock is released.

CONFIDENTIAL

Defined in: ZW_transport_api.h

Parameters:

bNodeID IN	Setting bNodeID to a response route destination node ID will lock it. If bNodeID is set to 0x00 the previous locked response route is unlocked

Serial API:

HOST->ZW: REQ | 0x90 | bNodeID

5.3.3.6 ZW_SendConst

void ZW_SendConst(void)

Macro: ZW_SEND_CONST()

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

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

Defined in: ZW_transport_api.h

Serial API (Not supported)

CONFIDENTIAL

5.3.4 Z-Wave TRIAC API

The built-in TRIAC Controller is targeted at TRIAC or FET controlled light / power dimming applications. For a detailed description of the TRIAC refer to [25] or [26] depending on the single chip used. The Application software can use the following TRIAC API calls to control the ZW0102/ZW0201/ZW0301 TRIAC Controller.

5.3.4.1 TRIAC_Init

ZW0102 version:

void TRIAC_Init(BYTE wiringType,
BYTE mainsfreq)

Macros:

ZW_TRIAC_INIT

ZW_TRIAC_INIT_2_WIRE

TRIAC_Init initializes the ZW0102 ASIC's integrated TRIAC for mainsfreq mains frequency usage and wiring system type. Sets the ZEROX and TRIAC pins up for triac control.

Defined in: ZW_triac_api.h

Parameters:

wiringType IN	Wiring types:	
	TRIAC_3_WIRE	3-wire TRIAC control. This kind of control provides improved efficiency, as well as reduced harmonics and mechanical noise.
	TRIAC_2_WIRE	2-wire TRIAC control
mainsfreq IN	Mains frequencies:	
	FREQUENCY_50HZ	Controlling a 50Hz AC mains supply
	FREQUENCY_60HZ	Controlling a 60Hz AC mains supply

Serial API (Not supported)

ZW0201 and ZW0301 version:

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

CONFIDENTIAL

Macros:

ZW_TRIAC_INIT

ZW_TRIAC_INIT_2_WIRE

TRIAC_Init initializes the ZW0201 ASIC's integrated TRIAC for mainsfreq AC mains frequency usage, bridge type, voltageDrop is the voltage drop across the ZERX input and the minimumPulse is the minimum pulse time of the TRIAC output signal. Configures the ZEROX and TRIAC pins for triac control.

Defined in: ZW_triac_api.h

Parameters:

bridgeType IN	Bridge types:	
	TRIAC_FULLBRIDGE	The TRIAC signal is triggered ONLY on the rising edge of the ZEROX signal which is fed through a FULL diode bridge.
	TRIAC_HALFBRIDGE	The TRIAC signal is triggered on the rising AND the falling edge of the ZEROX signal which is fed through a NON-FULL diode bridge.
mainsfreq IN	AC Mains frequencies:	
	FREQUENCY_50HZ	Controlling a 50Hz AC mains supply
	FREQUENCY_60HZ	Controlling a 60Hz AC mains supply
voltageDrop IN	Valid values for voltageDrop are from 0 to 2000mv with 100mv step.	The voltage drop values are defined as constants and are listed in the ZW_triac_api.h header file.
minimumPulse IN	Valid values for the triac minimum pulse are from 64 us to 512 us with 64 us step.	The minimum pulse values are defined as constants and are listed in the ZW_triac_api.h.

Serial API (Not supported)

CONFIDENTIAL

5.3.4.2 TRIAC_SetDimLevel

void TRIAC_SetDimLevel(BYTE dimLevel)

Macros:

ZW_TRIAC_DIM_SET_LEVEL(dimLevel)

ZW_TRIAC_LIGHT_SET_LEVEL(lightLevel)

TRIAC_SetDimLevel turns the triac controller ON and sets it to dim at dimLevel (1-99) or sets the light level to lightLevel (1-99) if the ZW_TRIAC_LIGHT_SET_LEVEL macro is used. This is done for the mains frequency selected in the TRIAC_Init function call (50/60Hz).

Defined in:	ZW_triac_api.h

Parameters:

dimLevel IN	Level (1…99)

Serial API (Not supported)

5.3.4.3 TRIAC_Off

void TRIAC_Off(void)

Macro:

ZW_TRIAC_OFF

TRIAC_Off turns the triac controller OFF.

Defined in:	ZW_triac_api.h

Serial API (Not supported)

CONFIDENTIAL

5.3.5 Z-Wave Timer API

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

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

Global tick counter:

WORD tickTime

5.3.5.1 TimerStart

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

Macro: ZW_TIMER_START(func,ticks,repeats)

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

Defined in: ZW_timer_api.h

Return value:

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

Parameters:

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

CONFIDENTIAL

Serial API (Not supported)

5.3.5.2 TimerRestart

BYTE TimerRestart(BYTE timerHandle)

Macro: ZW_TIMER_RESTART(handle)

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

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

Defined in: ZW_timer_api.h

Return value:

BYTE	TRUE	If timer restarted

Parameters:

timerHandle IN	Timer to restart

Serial API (Not supported)

5.3.5.3 TimerCancel

BYTE TimerCancel(BYTE timerHandle)

Macro: ZW_TIMER_CANCEL(handle)

Stop the specified timer.

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

Defined in: ZW_timer_api.h

Return value:

BYTE	TRUE	If timer cancelled

Parameters:

timerHandle IN	Timer number to stop

Serial API (Not supported)

CONFIDENTIAL

5.3.6 Z-Wave PWM API

The Z-Wave PWM API is an API that grants the application programmer protected access to a ZW0102/ZW0201/ZW0301 PWM / Timer interrupt. The hardware TIMER2 is used by the PWM API.

5.3.6.1 ZW_PWMSetup

BYTE ZW_PWMSetup (BYTE bValue)

Macro: ZW_PWM_SETUP(value)

Configure the mode and enables/disables the PWM/Timer.

Note:

ZW0102 version:	When configured as PWM the pin P3_4 (defined as bit register T0) will be enabled as output.
ZW0201/ZW0301 version:	When configured as PWM the pin P1_6 will be enabled as output.

Defined in: ZW_appltimer_api.h

Parameters (ZW0102 version):

bValue IN	PWM_MODE_BIT (bit 0)	
	0	TIMER mode
	1	PWM mode
	TIMER_RUN_BIT (bit 1)	
	0	Timer is inactive and counter is cleared.
	1	Timer is active and counter is enabled.
	(bit 2–7) don’t care.	

CONFIDENTIAL

Parameters (ZW0201/ZW0301 version):

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

Serial API (Not supported)

CONFIDENTIAL

5.3.6.2 ZW_PWMPrescale

BYTE ZW_PWMPrescale(BYTE bValueMSB, BYTE bValueLSB)

Macro ZW_PWM_PRESCALE(msb,lsb)

This function sets up either the PWM period or the timeout if timer mode is selected. CPU_FREQ is defined in a Z-Wave header file.

ZW0102 version:

PWM mode:

Total period:	Tpwm = 255*(msb+1)/CPU_FREQ
High time of PWM:	ThPWM = (msb+1)*lsb/CPU_FREQ

Timer mode (Interrupt period):

Tint = 255*(msb*256+lsb+1)/CPU_FREQ

ZW0201/ZW0301 version:

PWM mode:

Total period:	Tpwm = thPWM + tlPWM
High time of PWM:	thPWM = (msb*prescaler)/CPU_FREQ
Low time of PWM	tlPWM = (LSB*prescaler)/CPU_FREQ

Timer mode (Interrupt period):

Tint = (msb*256+lsb+1)*prescaler/CPU_FREQ

Defined in: ZW_appltimer_api.h

Parameters:

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

Serial API (Not supported)

CONFIDENTIAL

5.3.6.3 ZW_PWMClearInterrupt

BYTE ZW_PWMClearInterrupt(void)

Macro ZW _PWM_CLEAR_INTERRUPT()

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

Defined in: ZW_appltimer_api.h

Serial API (Not supported)

5.3.6.4 ZW_PWMEnable

BYTE ZW_PWMEnable(BOOL bValue)

Macro ZW_PWM_INT_ENABLE (value)

Enables or disables the ZW_PWM interrupt.

Defined in: ZW_appltimer_api.h

Parameters:

bValue IN	TRUE	Interrupt enabled.
	FALSE	Interrupt disabled.

Serial API (Not supported)

CONFIDENTIAL

5.3.7 Z-Wave Memory API

The memory application interface handles accesses to the application area of the EEPROM/flash.

Slave and routing slave nodes use flash for storing application data. Enhanced slave and all controller nodes use an external EEPROM for storing application data. The Z-Wave protocol uses the first part of the external EEPROM for home ID, node ID, routing table etc. The SPI interface on the ZW0x0x is used to access the external EEPROM. Therefore can the CLK, MOSI, and MISO pins only be used as GPIO by the application on a slave and routing slave.

When using the following memory functions, memory offset 0 is the first byte of the reserved area for application data.

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

5.3.7.1 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 intereste in reading the Node ID.

Defined in: ZW_mem_api.h

Parameters:

pHomeID OUT	Home-ID pointer
pNodeID OUT	Node-ID pointer

Serial API:

HOST->ZW: REQ | 0x20

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

CONFIDENTIAL

5.3.7.2 MemoryGetByte

BYTE MemoryGetByte(WORD offset)

Macro: ZW_MEM_GET_BYTE(offset)

Read one byte from the non-volatile memory

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

Serial API:

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

ZW->HOST: RES | 0x21 | RetVal

CONFIDENTIAL

5.3.7.3 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 works on EEPROM and it should be considered that the write operation have a somewhat long write time (2-5 msec.).

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

Defined in: ZW_mem_api.h

Return value:

BYTE	FALSE	If write buffer full.

Parameters:

offset IN	Application area offset
data IN	Data to store

Serial API:

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

ZW->HOST: RES | 0x22 | RetVal

CONFIDENTIAL

5.3.7.4 MemoryGetBuffer

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

Macro: ZW_MEM_GET_BUFFER(offset,buffer,length)

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

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

Defined in: ZW_mem_api.h

Parameters:

offset IN	Application area offset
buffer IN	Buffer pointer
length IN	Number of bytes to read

Serial API:

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

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

CONFIDENTIAL

5.3.7.5 MemoryPutBuffer

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

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

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

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

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

Defined in: ZW_mem_api.h

Return value:

BYTE	FALSE	If the buffer put queue is full.

Parameters:

offset IN	Application area offset	
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

CONFIDENTIAL

5.3.7.6 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)

5.3.7.7 ZW_MemoryFlush

void ZW_MemoryFlush(void)

Macro: ZW_MEM_FLUSH()

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

The data to be written to FLASH are not written immediately to the FLASH. Instead it is saved in a SRAM buffer and then written when the RF is not active. 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 Slave and 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)

CONFIDENTIAL

5.3.8 Z-Wave ADC API

The ADC API is both a slave and a controller application's interface to the ADC unit in the ZW0102/ZW0201/ZW0301. For a detailed description of the ZW0201/ZW0301 ADC refer to [27].

5.3.8.1 ADC_On (ZW0102 only)

void ADC_On( )

Macro: ZW_ADC_ON

Call turns the power on to the ADC unit. Be aware that the ADC is default powered off.

ZW0201 and ZW0301 platforms automatically turn on the ADC when ADC_Start is called.

Defined in: ZW_adcdriv_api.h

Serial API (Not supported)

5.3.8.2 ADC_Off

void ADC_Off( )

Macro: ZW_ADC_OFF

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

Defined in: ZW_adcdriv_api.h

Serial API (Not supported)

5.3.8.3 ADC_Start

void ADC_Start( )

Macro: ZW_ADC_START

Start the conversion process.

Defined in: ZW_adcdriv_api.h

Serial API (Not supported)

CONFIDENTIAL

5.3.8.4 ADC_Stop

void ADC_Stop( )

Macro: ZW_ADC_STOP

Stop the conversion process.

Defined in: ZW_adcdriv_api.h

Serial API (Not supported)

CONFIDENTIAL

5.3.8.5 ADC_Init

ZW0102 version:

Void ADC_Init(BYTE mode, BYTE ref, BYTE pin)

Macro: ZW_ADC_INIT(MODE,REFRENCE,INPUT)

Initialize the ADC unit to work in the wanted conversion mode etc. Remember to power on the ADC unit before configuring it. The ADC unit can work in one of four different modes.

The ADC unit has 2 multiplexed inputs. And can be referenced by either the VCC voltage or internal bandgap reference voltage

Defined in: ZW_adcdriv_api.h

Parameters:

mode IN	The ADC mode to be used the mode value can be on of the following:	
	ADC_SINGLE_MODE	Single conversion mode: The ADC will always stop after one conversion. The ADC should be started each time a conversion is wanted.
	ADC_MULTI_CON_MODE	Multi conversion continues mode: The ADC will always sample the input until the ADC is stopped.
	ADC_MULTI_STP_MODE	Multi conversion stop mode: The ADC will always sample the input until either the user stops the ADC or the sampled value is equal or greater than a specified threshold.
	ADC_MULTI_RST_MODE	Multi conversion reset generating mode: The ADC will always sample the input until either the user stops the ADC or the sampled value is equal or greater than a specified threshold, in that case the Adc will generate a reset signal to the CPU.

CONFIDENTIAL

ref IN	The source of the reference voltage used for the ADC:	
	ADC_REF_VDD	Internal voltage reference is VDD (supply voltage).
	ADC_REF_V125	Internal voltage reference is 1.25V generated internally.
pin IN	Select the I/O pin to be used as ADC input:	
	ADC_PIN_1	Equal to ADC1 in hardware documentation.
	ADC_PIN_2	Equal to ADC2 in hardware documentation.

CONFIDENTIAL

ZW0201/ZW0301 version:

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

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

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

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

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

Defined in: ZW_adcdriv_api.h

Parameters:

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

CONFIDENTIAL

pin_en IN	Enabling of the pins to be used by the ADC. To enabled pin 1 and pin 3 the value should be set to: ADC_PIN_1 | ADC_PIN_3.	
	ADC_PIN_1 (I/O P0.0)	Equal to ADC0 in hardware documentation.
	ADC_PIN_2 (I/O P0.1)	Equal to ADC1 in hardware documentation.
	ADC_PIN_3 (I/O P1.0)	Equal to ADC2 in hardware documentation.
	ADC_PIN_4 (I/O P1.1)	Equal to ADC3 in hardware documentation.
	ADC_PIN_BATT	Using the ADC as battery monitor for battery operated devices. Regarding details refer to [21]

Serial API (Not supported)

CONFIDENTIAL

5.3.8.6 ADC_SelectPin

ADC_SelectPin(BYTE adcPin);

Macro:

ZW_ADC_SELECT_AD1 -select pin 1 as the ADC input

ZW_ADC_SELECT_AD2 - select pin 2 as the ADC input

ZW_ADC_SELECT_AD3 - select pin 3 as the ADC input (ZW0201/ZW0301 only)

ZW_ADC_SELECT_AD4 - select pin 4 as the ADC input (ZW0201/ZW0301 only)

Select a pin to use as the active ADC input.

Defined in: ZW_adcdriv_api.h

Parameters:

adcPin IN	The pin to use as ADC input:	
	ADC_PIN_1	
	ADC_PIN_2	
	ADC_PIN_3	ZW0201/ZW0301 only
	ADC_PIN_4	ZW0201/ZW0301 only

Serial API (Not supported)

CONFIDENTIAL

5.3.8.7 ADC_Buf (ZW0201/ZW0301 only)

ADC_Buf (BYTE enable);

Macro:

ZW_ADC_BUFFER_ENABLE - Enable the input buffer.

ZW_ADC_BUFFER_DISABLE - Disable the input buffer.

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

Defined in: ZW_adcdriv_api.h

Parameters:

enable IN	Switch the input buffer on/off:	
	TRUE	Switch the input buffer on.
	FALSE	Switch the input buffer off.

Serial API (Not supported)

CONFIDENTIAL

5.3.8.8 ADC_SetAZPL (ZW0201/ZW0301 only)

ADC_SetAZPL (BYTE azpl);

Macro: ZW_ADC_SET_AZPL(PERIOD)

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

Defined in: ZW_adcdriv_api.h

Parameters:

apzl IN	Length of the ADC auto zero period:	
	ADC_AZPL_1024	Set the sample period to 1024 clocks, valid for high impedance sources. Only valid for 8 bit resolution.
	ADC_AZPL_512	Set the sample period to 512 clocks, valid for medium to high impedance sources. Only valid for 8 bit resolution.
	ADC_AZPL_256	Set the sample period to 256 clocks, valid for medium to low impedance sources. Only valid for 8 bit resolution.
	ADC_AZPL_128	Set the sample period to 128 clocks, valid for low impedance sources. Valid for both 8 bit and 12 bit resolution.

Serial API (Not supported)

5.3.8.9 ADC_SetResolution (ZW0201/ZW0301 only)

ADC_SetResolution (BYTE reso);

Macro:

ZW_ADC_RESOLUTION_8 - Set the ADC resolution to 8 bit.

ZW_ADC_RESOLUTION_12 - Set the ADC resolution to 12 bit.

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

CONFIDENTIAL

Defined in: ZW_adcdriv_api.h

Parameters:

reso IN	The resolution of the ADC	
	ADC_8_BIT	Set the ADC resolution to 8 bit.
	ADC_12_BIT	Set the ADC resolution to 12 bit.

Serial API (Not supported)

5.3.8.10 ADC_SetThresMode (ZW0201/ZW0301 only)

ADC_SetThresMode (BYTE thresMode);

Macro:

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

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

Set the ADC threshold type.

Defined in: ZW_adcdriv_api.h

Parameters:

thresMode IN	The ADC threshold mode.	
	ADC_THRES_UPPER	The ADC fire when input is above/equal to the threshold value.
	ADC_THRES_LOWER	The ADC fire when input is below/equal to the threshold value.

Serial API (Not supported)

CONFIDENTIAL

5.3.8.11 ADC_SetThres

ZW0102 version:

void ADC_SetThres(BYTE thresHold)

Macro: ZW_ADC_SET_THRESHOLD(THRES)

Set the ADC threshold value. The threshold value is used to trigger an event when the 8 MSB of the sampled value is equal or greater than the threshold value. The event triggered depend on the ADC mode:

Single conversion mode: The ADC interrupt routine will be called and the ADC will stop converting.

Multi conversion continues mode: The ADC interrupt routine will be called and the ADC will continue converting.

Multi conversion stop mode: The ADC interrupt routine will be called, and the ADC will stop converting.

Multi conversion reset generating mode: No interrupt will be called but the ADC will generate a reset signal to the CPU.

Defined in: ZW_adcdriv_api.h

Parameters:

thresHold IN	The ADC threshold value, it ranges from 0 to 255.

Serial API (Not supported)

ZW0201/ZW0301 version:

void ADC_SetThres(WORD thresHold)

Macro: ZW_ADC_SET_THRESHOLD(THRES)

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

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

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

Defined in: ZW_adcdriv_api.h

Parameters:

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

Serial API (Not supported)

CONFIDENTIAL

5.3.8.12 ADC_Int

void ADC_Int(BYTE enable)

Macro:

ZW_ADC_INT_ENABLE

ZW_ADC_INT_DISABLE

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

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

Defined in: ZW_adcdriv_api.h

Parameters:

enable IN	The start of the ADC interrupt routine.	
	TRUE	Enables ADC interrupt.
	FALSE	Disables ADC interrupt.

Serial API (Not supported)

5.3.8.13 ADC_IntFlagClr

void ADC_IntFlagClr()

Macro: ZW_ADC_CLR_FLAG

Clear the ADC interrupt flag.

Defined in: ZW_adcdriv_api.h

Serial API (Not supported)

CONFIDENTIAL

5.3.8.14 ADC_SetSamplingRate (ZW0102 only)

void ADC_SetSamplingRate(BYTE rate)

Macro: ZW_ADC_SAMPLE_RATE(RATE)

Set the sampling rate of the ADC unit.

The ADC sampling rate can be set to on of 64 discrete frequencies. The sampling rate frequency can be calculated as follow:

ADC sampling rate in Hz = CPU_FREQ/(176*rate), where rate = 1 to 64

NOTE: The rate value must not be selected to a value that results in a sampling rate frequency above 22.7 kHz. If for example the CPU_FREQ is 7.376974MHz. The lowest rate is 2 yielding a sample rate frequency of 20.96KHz

Defined in: ZW_adcdriv_api.h

Parameters:

rate IN	The sampling rate of the ADC unit. The rate value can be from 1 to 64.

Serial API (Not supported)

CONFIDENTIAL

5.3.8.15 ADC_GetRes

WORD ADC_GetRes()

Macro: ZW_ADC_GET_READING

ZW0102 version:

The call returns the result of the ADC conversion. Only the first 10 LSB are used. In single conversion mode the call will return the value ADC_NOT_FINISHED in case conversion isn't finished yet. In the other modes the call return the latest generated sample.

Defined in: ZW_adcdriv_api.h

Return value:

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

Serial API (Not supported)

ZW0201/ZW0301 version:

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

Defined in: ZW_adcdriv_api.h

Return value:

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

Serial API (Not supported)

CONFIDENTIAL

5.3.9 Z-Wave Power API

The purpose of the Power API is to define functions that make it easy for the Z-Wave application developer to use the power management capabilities of the ZW0102/ZW0201/ZW0301 ASIC. The API can be used both for slave and controller applications.

In ZW0102 the lowest possible power consumption that can be reached when in sleep mode is achieved by running with 32Khz oscillator and powering down the main clock oscillator. In controllers and enhanced slaves this is done in the protocol level otherwise the clock switch should be done in the application layer.

Controllers and Enhanced slaves: when the protocol is idle, the RF will be shut down if the listening property is off. The protocol then will switch to the 32kHz oscillator and power down the main crystal oscillator. After that the ZW0102 will go in power down mode. The ZW0102 will be waked by the RTC interrupt. The ZW0102 can also be awaked by external input interrupt, ADC interrupts, or the timer interrupts if they are enabled.

Slaves with listening property Enabled: when the protocol is idle then the ZW0102 will go in power down mode. The ZW0102 can be awaked by the same interrupts as in the case of enhanced slaves or in addition by the RF interrupt. If less power consumption is wanted, then the RF should be power downed. Enhanced slaves can be switched to the 32kHz oscillator and the main crystal oscillator should be powered down. If this is done, then the RF interrupt cannot wake the ZW012.

Normal slaves with listening property disabled: when the protocol is idle then the RF will be shut down. The protocol then will go in power down mode. The ZW0102 can be awaked by the same interrupts as in the case of controllers and enhanced slave. If less power consumption is required, the PWR_SET_STOP_MODE can be used.

5.3.9.1 PWR_SetStopMode (ZW0102 only)

void PWR_SetStopMode

Macro: ZW_PWR_SET_STOP_MODE

PWR_SetStopMode This function used to set the ASIC in the lowest power mode called stop mode. In this mode the CPU will halt and all the ASIC internal peripherals except for the ADC will stop. The only way to wake up from this mode is either by power recycle or system reset.

Note: In controllers and enhanced slaves the Z-Wave library switch to 32kHz clock and shut down the main oscillator. Doing this enable the ADC to wake up the ZW0102.

Defined in: ZW_power_api.h

Serial API

HOST->ZW: REQ | 0xB0

CONFIDENTIAL

5.3.9.2 PWR_Clk_PD (ZW0102 only)

void PWR_Clk_PD (BYTE flags)

Macro: ZW_PWR_CLK_POWERDOWN(FLAGS)

PWR_Clk_PD shut down the clock oscillators specified in the flags parameter.

Note: Powering off both clock oscillator will put the ASIC in the lowest possible power mode. The only way to get out from this mode is by power recycling the ASIC or by system reset.

Defined in: ZW_power_api.h

Parameters:

flags IN	The value of the flag input parameter is one or both of the following defines:	
	ZW_PWR_MAIN_CLK	The system clock
	ZW_PWR_32K_CLK	The 32 kHz clock

Serial API

HOST->ZW: REQ | 0xB1 | flags

5.3.9.3 PWR_Clk_Pup (ZW0102 only)

void PWR_Clk_PUp (BYTE flags)

Macro: ZW_PWR_CLK_POWERUP(FLAGS)

PWR_Clk_PUp Power up the clock oscillators specified in the flags parameter.

Defined in: ZW_power_api.h

Parameters:

flags IN	The value of the flag input parameter is one or both of the following defines:	
	ZW_PWR_MAIN_CLK	The system clock
	ZW_PWR_32K_CLK	The 32 kHz clock

Serial API

HOST->ZW: REQ | 0xB2 | flags

CONFIDENTIAL

5.3.9.4 PWR_Select_Clk (ZW0102 only)

void PWR_Select_Clk (BYTE clock)

Macro: ZW_PWR_SELECT_CLK(CLOCK)

PWR_Select_Clk Select the clock that the ASIC will run on.

Note: Selecting a 32 kHz clock, as the system clock will cause the system timer to not function properly. Also RF will always run on the main clock regardless of the clock source for the ASIC.

Defined in: ZW_power_api.h

Parameters:

clock IN	Clock option flags:	
	ZW_SELECT_MAIN_CLK	The primary clock (7.376974MHz)
	ZW_SELECT_32K_CLK_INT	The real time clock (32.768kHz) using an external crystal in combination with an internal oscillator.
	ZW_SELECT_32K_CLK_EXT	The real time clock (32.768kHz) using an external clock signal.

Serial API

HOST->ZW: REQ | 0xB3 | clock

5.3.9.5 ZW_SetWutTimeout (ZW0201/ZW0301 only)

void ZW_SetWutTimeout (BYTE wutTimeout)

Macro: ZW_SET_WUT_TIMEOUT(TIME)

ZW_SetWutTimeout Set the Time out value of the wake up timer before going in WUT mode.

Defined in: ZW_power_api.h

Parameters:

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

Serial API

HOST->ZW: REQ | 0xB4 | wutTimeout

CONFIDENTIAL

5.3.9.6 ZW_IsWutKicked (ZW0201/ZW0301 only)

BOOL ZW_IsWutKicked (void)

Macro: ZW_IS_WUT_KICKED()

Use this function after ASIC wake up from WUT mode to test if it was waked by the WUT time out, external interrupt or hardware reset.

Defined in: ZW_power_api.h

Return value:

BOOL	TRUE	If the ASIC was waked by the WUT timeout.
	FALSE	If the ASIC was reset by hardware reset or by external interrupt 1.

Serial API

HOST->ZW: REQ | 0xB5
ZW->HOST: REQ | 0xB5 | wutKickedStatus

5.3.10 UART interface API

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

5.3.10.1 UART_Init

void UART_Init(WORD baudRate)

Macro: ZW_UART_INIT(baud)

Initializes the MCU's integrated UART.

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

Defined in: ZW_uart_api.h

Parameters:

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

Serial API (Not supported)

CONFIDENTIAL

5.3.10.2 UART_RecStatus

BYTE UART_RecStatus(void)

Macro: ZW_UART_REC_STATUS

Read the UART receive data status

Defined in: ZW_uart_api.h

Return value:

BYTE	TRUE	If data received.

Serial API (Not supported)

5.3.10.3 UART_RecByte

BYTE UART_RecByte(void)

Macro: ZW_UART_REC_BYTE

Function receives a byte over the UART.

This function waits until data received. See also: UART_Read

Defined in: ZW_uart_api.h

Return value:

BYTE	Received data.

Serial API (Not supported)

CONFIDENTIAL

5.3.10.4 UART_SendStatus

BYTE UART_SendStatus(void)

Macro: ZW_UART_SEND_STATUS

Read the UART send data status.

Defined in: ZW_uart_api.h

Return value:

BYTE	TRUE	If transmitter busy

Serial API (Not supported)

5.3.10.5 UART_SendByte

void UART_SendByte(BYTE data)

Macro: ZW_UART_SEND_BYTE(data)

Function transmits a byte over the UART.

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

Defined in: ZW_uart_api.h

Parameters:

data IN	Data to send.

Serial API (Not supported)

CONFIDENTIAL

5.3.10.6 UART_SendNum

void UART_SendNum(BYTE data)

Macro: ZW_UART_SEND_NUM(data)

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

Defined in: ZW_uart_api.h

Parameters:

data IN	Data to convert and send.

Serial API (Not supported)

5.3.10.7 UART_SendStr

void UART_SendStr(BYTE *str)

Macro: ZW_UART_SEND_STRING(str)

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

Defined in: ZW_uart_api.h

Parameters:

str IN	String pointer.

Serial API (Not supported)

5.3.10.8 UART_SendNL

void UART_SendNL(void)

Macro: ZW_UART_SEND_NL

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

Defined in: ZW_uart_api.h

Serial API (Not supported)

CONFIDENTIAL

5.3.10.9 UART_Enable (ZW0201/ZW0301 only)

void UART_Enable(void)

Macro: ZW_UART_ENABLE

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

Defined in: ZW_uart_api.h

Serial API (Not supported)

5.3.10.10 UART_Disable (ZW0201/ZW0301 only)

void UART_Disable(void)

Macro: ZW_UART_DISABLE

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

Defined in: ZW_uart_api.h

Serial API (Not supported)

5.3.10.11 UART_ClearTx (ZW0201/ZW0301 only)

void UART_ClearTx(void)

Macro: ZW_UART_CLEAR_TX

Clear the UART transmit done flag.

Defined in: ZW_uart_api.h

Serial API (Not supported)

CONFIDENTIAL

5.3.10.12 UART_ClearRx (ZW0201/ZW0301 only)

void UART_ClearRx(void)

Macro: ZW_UART_CLEAR_RX

Clear the UART receiver ready flag.

Defined in: ZW_uart_api.h

Serial API (Not supported)

5.3.10.13 UART_Write (ZW0201/ZW0301 only)

void UART_Write(BYTE txByte)

Macro: ZW_UART_WRITE(TXBYTE)

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

Defined in: ZW_uart_api.h

Parameters:

txByte IN	Data to send.

Serial API (Not supported)

CONFIDENTIAL

5.3.10.14 UART_Read (ZW0201/ZW0301 only)

BYTE UART_Read(void)

Macro: ZW_UART_READ

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

Defined in: ZW_uart_api.h

Return value:

BYTE	The contents of the UART receive register.

Serial API (Not supported)

5.3.10.15 Serial debug output.

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

5.3.10.15.1 ZW_DEBUG_INIT(baud)

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

Example:

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

Defined in: ZW_uart_api.h

Serial API (Not supported)

CONFIDENTIAL

5.3.10.15.2 ZW_DEBUG_SEND_BYTE(data)

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

Example:

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

Defined in: ZW_uart_api.h

Serial API (Not supported)

5.3.10.15.3 ZW_DEBUG_SEND_NUM(data)

Example:

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

Defined in: ZW_uart_api.h

Serial API (Not supported)

5.3.11 Z-Wave Real Time Clock API (ZW0102 only)

The Application software can use the following Real Time Clock (RTC) functions to do different actions at user specified times. The RTC is a 7 days/24 hours clock having a time base of one minute.

Note: The RTC API is only available on the Enhanced Rourting Slave, Portable Controller, Installer Controller and Static Controller libraries based on the ZW0102 Single Chip.

The RTC API uses the ZW010x built-in RTC timer to generate an interrupt every minute. The interrupt will wakeup the CPU if in sleep mode. The interrupt function sets a signal each minute, and the list of RTC timers will be checked against the current time.

At system startup the RTC is initiated to 1.00:00 (monday after midnight).

Time is specified by the following structure:

```
typedef struct _CLOCK_ {
  BYTE weekday;  /* Weekday 1:monday...7:sunday   */
  BYTE hour;     /* Hour 0...23                   */
  BYTE minute;   /* Minute 0...59                 */
} CLOCK;
```

The weekday is defined as follow:

```
RTC_MONDAY
RTC_TUESDAY
RTC_WEDNESDAY
RTC_THURSDAY
```

CONFIDENTIAL

```
RTC_FRIDAY
RTC_SATURDAY
RTC_SUNDAY
RTC_WORKDAYS    Monday…Friday
RTC_WEEKEND     Saturday, Sunday
RTC_ALLDAYS     Monday…Sunday
```

A RTC timer element is specified by the following structure:

```
typedef struct _RTC_TIMER_ {
  BYTE  status;   /* Timer element status                                */
  CLOCK timeOn;   /* Weekday, all days, work days or weekend callback time   */
  CLOCK timeOff;  /*  , Hour callback time                               */
                  /*  , Minute callback time                             */
  BYTE  repeats;  /* Number of callback times.
                     Predefined values:
                       RTC_TIMER_ONE_TIME
                       RTC_TIMER_FOREVER (0xFF)   */
  VOID_CALLBACKFUNC(func)(BYTE status, BYTE parm); /* Timer function address */
  BYTE parm;      /* Parameter that is returned to the timer function       */
} RTC_TIMER;
```

RTC timer callback function parameters (func)

status	Bit 0-3 is specified by the Application when creating the timer.
RTC_STATUS_FIRED	is set when “On” time, and not set when “Off” time.
parm	Specified by application when creating the timer

5.3.11.1 ClockSet

BYTE ClockSet(CLOCK *pNewTime)

Macro: ZW_CLOCK_SET(pNewTime)

ClockSet write the specified time to the current Real Time Clock. The specified time is validated before setting the Real Time Clock. For making the first minute as accurate as possible the node must not go into sleep mode before a full minute has past the **ClockSet** call.

Defined in: ZW_rtc_api.h

Return value:

BYTE	FALSE if invalid time value

Parameters:

pNewTime IN	New time value to set.

Serial API:

HOST->ZW: REQ | 0x30 | pNewTime[ ](3 bytes) = weekday,hour,minute

ZW->HOST: RES | 0x30 | RetVal

CONFIDENTIAL

5.3.11.2 ClockGet

void ClockGet(CLOCK *pTime)

Macro: ZW_CLOCK_GET(pTime)

Copy the current Real Time Clock time to the specified time buffer.

Defined in: ZW_rtc_api.h

Parameters:

pTime OUT	Pointer to a time memory buffer.

Serial API:

HOST->ZW: REQ | 0x31

ZW->HOST: RES | 0x31 | pTime[ ] (3bytes) = weekday,hour,minute

5.3.11.3 ClockCmp

Char ClockCmp(CLOCK *pTime)

Macro: ZW_CLOCK_CMP(pTime)

ClockCmp is used to compare a specified time against the current Real Time Clock time.

Defined in: ZW_rtc_api.h

Return value:

Char	1	Specified time is before current time.
	0	Specified time is equal current time.
	-1	Specified time is past current time.

Parameters:

pTime IN	Pointer to a time memory buffer.

Serial API:

HOST->ZW: REQ | 0x32 | pTime[ ] (3bytes) = weekday,hour,minute

ZW->HOST: RES | 0x32 | char (1 byte)

CONFIDENTIAL

5.3.11.4 RTCTimerCreate

BYTE RTCTimerCreate(RTC_TIMER *timer, VOID_CALLBACKFUNC(func)(void))

Macro: ZW_RTC_CREATE(timer, func)

RTCTimerCreate creates a new timer element.

RTCTimerCreate's callback function is called when the RTC system completed the timer element creation. The RTC timer element callback function defined in the beginning of this paragraph is called at the specified "On" and "Expire" time events. The RTC timer is running during sleep mode and the timer callback function will be called again ("On" time) in case the device wakeup from sleep mode.

The timer element is kept in the EEPROM so that it would not be lost if the system is reset. The timer element data buffer can be reused when the completion function (**func**) is called.

Defined in: ZW_rtc_api.h

Return value:

BYTE	RTC timer handle (0xFF if failed)

Parameters:

timer IN	Pointer to the new timer structure
func IN	Timer element read buffer completed function pointer.

Serial API:

HOST->ZW: REQ | 0x33 | RTC_TIMER.status | RTC_TIMER.timeOn (3 bytes) | RTC_TIMER.timeOff (3 bytes) | RTC_TIMER.repeats | RTC_TIMER.parm | RTC_TIMER.funcID | funcID

ZW->HOST: RES | 0x33 | RetVal

ZW->HOST: REQ | 0x33 | funcID

When the Timer callback function is called

ZW->HOST: REQ | 0x36 | RTC_TIMER.funcID | status | parm

CONFIDENTIAL

5.3.11.5 RTCTimerRead

BYTE RTCTimerRead(BYTE *timerHandle, RTC_TIMER *timer)

Macro: ZW_RTC_READ(handle, timer)

Search in the RTC list and if a used element found, copy the timer element from the RTC timer list to the specified memory buffer.

Defined in: ZW_rtc_api.h

Return value:

BYTE	Next RTC timer handle in the list. 0xFF if no timer copied to memory buffer (end of list).

Parameters:

timerHandle IN/OUT	Timer handle pointer, handle value to start search from in the timer list. The returned handle value is the found timer handle (“Return value” not equal 0xFF).
timer IN	Pointer to RTC timer memory buffer.

Serial API:

HOST->ZW: REQ | 0x34 | timerHandle

ZW->HOST: RES | 0x34 | RetVal | timerHandle | RTC_TIMER.status | RTC_TIMER.timeOn (3 bytes) | RTC_TIMER.timeOff (3 bytes) | RTC_TIMER.repeats | RTC_TIMER.parm

CONFIDENTIAL

5.3.11.6 RTCTimerDelete

void RTCTimerDelete(BYTE timerHandle)

Macro: ZW_RTC_DELETE(handle)

RTCTimerDelete remove the specified timer element from the RTC timer list.

Defined in: ZW_rtc_api.h

Parameters:

timerHandle IN Timer element to remove

Serial API:

HOST->ZW: REQ | 0x35 | timerhandle

CONFIDENTIAL

5.3.12 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.

5.3.12.1 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)

5.3.12.2 ZW_NodeMaskClearBit

void ZW_NodeMaskClearBit(BYTE_P pMask, BYTE bNodeID)

Macro: ZW_NODE_MASK_CLEAR_BIT(pMask, bNodeID)

Clear the node bit in a node bit mask.

Defined in: ZW_nodemask_api.h

Parameters:

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

Serial API (Not supported)

CONFIDENTIAL

5.3.12.3 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)

5.3.12.4 ZW_NodeMaskBitsIn

BYTE ZW_NodeMaskBitsIn(BYTE_P pMask, BYTE bLength)

Macro: ZW_NODE_MASK_BITS_IN (pMask, bLength)

Number of bits set in node mask.

Defined in: ZW_nodemask_api.h

Return value:

BYTE	Number of bits set in node mask

Parameters:

pMask	IN Pointer to node mask
bLength	IN Length of node mask

Serial API (Not supported)

CONFIDENTIAL

5.3.12.5 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)

CONFIDENTIAL

5.4 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…).

5.4.1 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 don't want to learn new nodes but only get node info from nodes that are already known in the system.

Normally low power transmission mode is used during node inclusion. The option ADD_NODE_OPTION_HIGH_POWER can be added to the bMode parameter for High Power inclusion.

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

CONFIDENTIAL

	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 High Power inclusion
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_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
*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[ ]

CONFIDENTIAL

5.4.2 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 only be NULL if remove is turned off).	

Callback function Parameters (completedFunc):

*learnNodeInfo.bStatus Status of learn mode:

CONFIDENTIAL

IN	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[ ]

CONFIDENTIAL

5.4.3 ZW_ControllerChange

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

Macro: ZW_CONTROLLER_CHANGE(mode, func)

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

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

Defined in: ZW_controller_api.h

Parameters:

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

CONFIDENTIAL

Callback function Parameters (completedFunc):

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

Serial API:

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

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

CONFIDENTIAL

5.4.4 ZW_SetLearnMode

**void ZW_SetLearnMode (BOOL 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
2. The controller receives and stores the node table and routing table for the network
3. 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.

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:	
	TRUE	Start the learn mode on the controller
	FALSE	Stop learn mode on the controller
completedFunc IN	Callback function pointer (Should only be NULL if state is turned off).	

CONFIDENTIAL

Callback function Parameters (completedFunc):

*learnNodeInfo.bStatus IN	Status of learn mode:	
	LEARN_MODE_STARTED	The learn process has been started
	LEARN_MODE_DONE	The learn process is complete and the controller is now included into the network
	LEARN_MODE_FAILED	The learn process failed.
*learnNodeInfo.bSource IN	Node id of the new node	
*learnNodeInfo.pCmd IN	Pointer to Application Node information data (see **ApplicationNodeInformation** - nodeParm**)**. NULL if no information present. The pCmd only contain information when bLen is not zero, so the information should be stored when that is the case. Regardless of the bStatus.	
*learnNodeInfo.bLen IN	Node info length.	

Serial API:

HOST->ZW: REQ | 0x50 | mode | funcID

ZW->HOST: REQ | 0x50 | funcID | bStatus | bSource | bLen | pCmd[ ]

CONFIDENTIAL

5.4.5 ZW_GetControllerCapabilities

BYTE ZW_GetControllerCapabilities (void)

Macro: ZW_GET_CONTROLLER_CAPABILITIES()

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

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

Defined in: ZW_controller_api.h

Return value:

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

Serial API:

HOST->ZW: REQ | 0x05

ZW->HOST: RES | 0x05 | RetVal

CONFIDENTIAL

5.4.6 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 EEPROM memory:

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 (Z-Wave Appl. Specific Part)							
Specific	Specific Device Class (Z-Wave Appl. Specific Part)							

Figure 8 Node Information frame without command classes format

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

Defined in: ZW_controller_api.h

Parameters:

bNodeID IN	Node ID	1..232
nodeInfo OUT	Node info buffer (see Figure 8)	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 8)

CONFIDENTIAL

5.4.7 ZW_SetDefault

void ZW_SetDefault(VOID_CALLBACKFUNC(completedFunc)(void))

Macro: ZW_SET_DEFAULT(func)

Remove all Nodes, routing information, assigned homeID/nodeID and RTC timers from the EEPROM memory. This function set the Controller back to the factory default state.

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: This function should be used 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

CONFIDENTIAL

5.4.8 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

CONFIDENTIAL

5.4.9 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

CONFIDENTIAL

5.4.10 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 node or Enhanced 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. Attempts to assign new return routes when all 5 destinations already are allocated will be ignored. Allocated destinations can only be cleared by the API call ZW_DeleteReturnRoute. 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 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	(see **ZW_SendData**)

Serial API:

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

ZW->HOST: RES | 0x46 | retVal

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

CONFIDENTIAL

5.4.11 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 node or Enhanced 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

CONFIDENTIAL

5.4.12 ZW_RemoveFailedNodeID

**BYTE ZW_RemoveFailedNodeID(BYTE NodeID,
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 removing process aborted because the node was node found.
	ZW_FAILED_NODE_REMOVE_PROCESS_BUSY	The removing process is busy.
	ZW_FAILED_NODE_REMOVE_FAIL	The removing process could not be started because of transmitter busy.

Parameters:

nodeID IN	The node ID (1..232) of the failed node to be deleted.
completedFunc IN	Remove process completed call back function

CONFIDENTIAL

Callback function Parameters:

txStatus IN Status of removal of failed node:

ZW_NODE_OK	The node is working properly (removed from the failed nodes list).
ZW_FAILED_NODE_REMOVED	The failed node was removed from the failed nodes list.
ZW_FAILED_NODE_NOT_REMOVED	The failed node was not removed because the removing process cannot be completed.

Serial API:

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

ZW->HOST: RES | 0x61 | retVal

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

5.4.13 ZW_isFailedNode

BYTE ZW_isFailedNode(BYTE nodeID)

Macro: ZW_IS_FAILED_NODE_ID(nodeID)

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

Defined in: ZW_controller_api.h

Return value:

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

Parameters:

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

Serial API:

HOST->ZW: REQ | 0x62 | nodeID

ZW->HOST: RES | 0x62 | retVal

CONFIDENTIAL

5.4.14 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	If the controller is a primary controller in the network.
	FALSE	If the controller is a secondary controller in the network.

Serial API (Not supported)

5.4.15 ZW_ReplaceFailedNode

**BYTE ZW_ReplaceFailedNode(BYTE NodeID,
VOID_CALLBACKFUNC(completedFunc)(BYTE txStatus))**

Macro: ZW_REPLACE_FAILED_NODE(node,func)

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

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

Defined in: ZW_controller_api.h

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

BYTE	ZW_FAILED_NODE_REMOVE_STARTED	The replacing process started and now the new node must emit its node information frame to start the assign process.

CONFIDENTIAL

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 replacing process aborted because the node was found, thereby not a failing node.
	ZW_FAILED_NODE_REMOVE_PROCESS_BUSY	The replacing process is busy.
	ZW_FAILED_NODE_REMOVE_FAIL	The replacing process could not be started because of transmitter busy.

CONFIDENTIAL

Parameters:

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

CONFIDENTIAL

5.4.16 ZW_GetNeighborCount

BYTE ZW_GetNeighborCount(BYTE nodeID)

Macro: ZW_GET_NEIGHBOR_COUNT (nodeID)

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

Defined in: ZW_controller_api.h

Return value:

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

Parameters:

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

Serial API

HOST->ZW: REQ | 0xBB | nodeID

ZW->HOST: RES | 0xBB | retVal

CONFIDENTIAL

5.4.17 ZW_AreNodesNeighbours

BYTE ZW_AreNodesNeighbours (BYTE bNodeA, BYTE bNodeB)

Macro: ZW_ARE_NODES_NEIGHBOURS (nodeA, nodeB)

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

Defined in: ZW_controller_api.h

Return value:

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

Parameters:

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

Serial API

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

ZW->HOST: RES | 0xBC | retVal

CONFIDENTIAL

5.4.18 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 **ZE_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

CONFIDENTIAL

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

5.4.19 ZW_GetRoutingInfo

void ZW_GetRoutingInfo(BYTE bNodeID,
BYTE_P pMask,
BYTE bRemoveBad,
BYTE bRemoveNonReps)

Macro: ZW_GET_ROUTING_INFO(bNodeID, pMask, bRemoveBad, bRemoveNonReps)

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_installer_api.h

Parameters:

bNodeID IN	Node ID (1…232) on node whom routing info is needed on.
pMask OUT	Pointer to buffer where routing info should be put. The buffer should be at least ZW_MAX_NODES/8 bytes
bRemoveBad IN	TRUE Remove bad link from routing info.
bRemoveNonReps IN	TRUE Remove non-repeaters from the routing info.

Serial API:

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

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

CONFIDENTIAL

5.4.20 ZW_GetSUCNodeID

BYTE ZW_GetSUCNodeID(void)

Macro: ZW_GET_SUC_NODE_ID()

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

5.4.21 ZW_SetSUCNodeID

BYTE ZW_SetSUCNodeID (BYTE nodeID,
BYTE SUCState,
BYTE bTxOption,
BYTE capabilities,
VOID_CALLBACKFUNC (completedFunc)(BYTE txStatus))

Macro: ZW_SET_SUC_NODE_ID(nodeID, SUCState, bTxOption, capabilities, func)

Used to configure a static/bridge controller to be a SUC/SIS node or not. The primary controller should use this function to set a static/bridge controller to be the SUC/SIS node, or it could be used to stop previously chosen static/bridge controller being a SUC/SIS node.

A controller can set itself to a SUC/SIS by calling **ZW_EnableSUC** and **ZW_SetSUCNodeID** with its own node ID. It's recommended to do this when the Z-Wave network only comprise of the primary controller to get the SUC/SIS role distributed when new nodes are included. It's possible to include a virgin primary controller with SUC/SIS capabilities configured into another Z-Wave network.

Note: If the function is used in a situation where the remotes are close to each other, then it is recommended that the function should be called with the bTxOption set to TRUE.

CONFIDENTIAL

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.

CONFIDENTIAL

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.

CONFIDENTIAL

5.4.22 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

CONFIDENTIAL

5.4.23 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

CONFIDENTIAL

5.4.24 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

CONFIDENTIAL

5.4.25 ZW_RequestNodeInfo

BOOL ZW_RequestNodeInfo (BYTE nodeID,
VOID (*completedFunc)(BYTE txStatus))

Macro: ZW_REQUEST_NODE_INFO(NODEID)

This function is used to request the node information frame from a controller based node in the network. The Node info is retrieved using the **ApplicationControllerUpdate** callback function with the status UPDATE_STATE_NODE_INFO_RECEIVED. This call is also available for routing slaves.

Defined in: ZW_controller_api.h

Return value:

BOOL	TRUE	If the request could be put in the transmit queue successfully.
	FALSE	If the request could not be put in the transmit queue. Request failed.

Parameters:

nodeID IN	The node ID (1...232) of the node to request the node information frame from.
completedFunc 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 **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.

CONFIDENTIAL

5.5 Z-Wave Static Controller API

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

5.5.1 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

CONFIDENTIAL

5.5.2 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

CONFIDENTIAL

	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[ ]

CONFIDENTIAL

5.6 Z-Wave Bridge Controller API

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

5.6.1 ZW_SendSlaveData

```
BYTE ZW_SendSlaveData(BYTE srcID,
                      BYTE destID,
                      BYTE *pData,
                      BYTE dataLength,
                      BYTE txOptions,
                      VOID_CALLBACKFUNC(completedFunc)(BYTE txStatus))
```

Macro: ZW_SLAVE_SEND_DATA(srcID, destID, data,length, options, func)

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

The transmit option TRANSMIT_OPTION_ACK request the destination node to return a transfer acknowledge that ensure proper transmission. The transmitting node will retry the transmission if no acknowledge received. If destID is NODE_BROADCAST then the transmit option is just ignored.

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 (response route). 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: ZW_SendSlaveData uses a transmit queue in the API so calling this function in a loop will overflow the transmit queue and fail. Instead the completedFunc callback must be used to determine when the next frame can be send.

Defined in: ZW_controller_bridge_api.h

Return value:

BYTE	FALSE	If transmit queue overflow or if srcID is not a Virtual Slave node.

CONFIDENTIAL

Parameters:

srcID	IN Source node ID indicating which Virtual Slave node transmits the frame.	
destID	IN Destination node ID (NODE_BROADCAST == all nodes)	
pData	IN Data buffer pointer	
dataLength	IN Data buffer length	
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	

Callback function Parameters:

txStatus	(see **ZW_SendData**)

Serial API:

HOST->ZW: REQ | 0xA3 | srcID | destID | dataLength | pData[ ] | txOptions | funcID

ZW->HOST: RES | 0xA3 | retVal

ZW->HOST: REQ | 0xA3 | funcID | txStatus

CONFIDENTIAL

5.6.2 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 (see **ZW_SendSlaveData**)
completedFunc IN	Transmit completed call back function

Callback function Parameters:

txStatus	(see **ZW_SendSlaveData**)

Serial API:

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

ZW->HOST: RES | 0xA2 | retVal

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

CONFIDENTIAL

5.6.3 ZW_SetSlaveLearnMode

BYTE ZW_SetSlaveLearnMode(BYTE node,
BYTE mode,
VOID_CALLBACKFUNC(learnSlaveFunc)(BYTE state, BYTE orgID, BYTE newID))

Macro: ZW_SET_SLAVE_LEARN_MODE (node, mode, func)

ZW_SetSlaveLearnMode enables the possibility for enabling or disabling “Slave Learn Mode”, which when enabled makes it possible for other controllers (primary or inclusion controllers) to add or remove a Virtual Slave Node to the Z-Wave network. Also is it possible for the bridge controller (only when primary or inclusion controller) to add or remove a Virtual Slave Node without involving other controllers. Available Slave Learn Modes are:

VIRTUAL_SLAVE_LEARN_MODE_DISABLE – Disables the Slave Learn Mode so that no Virtual Slave Node can be added or removed.

VIRTUAL_SLAVE_LEARN_MODE_ENABLE – Enables the possibility for other Primary/Inclusion controllers to add or remove a Virtual Slave Node. To add a new Virtual Slave node to the Z-Wave Network the provided “node” ID must be ZERO and to make it possible to remove a specific Virtual Slave Node the provided “node” ID must be the nodeID for this specific (locally present) Virtual Slave Node. When the Slave Learn Mode has been enabled the Virtual Slave node must identify itself to the external Primary/Inclusion Controller node by sending a “Node Information” frame (see **ZW_SendSlaveNodeInformation**) to make the add/remove operation commence.

VIRTUAL_SLAVE_LEARN_MODE_ADD - Add Virtual Slave Node to the Z-Wave network without involving any external controller. This Slave Learn Mode is only possible when bridge controller is either a Primary controller or an Inclusion controller.

VIRTUAL_SLAVE_LEARN_MODE_REMOVE - Remove a locally present Virtual Slave Node from the Z-Wave network without involving any external controller. This Slave Learn Mode is only possible when bridge controller is either a Primary controller or an Inclusion controller.

The **learnSlaveFunc** is called as the "Assign" process progresses. The returned “orgID” is the Virtual Slave node put into Slave Learn Mode, the “newID” is the new Node ID. If the Slave Learn Mode is VIRTUAL_SLAVE_LEARN_MODE_ENABLE and nothing is received from the assigning controller the callback function will not be called. It is then up to the main application code to switch of Slave Learn mode by setting the VIRTUAL_SLAVE_LEARN_MODE_DISABLE Slave Learn Mode. Once the assignment process has been started the Callback function may be called more than once.

NOTE: Slave Learn Mode should only be set to VIRTUAL_SLAVE_LEARN_MODE_ENABLE when necessary, and it should always be set to VIRTUAL_SLAVE_LEARN_MODE_DISABLE again as quickly as possible. It is recommended that Slave Learn Mode is never set to VIRTUAL_SLAVE_LEARN_MODE_ENABLE for more than 1 second.

CONFIDENTIAL

Defined in: ZW_controller_bridge_api.h

Return value:

BYTE	TRUE	If learnSlaveMode change was succesful.
	FALSE	If learnSlaveMode change could not be done.

Parameters:

node IN	Node ID (1...232) on node to set in Slave Learn Mode, ZERO if new node is to be learned.	
mode IN	Valid modes:	
	VIRTUAL_SLAVE_LEARN_MODE_DISABLE	Disable Slave Learn Mode
	VIRTUAL_SLAVE_LEARN_MODE_ENABLE	Enable Slave Learn Mode
	VIRTUAL_SLAVE_LEARN_MODE_ADD	ADD: Create locally a Virtual Slave Node and add it to the Z-Wave network (only possible if Primary/Inclusion Controller).
	VIRTUAL_SLAVE_LEARN_MODE_REMOVE	Remove locally present Virtual Slave Node from the Z-Wave network (only possible if Primary/Inclusion Controller).
learnFunc IN	Slave Learn mode complete call back function	

CONFIDENTIAL

Callback function Parameters:

bStatus	Status of the assign process.	
	ASSIGN_COMPLETE	Is returned by the callback function when in the VIRTUAL_SLAVE_LEARN_MODE_ENABLE Slave Learn Mode and assignment is done. Now the Application can continue normal operation.
	ASSIGN_NODEID_DONE	Node ID have been assigned. The “orgID” contains the node ID on the Virtual Slave Node who was put into Slave Learn Mode. The “newID” contains the new node ID for “orgID”. If “newID” is ZERO then the “orgID” Virtual Slave node has been deleted and the assign operation is completed. When this status is received the Slave Learn Mode is complete for all Slave Learn Modes except the VIRTUAL_SLAVE_LEARN_MODE_ENABLE mode.
	ASSIGN_RANGE_INFO_UPDATE	Node is doing Neighbour discovery Application should not attempt to send any frames during this time, this is only applicable when in VIRTUAL_SLAVE_LEARN_MODE_ENABLE.
orgID	The original node ID that was put into Slave Learn Mode.	
newID	The new Node ID. Zero if “OrgID” was deleted from the Z-Wave network.	

Serial API:

HOST->ZW: REQ | 0xA4 | node | mode | funcID

ZW->HOST: RES | 0xA4 | retVal

ZW->HOST: REQ | 0xA4 | funcID | bStatus | OrgID | newID

CONFIDENTIAL

5.6.4 ZW_IsVirtualNode

BYTE ZW_IsVirtualNode(BYTE nodeID)

Macro: ZW_IS_VIRTUAL_NODE (nodeid)

Checks if "nodeID" is a Virtual Slave node.

Defined in: ZW_controller_bridge_api.h

Return value:

BYTE	TRUE	If "nodeID" is a Virtual Slave node.
	FALSE	If "nodeID" is not a Virtual Slave node.

Parameters:

nodeID IN	Node ID (1...232) on node to check if it is a Virtual Slave node.

Serial API:

HOST->ZW: REQ | 0xA6 | nodeID

ZW->HOST: RES | 0xA6 | retVal

CONFIDENTIAL

5.6.5 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]

CONFIDENTIAL

5.7 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.

5.7.1 zwTransmitCount

BYTE zwTransmitCount

Macro: ZW_TX_COUNTER

ZW_TX_COUNTER is a variable that returns the number of transmits that the protocol has done since last reset of the variable. If the number returned is 255 then the number of transmits ≥ 255. The variable should be reset by the application, when it is to be restarted.

Defined in: ZW_controller_installer_api.h

Serial API:

To read the transmit counter:

HOST->ZW: REQ | 0x81| (FUNC_ID_GET_TX_COUNTER)

ZW->HOST: RES | 0x81 | ZW_TX_COUNTER (1 byte)

To reset the transmit counter:

HOST->ZW: REQ | 0x82| (FUNC_ID_RESET_TX_COUNTER)

CONFIDENTIAL

5.7.2 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

CONFIDENTIAL

5.7.3 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

CONFIDENTIAL

5.8 Z-Wave Slave API

The Slave application interface is an extension to the Basis application interface enabling inclusion/exclusion of Slave, Rounting Slave, and Enhanced Slave nodes.

5.8.1 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. This function is used 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: Learn mode should only be enabled when necessary, and it should always be disabled again as quickly as possible. It is recommended that learn mode is never enabled in more than 1 second.

Defined in: ZW_slave_api.h

Parameters:

mode IN	TRUE: Enable; FALSE: Disable
learnFunc IN	Node ID learn mode completed call back function

CONFIDENTIAL

Callback function Parameters:

bStatus	Status of the assign process	
	ASSIGN_COMPLETE	Assignment is done and Application can continue normal operation.
	ASSIGN_NODEID_DONE	Node ID has been assigned. More information may follow.
	ASSIGN_RANGE_INFO_UPDATE	Node is doing Neighbor discovery Application should not attempt to send any frames during this time.
nodeID	The new (learned) Node ID (1...232)	

Serial API:

HOST->ZW: REQ | 0x50 | mode | funcID

ZW->HOST: REQ | 0x50 | funcID | nodeID

5.8.2 ZW_SetDefault

void ZW_SetDefault(void)

Macros: ZW_SET_DEFAULT

The API call **ZW_SetDefault** is used to reset all types of slaves to its default state by clearing the home ID and node ID to zero. In addition is the routing information cleared in routing and enhanced slaves. Finally is the RTC timers also cleared in enhanced slaves.

Defined in: ZW_slave_api.h

Serial API:

HOST->ZW: REQ | 0x42 | funcID

ZW->HOST: REQ | 0x42 | funcID

CONFIDENTIAL

5.8.3 ZW_Support9600Only (ZW0201/ZW0301 only)

BOOL ZW_Support9600Only(BOOL bValue)

Macros: ZW_SUPPORT9600_ONLY(value)

The API call **ZW_Support9600Only** can select that non-listening ZW0201/ZW0301 slaves only want to support 9.6kbit/s baudrate.

Important: This call should **only** be placed in ApplicationInitSW

Defined in: ZW_slave_api.h

Return value:

BOOL	TRUE	The baudrate change was succesfull.
	FALSE	Baudrate could not be changed because the node was listening.

Parameters:

bValue	Select if this node should only support 9.6kbit/s	
	TRUE	This node will now act as a 9.6kbit/s
	FALSE	This node will respond on all supported baudrates

Serial API:

Not implemented

5.9 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.

5.9.1 ZW_RequestNewRouteDestinations

BYTE ZW_RequestNewRouteDestinations(BYTE *pDestList,
BYTE bDestListLen ,
VOID_CALLBACKFUNC (completedFunc)(BYTE txStatus))

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 list of new destinations to request
bDestListLen	Length of Destinationlist buffer. Must not be more than ZW_MAX_RETURN_ROUTE_DESTINATIONS
completedFunc IN	Transmit completed call back function

Callback function parameters:

ZW_SUC_UPDATE_DONE	The update process is ended successfully
ZW_SUC_UPDATE_ABORT	The update process aborted because of error
ZW_SUC_UPDATE_WAIT	The SUC node is busy
ZW_SUC_UPDATE_DISABLED	The SUC functionality is disabled

CONFIDENTIAL

Serial API:

Not supported

5.9.2 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:

Not supported

5.9.3 ZW_RequestNodeInfo

BOOL ZW_RequestNodeInfo (BYTE nodeID,
VOID (*completedFunc)(BYTE txStatus))

Macro: ZW_REQUEST_NODE_INFO(NODEID)

This function is used to request the node information frame from a slave based node in the network. The Node info is retrieved using the **ApplicationSlaveUpdate** callback function with the status UPDATE_STATE_NODE_INFO_RECEIVED. This call is also available for controllers.

Defined in: ZW_slave_routing_api.h

Return value:

BOOL	TRUE	If the request could be put in the transmit queue successfully.
	FALSE	If the request could not be put in the transmit queue. Request failed.

Parameters:

nodeID IN	The node ID (1...232) of the node to request the node information frame from.
completedFunc 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.

CONFIDENTIAL

5.10 Z-Wave Zensor Net Routing Slave API

The Zensor Net routing slave API contains all the functions in the Slave, Routing Slave and Enhanced Slave API. The following functions are only available in the Zensor Net Routing Slave library.

5.10.1 ZW_ZensorNetBind

BYTE ZW_ZensorNetBind(BYTE bMode,
VOID (*completedFunc)(BYTE bStatus, BYTE bZensorID))

Start or stop Zensor Net bind mode according to the specified mode.

Defined in: ZW_sensor_api.h

Return value:

	TRUE	The Zensor has entered the requested mode
	FALSE	The Zensor could not enter bind mode because the requested mode doesn't match current Slaev/Master state.

Parameters:

bMode IN	BIND_MODE_MASTER	The node will become master in the Zensor Net and will bind other nodet to the net on request.
	BIND_MODE_BIND_SLAVE	The noe will become slave in the Zensor net and will bind to a mMaster if one in avalible..
	BIND_MODE_BIND_ANY	The node will become Master or Slave depending on how the other part in the bind process has set its bind mode.
	BIND_MODE_UNBIND_SLAVE	If the node is a Slave node it will unbind from the network when a Master is avalible if it is a slave node. If the node is a Master it will return FALSE in the call.
	BIND_MODE_OFF	Turn off bind mode.
CompletedFunc IN	Callback function called when the bind process completes.	

Callback function Parameters:

bStatus	BIND_COMPLETE_SLAVE_OK	Bind was completed succesfully and the node is now a slave in the Zensor Net

CONFIDENTIAL

	BIND_COMPLETE_MASTER_OK	Bind was completed succesfully and the node is now the master in the Zensor Net
	BIND_COMPLETE_FAILED	Bind failed
bZensorID	The new ZensorID of thsis node	

Serial API:

Not supported

5.10.2 ZW_ZensorSendDataFlood

```
BYTE ZW_ZensorSendDataFlood(BYTE bNodeID,
                           BYTE *pData
                           BYTE bDataLength
                           BYTE txOptions
                           VOID  (*completedFunc)(BYTE txStatus))
```

Send a flooded frame out in the Zensor Net. The frame will be forwarded by all other Zensor Net Routing Slaves in the network.

Defined in: ZW_sensor_api.h

Return value:

TRUE	The frame was queued for transmission
FALSE	Transmit queue full, frame will not be send.

Parameters:

bNodeID IN	Node ID the frame should be send to. Normally this should be set to 0xFF
pData IN	Pointer to data that should be send in the frame.
bDataLength IN	Number of bytes in the data buffer
txOptions IN	N/A
CompletedFunc IN	Callback function called on transmit complete.
CompletedFunc IN	Callback function called when the bind process completes.

CONFIDENTIAL

Callback function Parameters:

txStatus	TRANSMIT_COMPLETE_OK	The frame was send succesfully.
	TRANSMIT_COMPLETE_FAIL	The frame could not be send because of lack of resources or to many collisions

Serial API:

Not supported

5.10.3 ZW_ZensorGetID

BYTE ZW_ZensorGetID()

Get the current ZensorID of this node..

Defined in: ZW_sensor_api.h

Return value:

The ZensorID of this node

Serial API:

Not supported

5.10.4 ZW_ZensorSetDefault

void ZW_ZensorSetDefault()

Set the Zensor Net part of this node to default values.

NOTE: The Z-Wave nodeID, HomeID and any assigned return routes will not be reset by calling this function.

Defined in: ZW_sensor_api.h

Serial API:

Not supported

CONFIDENTIAL

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.

CONFIDENTIAL

Commands:

H[elp]		Display the help text.
D[ump]	<addr> [<length>]	Dump idata (0-7F) or SFR memory (80-FF).
DX	<addr> [<length>]	Dump xdata (SRAM) memory.
DI	<addr> [<length>]	Dump “internal EEPROM” flash (slave only).
DE	<addr> [<length>]	Dump external EEPROM (controllers/slave_enhanced only).
DF	<addr> [<length>]	Dump FLASH memory.
E[dit]	<addr>	Edit idata (0-7F) or SFR memory (80-FF).
EX	<addr>	Edit xdata memory.
EI	<addr>	Edit “internal EEPROM” flash (slave only).
EE	<addr>	Edit external EEPROM (controllers/slave_enhanced only).
W[atch]	<addr>	Watch idata (0-7F) or SFR memory (80-FF).
WX	<addr>	Watch xdata memory.
WI	<addr>	Watch “internal EEPROM” flash (slave only).
WE	<addr>	Watch external EEPROM memory (controllers/slave_enhanced only).
WF	<addr>	Watch FLASH memory.

The Watch pointer gives the following log (when memory change):

idata SRAM memory	Rnn	
xdata SRAM memory	Xnn	
Internal EEPROM flash	Inn	(slave only)
External EEPROM	Enn	(controllers/slave_enhanced only)

Examples:

```
>dx 0                ; Edit offset 0x0000 and 0x0001 of xdata SRAM
0000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
>ex 0                ; Edit offset 0x0000 and 0x0001 of xdata SRAM
0000 00-1 00-2
>dx 0                ; Dump offset 0x0000 to 0x000f of xdata SRAM
0000  01 02 00 00 00 00 00 00  00 00 00 00 00 00 00 00
>wx 1X02             ; Watch offset 0x0001 of xdata SRAM
>ex 1
0001 02-1X01
>
```

CONFIDENTIAL

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)

CONFIDENTIAL

5.12 RF Settings in App_RFSetup.a51 file

The Z-Wave libraries are capable of transmitting/receiving on either 868.42MHz (EU) or 908.42MHz (US) or 921.42MHz (ANZ) or 919.82MHz (HK). The default frequency is set to US.

5.12.1 ZW0102 RF parameters

To allow for the selection of frequency, match capacity array values (for RF RX and TX mode) and transmit power levels (normal and low power) every application must have the App_RFSetup.a51 module linked in. The App_RFSetup.a51 module contains the definition of a const block placed in flash:

Table 9. App_RFSetup.a51 module definitions for ZW0102

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.
1	FLASH_APPL_FREQ_OFFS	0x01	0x00-0x01	0x00 = EU. 0x01 = US. 0xXX = use default.
2	FLASH_APPL_RX_MATCH_OFFS	0xFF		If 0xff the default lib value is used: (0x20).
3	FLASH_APPL_TX_MATCH_OFFS	0xFF		If 0xff the default lib value is used: (0x00).

4	FLASH_APPL_NORM_POWER_OFFS	0xFF		If 0xff the default lib value is used:
				US = 0x60
				EU = 0xF0

5	FLASH_APPL_LOW_POWER_OFFS	0xFF		If 0xff the default lib value is used:(0x10).

An application programmer can select RF frequency (US - 908.42MHz or EU – 868.42MHz), and the values for normal and low power transmissions by changing the const block defined in the App_RFSetup.a51 module. The RF frequency to use can be set by defining either US or UE in the application makefile. The TXmatch and RXmatch values don't need to be adjusted when using a standard Z-Wave module. In case the layout is modified then refer to [9] regarding how to change the TXmatch and Rxmatch. The TXnormal Power needs to be adjusted when making the FCC compliance test. According to the FCC part 15, the output radiated power shall not exceed 94dBuV/m. This radiated power is the result of the module output power and your product antenna gain. As the antenna gain is different from product to product, the module output power needs to be adjusted to comply with the FCC regulations.

The match and power values can be adjusted directly on the module by the Z-Wave Programmer [14].

CONFIDENTIAL

5.12.2 ZW0201/ZW0301 RF parameters

To allow for the selection of frequency, and transmit power levels (normal and low power) every application must have the App_RFSetup.a51 module linked in. The App_RFSetup.a51 module contains the definition of a const block placed in flash:

Table 10. 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.
1	FLASH_APPL_FREQ_OFFS	0x01	0x00-0x03	0x00 = EU 0x01 = US 0x02 = ANZ 0x03 = HK 0xXX = use default.

2	FLASH_APPL_NORM_POWER_OFFS	0xFF		If 0xFF the default lib value is used:
				US = 0x1B
				EU = 0x2A
				ANZ = 0x2A
				HK = 0x2A

4	FLASH_APPL_LOW_POWER_OFFS	0xFF		If 0xFF the default lib value is used:0x14
5	FLASH_APPL_PLL_STEPUP_OFFS	0xFF	0x00	Only supported on ZW0301

An application programmer can select RF frequency (US - 908.42MHz or EU – 868.42MHz or ANZ - 921.42MHz or HK - 919.82MHz), and the values for normal and low power transmissions by changing the const block defined in the App_RFSetup.a51 module. The RF frequency to use can be set by defining either US or UE or ANZ or HK in the application makefile. The TXnormal Power needs to be adjusted when making the FCC compliance test. According to the FCC part 15, the output radiated power shall not exceed 94dBuV/m. This radiated power is the result of the module output power and your product antenna gain. As the antenna gain is different from product to product, the module output power needs to be adjusted to comply with the FCC regulations.

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

The match and power values can be adjusted directly on the module by the Z-Wave Programmer [14].

CONFIDENTIAL

6 HARDWARE SUPPORT DRIVERS

While the previous sections describe the generic Z-Wave modules that handle the wireless communication between the Z-Wave nodes, this section describe interfaces to common hardware components.

6.1 Hardware Pin Definitions

The hardware specific directories in the \product directory contain a ZW_pindefs.h file that defines macros for access to the I/O pins on the module.

Macros for accessing the I/O pins:

PIN_IN(pin, pullup)

Set I/O pin as input.

Parameters:

pin IN	Z-Wave pin name
pullup IN	If not zero activate the internal pull-up resistor

Example:

PIN_IN(IO1,0) ; define pin IO1 as an input pin and disables the internal pull-up resistor.

NOTE: The pull-up feature is not available in the ZW010x ASIC

PIN_OUT(pin)

Set I/O pin as output.

Parameters:

pin IN	Z-Wave pin name

Example:

PIN_OUT(IO2) ; define pin IO2 as an output pin.

CONFIDENTIAL

PIN_GET(pin)

Read pin value:

Parameters:

pin IN	Z-Wave pin name

Example:

```
if (PIN_GET(IO1))
  /* action when pin IO1 is 1 */
```

PIN_ON(pin)

Set output pin to 1 (on).

Parameters:

pin IN	Z-Wave pin name

Example:

```
PIN_ON(IO2);  /* set pin IO2 to 1 */
```

PIN_OFF(pin)

Set output pin to 0 (off).

Parameters:

pin IN	Z-Wave pin name

Example:

```
PIN_OFF(IO2);/* set pin IO2 to 0 */
```

CONFIDENTIAL

PIN_TOGGLE(pin)

Toggle output pin.

Parameters:

pin IN	Z-Wave pin name

Example:

```
PIN_TOGGLE(IO2); /* toggle pin IO2 */
```

CONFIDENTIAL

7 APPLICATION SAMPLE CODE

The Z-Wave Developer's Kit includes several sample applications: a serial controller application, a LED dimmer application, a binary sensor and a battery operated binary sensor application for the Z-Wave module. The sample application realizes a light control system to help the developer to understand how the various components can interact. In addition the Z-Wave Developer's Kit also comprises of a number of PC centric sample applications for displaying advanced functionalities of the Z-Wave protocol:

- How a Z-Wave Module can be controlled from a PC.
- Installation including display of network topology.
- Bridging to and from other networks.

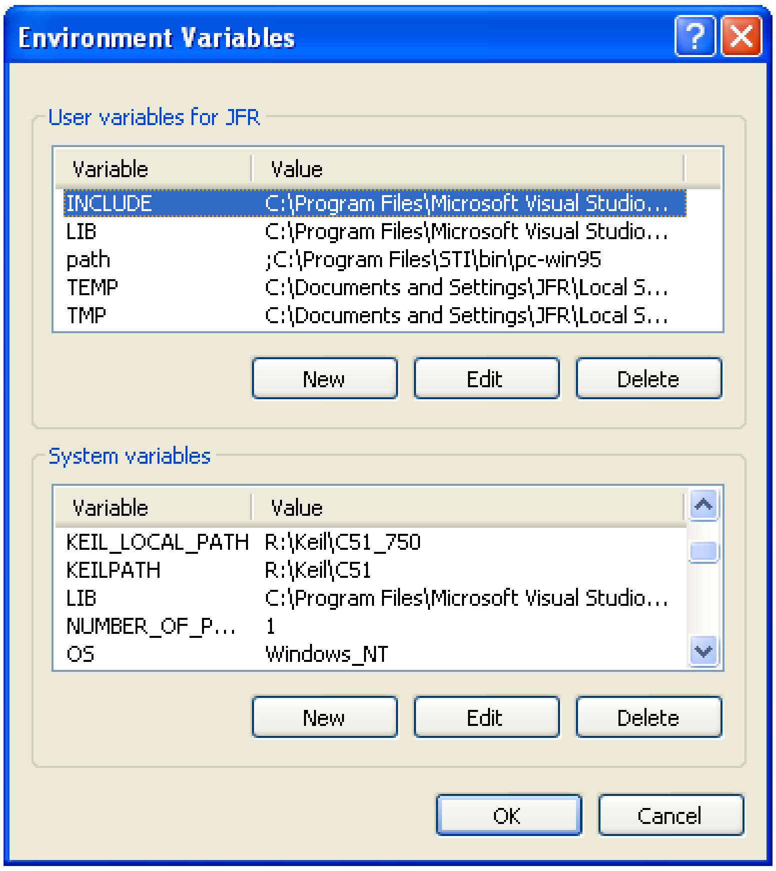
7.1 Building ZW0x0x Sample Code

All the sample applications for the ZW0102/ZW0201/ZW0301 contains source code and make files that allows the developer to modify and compile the applications without a lot of make file configuration. All sample applications are built by calling the MK.BAT file that is located in the sample application directory.

CONFIDENTIAL

7.1.1 MK.BAT

This batch-file calls the make tool which then, via the makefiles, builds the sample application to the different RF frequencies (ANZ/EU/HK/US) used when transmitting/receiving and Z-Wave module targets. Three environment variables must be defined before it is possible to build the sample applications. The procedure is on a PC using Windows XP as follows:



User variables for JFR

Variable	Value
INCLUDE	C:\Program Files\Microsoft Visual Studio...
LIB	C:\Program Files\Microsoft Visual Studio...
path	;C:\Program Files\STI\bin\pc-win95
TEMP	C:\Documents and Settings\JFR\Local S...
TMP	C:\Documents and Settings\JFR\Local S...

System variables

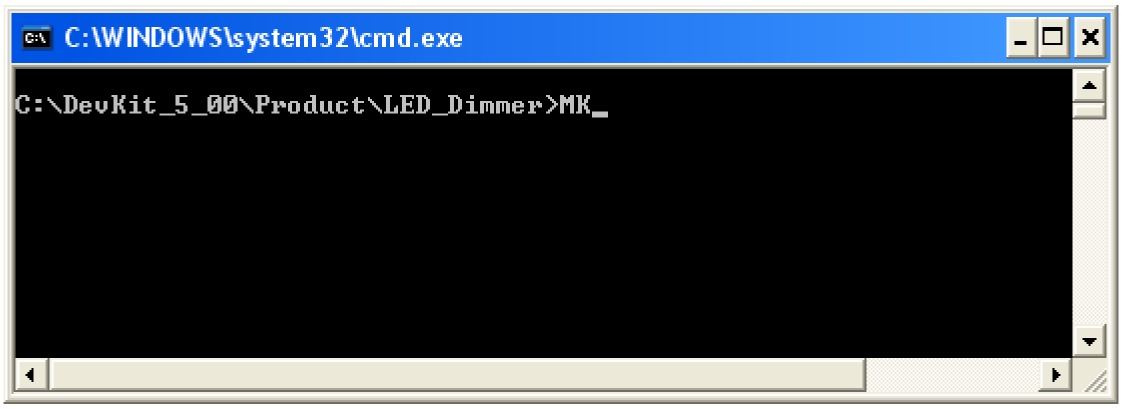
Variable	Value
KEIL_LOCAL_PATH	R:\Keil\C51_750
KEILPATH	R:\Keil\C51
LIB	C:\Program Files\Microsoft Visual Studio...
NUMBER_OF_P...	1
OS	Windows_NT

Figure 9 Configuring environment variables

1. Select **Start**, **Control Panel** and **System**
2. Select **Advanced** tab and activate the **Environment Variables** button
3. Under **System variables** activate the **New** button
4. In the **Variable name** textbox enter **KEILPATH** (use capital letters because Windows XP is case sensitive)
5. In the **Variable value** textbox enter **C:\KEIL\C51** and activate the **OK** button
6. Under **System variables** activate the **New** button
7. In the **Variable name** textbox enter **KEIL_LOCAL_PATH** (use capital letters because Windows XP is case sensitive)
8. In the **Variable value** textbox enter **C:\KEIL\C51_750** and activate the **OK** button
9. Under **System variables** activate the **New** button
10. In the **Variable name** textbox enter **TOOLSDIR** (use capital letters because Windows XP is case sensitive)
11. In the **Variable value** textbox enter **C:\Devkit_5_00\TOOLS** and activate the **OK** button

Afterwards open a command prompt (DOS box) in the relevant sample application directory to build the application.

CONFIDENTIAL



```
C:\DevKit_5_00\Product\LED_Dimmer>MK_
```

Figure 10 Building sample applications

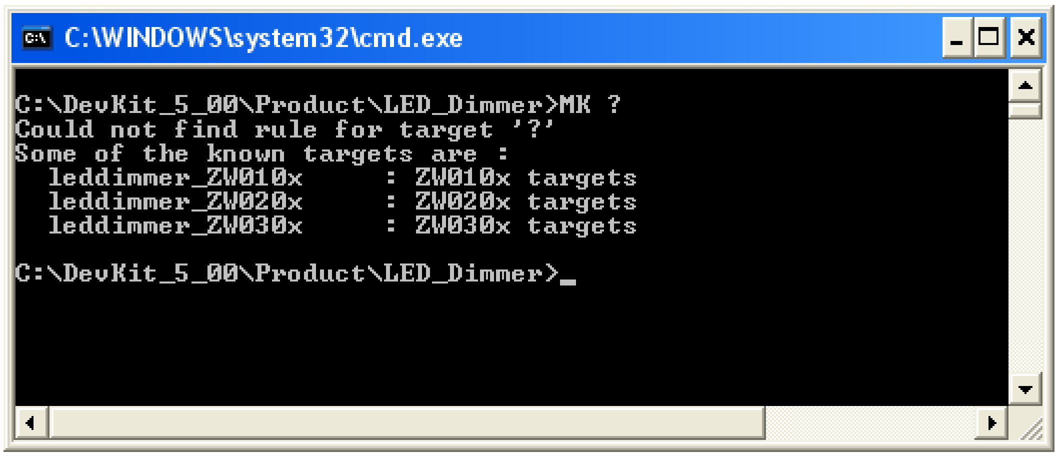
Remember to use upper case in **KEILPATH, KEIL_LOCAL_PATH** and **TOOLSDIR** when using Windows XP, because this operating system is case sensitive. If the environment variables are not defined then MK.BAT will prompt the user to define them.

Opening a command prompt to a particular directory from Explorer is enabled in the following way:

1. Start Regedit
2. Go to HKEY_CLASSES_ROOT \ Directory \ shell
3. Create a new key called *Command*
4. Give it the value of the name you want to appear in the Explorer. Something like *Open DOS Box*
5. Under this create a new key called *command*
6. Give it a value of *cmd.exe /k "cd %L"*
7. Now when you are in the Explorer, right click on a folder, select *Open DOS Box*, and a command prompt will open to the selected directory.

CONFIDENTIAL

The batch file MK.BAT builds all versions with respect to targets and RF frequencies. The wanted target can also be entered as a parameter on the command line. The figure below displays the possible targets for a given product.



```
C:\DevKit_5_00\Product\LED_Dimmer>MK ?
Could not find rule for target '?'
Some of the known targets are :
  leddimmer_ZW010x      : ZW010x targets
  leddimmer_ZW020x      : ZW020x targets
  leddimmer_ZW030x      : ZW030x targets

C:\DevKit_5_00\Product\LED_Dimmer>_
```

Figure 11 Possible sample application targets

Remember to enter the targets as shown when using Windows XP, because this operating system is case sensitive.

When MK.BAT is executed the following directory structure is created within the source code directory

For ZW0102 targets:

```
- <application>
    - build
        - <application>_ZW010x_EU
            - list                          - contains list files
            - Rels                          - contains object files and map file
            <application>_ZW010x_EU.hex     - flash Intel hex file for EU ZW0102 based module
        -<application>_ZW010x_US
            - list                          - contains list files
            - Rels                          - contains object files and map file
            <application>_ZW010x_US.hex     - flash Intel hex file for US ZW0102 based module
```

Some flash Intel hex files can depend on the targeted ZW0102 module.

CONFIDENTIAL

For ZW0201 targets:

```
- <application>
    - build
        - <application>_ZW020x_EU
            - list                          - contains list files
            - Rels                          - contains object files and map file
            <application>_ZW020x_EU.hex     - flash Intel hex file for EU ZW0201 based module
        -<application>_ZW020x_US
            - list                          - contains list files
            - Rels                          - contains object files and map file
            <application>_ZW020x_US.hex     - flash Intel hex file for US ZW0201 based module
        -<application>_ZW020x_ANZ
            - list                          - contains list files
            - Rels                          - contains object files and map file
            <application>_ZW020x_ANZ.hex    - flash Intel hex file for ANZ ZW0201 based module
        -<application>_ZW020x_HK
            - list                          - contains list files
            - Rels                          - contains object files and map file
            <application>_ZW020x_HK.hex     - flash Intel hex file for HK ZW0201 based module
```

For ZW0301 targets:

```
- <application>
    - build
        - <application>_ZW030x_EU
            - list                          - contains list files
            - Rels                          - contains object files and map file
            <application>_ZW030x_EU.hex     - flash Intel hex file for EU ZW0301 based module
        -<application>_ZW030x_US
            - list                          - contains list files
            - Rels                          - contains object files and map file
            <application>_ZW030x_US.hex     - flash Intel hex file for US ZW0301 based module
        -<application>_ZW030x_ANZ
            - list                          - contains list files
            - Rels                          - contains object files and map file
            <application>_ZW030x_ANZ.hex    - flash Intel hex file for ANZ ZW0301 based module
        -<application>_ZW030x_HK
            - list                          - contains list files
            - Rels                          - contains object files and map file
            <application>_ZW030x_HK.hex     - flash Intel hex file for HK ZW0301 based module
```

7.1.2 Makefiles

Makefile

This file is part of the make job. It creates the directory structure and defines the build targets and calls the other make files in the build depending on the target.

NOTE: The Makefile might contain test or debug targets that are not build in the default build.

Product\Common directory

CONFIDENTIAL

The common directory contains a set of standard make files that are used to build all the sample application. The make files define the compiler and linker options used to build all Z-Wave applications and the correct defines for all the different library types. The following compiler control line defines are used by the common makefiles:

ZW_CONTROLLER	Basic controller
ZW_CONTROLLER_32	Adding 32KB flash and controller functionality
ZW_CONTROLLER_STATIC	Adding static controller functionality
ZW_INSTALLER	Adding installer controller functionality
ZW_CONTROLLER_BRIDGE	Adding bridge controller functionality
ZW_SLAVE	Basic slave
ZW_SLAVE_32	Adding 32KB flash and enhanced slave functionality
ZW_SLAVE_ROUTING	Adding routing slave functionality
ZW010x	Basic 100 Series ASIC functionality
ZW0102	ZW0102 ASIC specific functionality
ZW020x	Basic 200 Series ASIC functionality
ZW0201	ZW0201 ASIC specific functionality
ZW030x	Basic 300 Series ASIC functionality
ZW0301	ZW0301 ASIC specific functionality
NEW_NODEINFO	Supports all node information frame formats

CONFIDENTIAL

7.2 Binary Sensor Sample Code

The Developer's Kit contains sample code for a Binary sensor. This device is in effect a binary sensor where the sensor input is the pin also used as a button input on the device module. The Bin_Sensor will on every button release transmit a basic set frame to any associated devices. If the button is held for a little while instead a nodeInfo frame will be transmitted. A static controller such as the one shown in 7.10 can control, configure and assign routes to the Bin_Sensor.

The Bin_Sensor is a binary sensor that supports the association command class described in the device class specification (see ref [1]). This device complies with the specific device class named routing binary sensor device class (4.1). The binary sensor advertises via the node information frame support for the following command classes:

- Binary Sensor command class
- Association command class
- Version command class
- Manufacturer Specific command class

CONFIDENTIAL

The Bin_Sensor advertises the binary sensor command class, the configuration command class and the association command class in the node information message.

The Bin_Sensor is a slave device based on the enhanced slave API. During initialization, the Bin_Sensor will initialize the mounted button, the 4 LED's and a timer function that handles the button input and sensor input (in this example the same as the button input). It will also get stored data from the non-volatile memory. After the initialization the Z-Wave basis software will continually call the **ApplicationPoll** function, which contains the Bin_Sensor main function. The **ApplicationPoll** function checks if the button or the sensor input has changed state and then acts accordingly to the current state the Bin_Sensor is in. The other main function is the **ApplicationCommandHandler** function that is called every time a command has been received, destined for the Bin_Sensor. This function checks the command and acts according to the command. When transmitting the Bin_Sensor will, if routes have been assigned use these.

The Bin_Sensor implements Lost functionality and network topology maintenance by using a series of methods. If the device is unsuccessful in sending a message a predefined count it will enter lost state, and attempt to find a SUC in the network, and if successful ask the SUC for routes to the failing devices. At regular intervals the Bin_Sensor will transmit a Static Route Request, which asks the SUC for any updates done to the network.

7.2.1 Bin_Sensor Files

The Product\Bin_Sensor directory contains sample source code for a slave application on a Z-Wave module.

Makefile

This file is part of the make job. It creates the directory structure and defines targets. The following compiler control line defines are used in the makefiles:

US	: Build US frequency target.
EU	: Build EU frequency target.
ANZ	: Build ANZ frequency target.
HK	: Build HK frequency target.

CONFIDENTIAL

Makefile.binsensor_ZW010x_EU
Makefile.binsensor_ZW010x_US

This file is part of the make job. Is the main makefile for making the EU/US version of the Bin_Sensor with description of which modules is included in the application and which application specific defines is used in the compilation of the modules.

Makefile.binsensor_ZW020x_EU
Makefile.binsensor_ZW020x_US
Makefile.binsensor_ZW020x_ANZ
Makefile.binsensor_ZW020x_HK

This file is part of the make job. Is the main makefile for making the ANZ/EU/HK/US version of the Bin_Sensor to the ZW0201 Z-Wave Slave/Controller Unit, with description of which modules is included in the application and which application specific defines is used in the compilation of the modules.

Makefile.binsensor_ZW030x_EU
Makefile.binsensor_ZW030x_US
Makefile.binsensor_ZW030x_ANZ
Makefile.binsensor_ZW030x_HK

This file is part of the make job. Is the main makefile for making the ANZ/EU/HK/US version of the Bin_Sensor to the ZW0301 Z-Wave Slave/Controller Unit, with description of which modules is included in the application and which application specific defines is used in the compilation of the modules.

Bin_Sensor.c

Contains the source for the Bin_Sensor application.

Bin_Sensor.h

Contains definitions for the Bin_Sensor application.

Please refer to section 3.3.13 for the utility library functions which are used by the application.

CONFIDENTIAL

Macros for accessing the LED's:

LED_ON(led)

Turn LED on.

Parameter:
led - LED number

Example:
```
    PIN_OUT(LED1); /* define LED1 as an output pin */
       LED_ON(1);       /* turn LED 1 on */
```

LED_OFF(led)

Turn LED off.

Parameter:
led - LED number

Example:
```
       LED_OFF(1);       /* turn LED 1 off */
```

LED_TOGGLE(led)

Toggle the LED OFF if the LED was ON and ON if the LED was OFF.

Parameter:
led - LED number

Example:
```
  LED_TOGGLE(1);       /* toggle LED 1 */
```

CONFIDENTIAL

7.3 Binary Sensor Battery Sample Code

The Developer's Kit contains sample code for a battery powered Binary sensor. This device is in effect a binary sensor where the sensor input is the pin also used as a button input on the device module. When the Binary Sensor is inactive the ASIC will be powered down. The Binary sensor will power up when the button is pressed or the RTC / WUT[6] is fired. Upon wake up, be it button press or RTC/WUT a Wake Up Notification Frame is sent either as broadcast or as singlecast to the device that configured the wake up settings. If the devie has any associations it will transmit a basic set to the associated devices. If the button is held for a longer time a Node Information Frame is transmitted. A static controller such as the one shown in 7.10 can control, configure and assign routes to the Bin_Sensor_Battery.

The Bin_Sensor_Battery is a binary sensor that supports the association command class and the wake up command class described in the device class specification (see ref [1]). This device complies with the specific device class named routing binary sensor device class (4.1). The battery-operated binary sensor advertises via the node information frame support for the following command classes:

- Binary Sensor command class
- Wake Up command class
- Association command class
- Version command class
- Manufacturer Specific command class

The Bin_Sensor_Battery is a slave device based on the enhanced slave API. During initialization, the Bin_Sensor_Battery will initialize the mounted button, the 4 LED's and a timer function that handles the button input and sensor input (in this example the same as the button input). It will also get stored data from the non-volatile memory. After the initialization will go in power down mode and it will wake up again either when the button is pressed or when the RTC timer / WUT is fired. While the Bin_sensor_Battery is wake the Z-Wave basis software will continually call the **ApplicationPoll** function, which contains the Bin_Sensor_Battery main function. The **ApplicationPoll** function checks if the button or the sensor input has changed state and then acts accordingly to the current state the Bin_Sensor_Battery is in. The other main function is the **ApplicationCommandHandler** function that is called every time a command has been received, destined for the Bin_Sensor_Battery. This function checks the command and acts according to the command. When transmitting the Bin_Sensor_Battery will, if routes have been assigned use these. If the Bin_sensor_Battery was waked by the sensor input or button activity, then it will power down again it is done executing any event caused by the sensor input or the button. If the binary sensor is woken up by RTC timer / WUT and the wake up time interval is expired then it will send wake notification frame and wait for 5 second before powering down again.

The Bin_Sensor_Battery implements Lost functionality and network topology maintenance by using a series of methods. If the device is unsuccessful in sending a message a predefined count it will enter lost state, and attempt to find a SUC in the network, and if successful ask the SUC for routes to the failing devices. At regular intervals the Bin_Sensor_Battery will transmit a Static Route Request, which asks the SUC for any updates done to the network.

Note that the wake up notification frame will only be sent when the Bin_sensor_Battery has been assigned a node ID.

[6] RTC is used in ZW0102 and WUT is used in ZW0201.

CONFIDENTIAL

On the ZW0201 some of the uninitialized RAM bytes are used to keep track of the WUT timer. See also section 4.1

The Bin_Sensor and Bin_Sensor_Battery share the same code base. They are distinquished between by defining BATTERY when compiling which will also enable use of the utility function file battery.c/h.

7.3.1 Bin_Sensor_Battery Files

The Product\Bin_Sensor_Battery directory contains sample source code for a slave application on a Z-Wave module.

Makefile

This file is part of the make job. It creates the directory structure and defines targets. The following compiler control line defines are used in the makefiles:

US	: Build US frequency target.
EU	: Build EU frequency target.
ANZ	: Build ANZ frequency target.
HK	: Build HK frequency target.

Makefile.Binsensor_Battery_ZW010x_EU
Makefile.Binsensor_Battery_ZW010x_US

This file is part of the make job. Is the main makefile for making the EU/US version of the Bin_Sensor_Battery with description of which modules is included in the application and which application specific defines is used in the compilation of the modules.

Makefile.binsensor_Battery_ZW020x_EU
Makefile.binsensor_Battery_ZW020x_US
Makefile.binsensor_Battery_ZW020x_ANZ
Makefile.binsensor_Battery_ZW020x_HK

This file is part of the make job. Is the main makefile for making the ANZ/EU/HK/US version of the Bin_Sensor_Battery to the ZW0201 Z-Wave Slave/Controller Unit, with description of which modules is included in the application and which application specific defines is used in the compilation of the modules.

Makefile.binsensor_Battery_ZW030x_EU
Makefile.binsensor_Battery_ZW030x_US
Makefile.binsensor_Battery_ZW030x_ANZ
Makefile.binsensor_Battery_ZW030x_HK

This file is part of the make job. Is the main makefile for making the ANZ/EU/HK/US version of the Bin_Sensor_Battery to the ZW0301 Z-Wave Slave/Controller Unit, with description of which modules is included in the application and which application specific defines is used in the compilation of the modules.

Bin_Sensor_Batt.c

Contains the source for the Bin_Sensor_Battery application.

CONFIDENTIAL

Bin_Sensor_Batt.h

Contains definitions for the Bin_Sensor_Battery application.

Please refer to section 3.3.13 for the utility library functions which are used by the application.

CONFIDENTIAL

Macros for accessing the LED's:

LED_ON(led)

Turn LED on.

Parameter:
led - LED number

Example:
```
   PIN_OUT(LED1); /* define LED1 as an output pin */
      LED_ON(1);       /* turn LED 1 on */
```

LED_OFF(led)

Turn LED off.

Parameter:
led - LED number

Example:
```
      LED_OFF(1);       /* turn LED 1 off */
```

LED_TOGGLE(led)

Toggle the LED OFF if the LED was ON and ON if the LED was OFF.

Parameter:
led - LED number

Example:
```
  LED_TOGGLE(1);       /* toggle LED 1 */
```

CONFIDENTIAL

7.4 Development Controller Sample Code

The Developer's Kit contains sample code that demonstrates how the basic tasks of adding, removing and controlling devices in a Z-Wave network can be accomplished using the Z-Wave API.

The Application complies with the Generic Controller command class (see [1]). The controller does not advertise any command classes in the node information frame. The Controller is able to control devices using the Binary Switch and Multilevel Switch command classes.

The Z-Wave basis software continually calls the **ApplicationPoll** function. The **ApplicationPoll** function contains a state machine, which initiates actions from user input. The **ApplicationCommandHandler** function is only called when the Z-Wave basis software receives information for the application. This could be a BASIC_REPORT indicating the Dim level of a Multilevel switch.

The Controller application is based on the Controller API using 5 push buttons mounted on the Development module. Any information to the user is indicated with 2 LED's also mounted on the Development module.

7.4.1 Dev_Ctrl Files.

The Product\Dev_ctrl directory contains the source code for the controller application.

Makefile

This file is part of the make job. It creates the directory structure and defines the targets that can be selected during make. The following compiler control line defines are used in the makefiles:

US	: Build US frequency target.
EU	: Build EU frequency target.
ANZ	: Build ANZ frequency target.
HK	: Build HK frequency target.
ZW_DEBUG	: Enable text output via UART.
ZW_DEBUG_CMD	: Enable command line debug via UART.
ZW_ID_SERVER	: Enable development controller as SIS.

cmd_class.h

The defines in this file is used to access ZW_frames that can not be accessed using the structs defined in ZW_classcmd.h

dev_ctrl.c

This file contains the main source code for the application. Both **ApplicationPoll** and **ApplicationCommandHandler** are defined in this file.

dev_ctrl_if.h

This file defines how the IO connections on the Z-Wave module are connected to the Development Module.

eeprom.c + eeprom.h

These files contain functions and define for accessing the application data in the external EEPROM.

p_button.c + p_button.h

CONFIDENTIAL

These files contain functions and define for detecting Push button presses. This includes Debounce checking.

CONFIDENTIAL

7.5 Door Bell Sample Code

The developer's Kit contains sample code for a Door Bell sample application. This device an example of how a battery operated chime in a door bell system could be build. The Door Bell uses the frequently listening mode where it powers up the radio for a short period every 1 second and if it receives a command it will power up entirely and turn on the LED's.

The Door Bell based on the routing slave library and it has its generic device class set to Binary Switch and the specific device class set to none. The Door Bell supports the following command classes

- Binary Switch command class
- Version command class

NOTE: This node will fail certification because when its level is set to on with a binary set command it will toggle its state back to off again after a timeout to emulate the behavior of a door bell.

7.5.1 User interface

The button on the Z-Wave module has the following functionality in the Door Bell:

Press shortly	Wakeup for 2 sec and send out node information frame
Press and hold for 1s	Enter learn mode and timeout after 3 sec

The LEDs on the Z-Wave module has the following meaning:

LED 0	LED 1	LED 2	Description
Off	Off	Off	The door bell is in powerdown mode (Frequently listening mode)
On	Off	Off	The node was woken up by button press or reset
Off	On	Off	The node was woken up by an RF beam
Off	Off	On	The node is in learn mode
On	On	On	Bell was turned on by Binary or Basic set command

7.5.2 Door Bell Files

The Product\DoorBell directory contains the source code and makefiles for the application.

Mk.bat

Batch file to start compiling the sample application

Makefile

Theist file defines the targets that can be built in this directory.

Makefile.doorbell_common

This makefile defines what source files that should be compiled for a specific target and it calls the appropriate makefiles in the Product\Common directory.

Bell.c + Bell.h

CONFIDENTIAL

This file contains the source code for the Door Bell sample application

CONFIDENTIAL

7.6 LED Dimmer Sample Code

The Developer's Kit contains sample code for a LED Dimmer. This device is in effect a light switch with a built in dimmer where the light bulb is substituted with 4 LED's when using ZW0102. In case a ZW0201 is used then the light bulb is substituted with only 3 LED's because it has an output pin less. A controller such as the Development Controller sample code can control the LED dimmer.

The LED dimmer is a multilevel switch that supports the all switch command class, the protection command class and the powerlevel command class described in the device class specification (see ref [1]). This device complies with the specific device class named multilevel power switch device class (4.1). The LED dimmer does not support the optional basic clock command class. The LED dimmer advertises via the node information frame support for the following command classes:

- Multilevel Switch command class
- All Switch command class
- Protection command class
- Powerlevel command class
- Version command class
- Manufacturer Specific command class

The LED dimmer is a slave device based on the slave API. During initialization, the LED dimmer will initialize the mounted button and the 4 LED's. It will also get stored data from the non-volatile memory. After the initialization the Z-Wave basis software will continually call the **ApplicationPoll** function, which contains the LED dimmer main function. The **ApplicationPoll** function checks if the button has been pressed and act according to the state the LED dimmer is in. The other main function is the **ApplicationCommandHandler** function that is called every time a command has been received, destined for the LED dimmer. This function checks the command and acts according to the command.

7.6.1 Build Targets

The LED dimmer sample application will as default create 4 targets for ZW0102. The 2 standard EU and US targets for the ZM1220 module and in addition EU and US target for the ZM1206 module where _sff_ is added to the target name. The only difference between the two target types is that the ZM1206 module uses the IO10 pin to detect the production test mode and the ZM1220 module uses the ZEROX pin.

CONFIDENTIAL

7.6.2 LED_Dimmer Files

The Product\LED_Dimmer directory contains sample source code for a slave application on a Z-Wave module.

Makefile

This file is part of the make job. It creates the directory structure and defines targets. The following compiler control line defines are used in the makefiles:

US	: Build US frequency target.
EU	: Build EU frequency target.
ANZ	: Build ANZ frequency target.
HK	: Build HK frequency target.
SFF	: When using the small form factor module ZM1206.
NOOFFONDIM	: When enabled not all the LED's will turn off when dimming.
APPL_PROD_TEST	: Enable the production test.

Makefile.leddimmer_ZW010x_EU
Makefile.leddimmer_ZW010x_US

This file is part of the make job. Is the main makefile for making the EU/US version of the LEDdimmer with description of which modules is included in the application and which application specific defines is used in the compilation of the modules.

Makefile.leddimmer_ZW010x_sff_EU
Makefile.leddimmer_ZW010x_sff_US

This file is part of the make job. Is the main makefile for making the EU/US version of the LEDdimmer to the ZM1206 module, with description of which modules is included in the application and which application specific defines is used in the compilation of the modules.

Makefile.leddimmer_ZW020x_EU
Makefile.leddimmer_ZW020x_US
Makefile.leddimmer_ZW020x_ANZ
Makefile.leddimmer_ZW020x_HK

This file is part of the make job. Is the main makefile for making the ANZ/EU/HK/US version of the LEDdimmer to the ZW0201 Z-Wave Slave/Controller Unit, with description of which modules is included in the application and which application specific defines is used in the compilation of the modules.

Makefile.leddimmer_ZW030x_EU
Makefile.leddimmer_ZW030x_US
Makefile.leddimmer_ZW030x_ANZ
Makefile.leddimmer_ZW030x_HK

This file is part of the make job. Is the main makefile for making the ANZ/EU/HK/US version of the LEDdimmer to the ZW0301 Z-Wave Slave/Controller Unit, with description of which modules is included in the application and which application specific defines is used in the compilation of the modules.

LEDdim.c

Contains the source for the LEDdimmer application.

LEDdim.h

Contains definitions for the LEDdimmer application.

CONFIDENTIAL

Macros for accessing the LED's:

LED_ON(led)

Turn LED on.

Parameter:
    led - LED number

Example:
```
    PIN_OUT(LED1); /* define LED1 as an output pin */
       LED_ON(1);      /* turn LED 1 on */
```

LED_OFF(led)

Turn LED off.

Parameter:
    led - LED number

Example:
```
       LED_OFF(1);      /* turn LED 1 off */
```

LED_TOGGLE(led)

Toggle the LED OFF if the LED was ON and ON if the LED was OFF.

Parameter:
    led - LED number

Example:
```
       LED_TOGGLE(1);      /* toggle LED 1 */
```

CONFIDENTIAL

7.7 Production Test DUT

The Developer's Kit contains sample code that demonstrates how the basic tasks of testing devices in a Z-Wave network can be accomplished using the Z-Wave API.

The Z-Wave basis software continually calls the **ApplicationPoll** function. The **ApplicationPoll** function contains a state machine, which initiates actions from user input.

The Production Test DUT sample application is based on the ZW_slave_prodtest_dut library.

This sample application has two functions that can be used during production test

- The radio will start to transmit continuously if the P1.5 (SS_N) pin on the ZM2102 is pulled low during power up.
- If the pin isn't pulled low the radio will go into receive mode and send acknowledge to all frames send to the module.

The program execution flow is as follows:

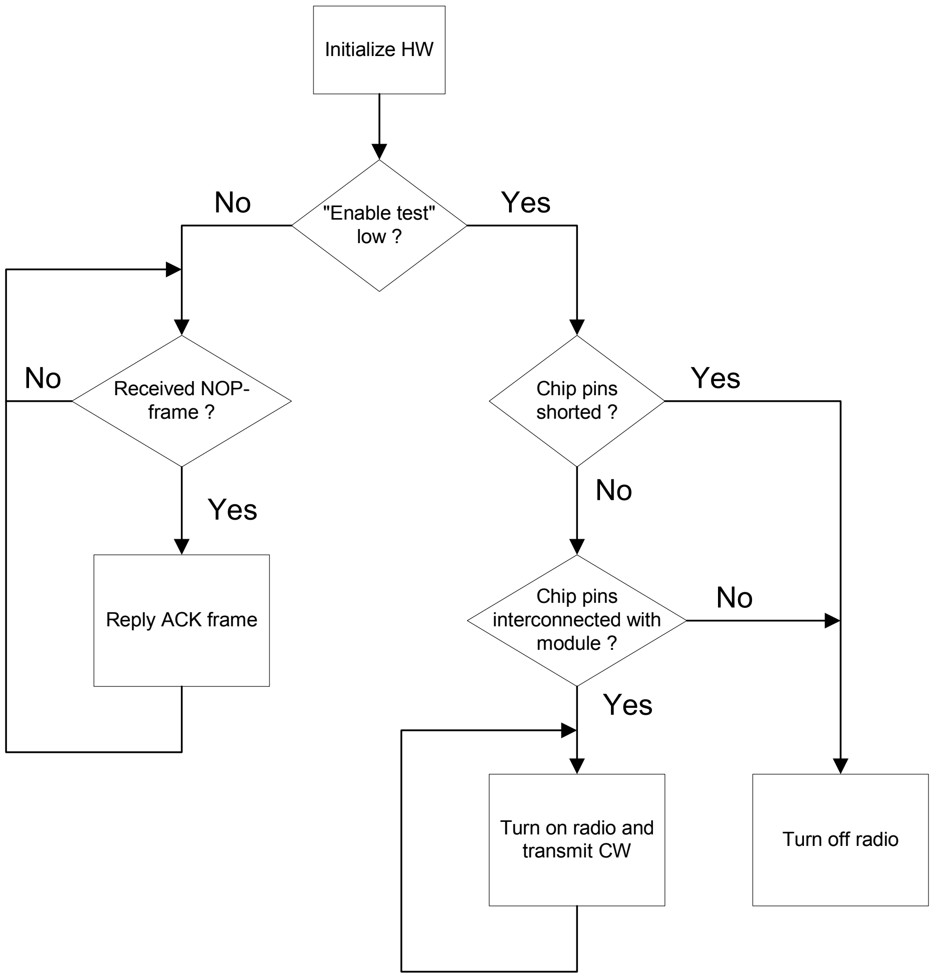


Figure 12, Prod_Test_DUT test program flow

CONFIDENTIAL

Immediately after program execution start, the state of the “Enable test” pin (pin P1.5 on the ZW0201) is tested. If the pin is high, “normal” Z-Wave slave code is started, and the DUT will reply to a NOP frame with an ACK frame. This behavior is used during the Link test, where 10 NOP’s are transmitted from the Z-Wave test box and 10 ACK’s are expected from the DUT.

If the “Enable test” pin is low, a test program flow is started.

All pins not used during chip programming and test enabling are tested for shorts. The CPU of the ZW0201 writes a “0101…” pattern to its pins and then reads back the state of the pins. If no shorts, the pattern read will be “0101...”. Then a “1010…” pattern is written, and “1010...” is expected when reading back.

The interconnections form the ZW0201 chip to the castellation notched of the ZM2102 module are tested. The CPU enables the internal pull-up resistors on the ZW0201 chip and sets the pins high. The read back of the pins should then be high. All pins are then set to low, and because of the external 10 kOhm pull down resistors, all pins should be read back as being low. If an interconnection fails, the internal pull up will lead the CPU to read the pin as being high instead of low.

If one of the above tests fails, the radio will be turned off.

If both of the above tests pass, the radio will be turned on to transmit a CW, and the spectrum analyzer is then able to measure the RF frequency and RF output power.

7.7.1 Production Test DUT Files

The Product\Prod_Test_DUT directory contains the source code for the Production Test DUT sample application.

Makefile

This file is part of the make job. It creates the directory structure and defines the targets that can be selected during make. The following compiler control line defines are used in the makefiles:

US	: Build US frequency target.
EU	: Build EU frequency target.
ANZ	: Build ANZ frequency target.
HK	: Build HK frequency target.

prodtestdut.c

This file contains the main source code for the sample application. Both **ApplicationPoll** and **ApplicationCommandHandler** are defined in this file.

prodtestdut.h

This file contains definitions for the prod_test_dut sample application.

CONFIDENTIAL

7.8 Production Test Generator

The Developer's Kit contains sample code that demonstrates how the basic tasks of testing devices in a Z-Wave network can be accomplished using the Z-Wave API. The Z-Wave generator is used to verify the TX / RX circuits on Z-Wave enabled products.

The generator consists of an Interface Module and a ZMxx20 Z-Wave Module to which a push-button and two LED's, a red LED and a green LED are connected. After connection to power, the red LED will be on. The green LED is the LED position 'D6' on the Interface Module and the red LED is the LED position 'D2'. The push button is the 'S1' push button on the ZMxx20 Z-Wave Module.

When the button is pressed, 10 NOP's will be transmitted through the RF-connector of the generator. The DUT is expected to verify the reception of each NOP with an ACK. During transmission, the red LED will blink.

If all NOP's are replied correctly, the red LED will turn off and the green LED will turn on and be on until the next test is conducted. If the DUT does not reply correct, the red LED will stop blinking and turn on.

The principal schematics of the generator looks like this :

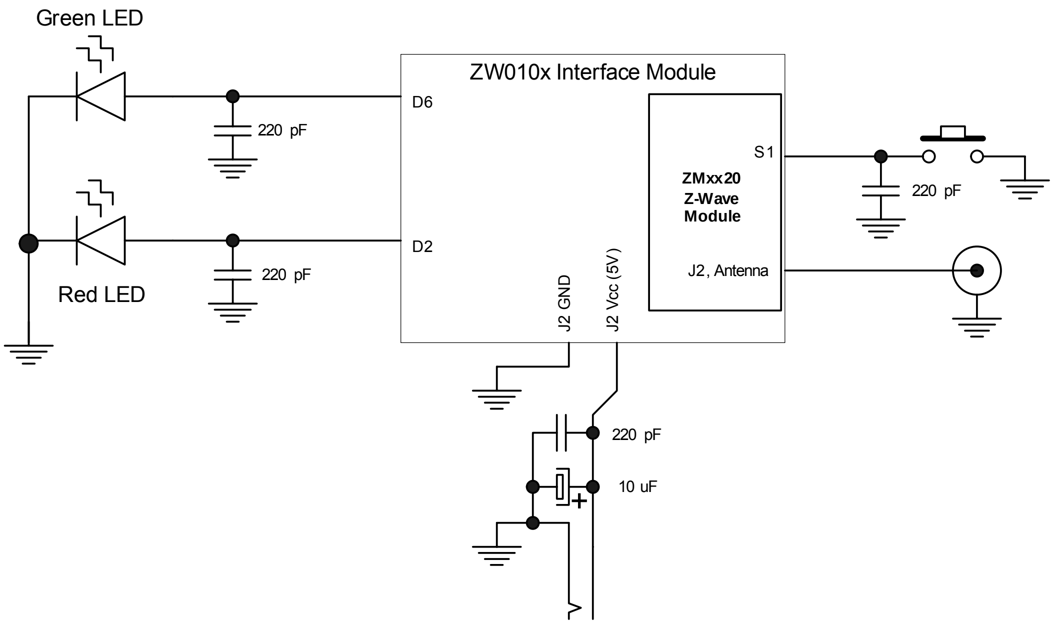


Figure 13, Z-Wave test generator

The Z-Wave basis software continually calls the **ApplicationPoll** function. The **ApplicationPoll** function contains a state machine, which initiates actions from user input. The **ApplicationCommandHandler** function is only called when the Z-Wave basis software receives information for the application.

The Production Test Generator sample application is based on the ZW_slave_prodtest_gen library.

CONFIDENTIAL

The application is controlled via RS232 (115200,8,N,1) or button with fixed timings:
Device will resopond to any char received with an ASCII SPACE followed by a command answer or error '!' followed by error information:
Following ASCII commands are implemented.
Received:

'U':
Frequency US is selected
Response is: ' ' 'U' 'S'

'E':
Frequency EU is selected
Response is: ' ' 'E' 'U'

'S':
Start test
Response is ' ' 'S' 'T'

'C':
Set the number of NOPs to send
Response: ' ' 'C' 'O'

'N':
Set the destination node ID.
Response: ' ' 'N ''I'

'R':
Reset the hardware
Response: ' ' 'R' 'S'

For ZW020x series and ZW030x series
'B':
'4' - Use 40KBit. Response "B:40k"
'9' - Use 9.6kbit. Response "B:9.6k"

On Unknown:
'!' 'received Char'

CONFIDENTIAL

7.8.1 Production Test Generator Files

The Product\Prod_Test_Gen directory contains the source code for the Production Test Generator sample application.

Makefile

This file is part of the make job. It creates the directory structure and defines the targets that can be selected during make. The following compiler control line defines are used in the makefiles:

US	: Build US frequency target.
EU	: Build EU frequency target.
ANZ	: Build ANZ frequency target.
HK	: Build HK frequency target.

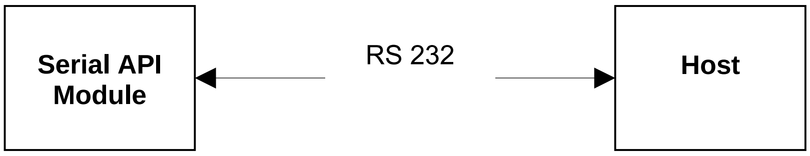
prod_test_gen.c

This file contains the main source code for the sample application. Both **ApplicationPoll** and **ApplicationCommandHandler** are defined in this file.

7.9 Serial API Embedded Sample Code

The purpose of the Serial API embedded sample code is to show how a ZW0102/ZW0201/ZW0301 Z-Wave module can be controlled via the serial port by a host. The following host based PC applications are available on the Developer's Kit CD:

- The PC based Controller application showing the available functionality in a Serial API based on a static controller API.
- The PC based Installer Tool application showing the available functionality in a Serial API based on an installer API.
- The PC based Z-Wave Bridge application showing the available functionality in a Serial API based on a bridge controller API.



The Serial API can be used as it is or it can be changed to fit specific needs. If changing it be aware that the serial debug commands described in section 5.3.10.15 cannot be used to debug the application, as the UART already is used by the Serial API communication. The UART on the Z-Wave Module is initialized for 115200 baud, no parity, 8 data bits and 1 stop bit.

7.9.1 Supported API Calls

Only a subset of the API calls is available via the serial interface. In Chapter 5 each API call has a description regarding Serial API support and the corresponding frame format and flow.

7.9.2 Implementation

The Serial API embedded sample code is provided on the Z-Wave Developer's Kit. Be aware that altering the function ID's and frame formats in the Serial API embedded sample code can result in interoperability problems with the Z-Wave DLL supplied on the Developer's Kit as well as commercially available GUI applications. Regarding how to determine the current version of the Serial API protocol in the embedded sample code please refer to the API call **ZW_Version**. The following sections describe the Serial API implementation and how a host can communicate with the Serial API embedded sample code.

7.9.2.1 Frame Layout

The protocol between the PC (host) and the Z-Wave Module (ZW) consists of three frame types: ACK frame, NAK frame and Data frame. Each Data frame is prefixed with SOF byte and Length byte and suffixed with a Checksum byte. As of Serial API Version 4 a fourth frame type has been defined; the CAN frame.

ACK frame:

The ACK frame is used to acknowledge a successful transmission of a data frame. The format is as follows:

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

NAK frame:

The NAK frame is used to de-acknowledge an unsuccessful transmission of a data frame. The format is as follows:

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

Only a frame with a LRC checksum error is de-acknowledged with a NAK frame.

CAN frame:

The CAN frame is used by the ZW to instruct the host that a host transmitted data frame has been dropped. The format is as follows:

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

CONFIDENTIAL

Data frame:

The Data frame contains the Serial API command including parameters for the command in question. The format is as follows:

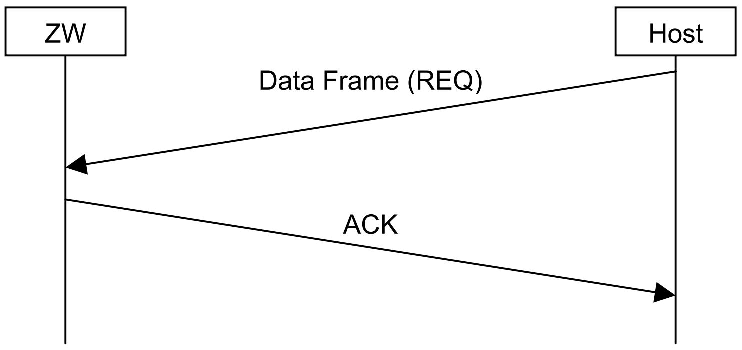
7	6	5	4	3	2	1	0
SOF							
Length							
Type							
Serial API Command ID							
Command Specific Data							
…							
Checksum							

SOF	Start Of Frame. Used for synchronization and is equal to 0x01
Length	Number of bytes in the frame, exclusive SOF and Checksum. The host application is responsible for entering the correct length field. The current Serial API embedded sample code does no validation og the length field.
Type	Used to distinguish between unsolicited calls and immediate responses (not callback). The request (REQ) is equal to 0x00 and response (RES) is equal to 0x01.
Serial API Command ID	Unique command ID for the function to be carried out. Any data frames returned by this function will contain the same command ID
Command Specific Data	One or more bytes of command specific data. Possible callback handling is also defined here.
Checksum	LRC checksum used to check for frame integrity. Checksum calculation includes the **Length**, **Type**, **Serial API Command Data** and **Command Specific Data** fields. The Checksum is a XOR checksum with an initial checksum value of 0xFF. For a checksum implementation refer to the function ConTxFrame in the conhandle.c module

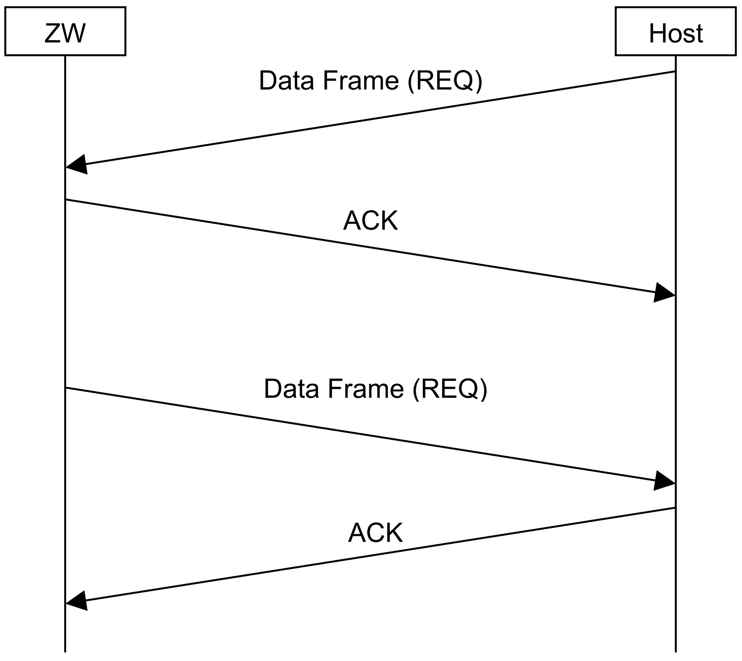
CONFIDENTIAL

7.9.2.2 Frame Flow

The frame flow between a host and a Z-Wave module (ZW) running the Serial API embedded sample code depends on the API call. There are four different ways to conduct communication between the host and ZW.

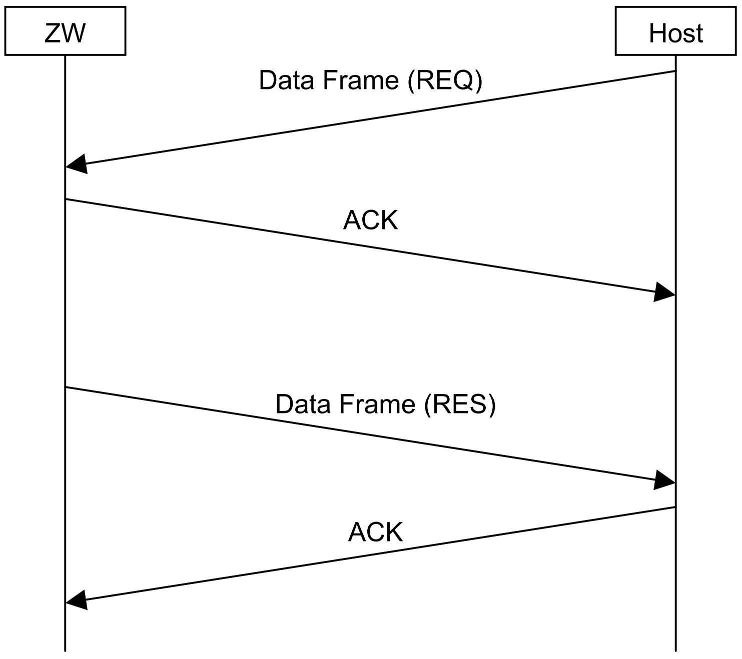


Data frame from host, which is acknowledged by ZW when successfully received. An example could be the API call **ZW_LockRoute**.



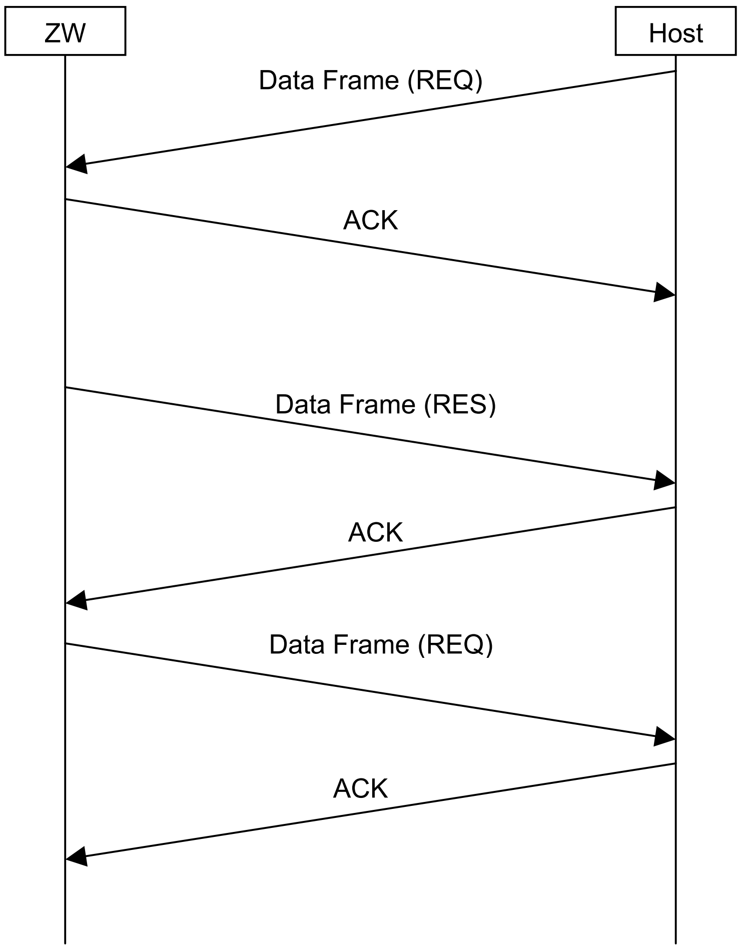
Data frame with callback function enabled from host, which is acknowledged by ZW when successfully received. A data frame (callback) is returned by ZW with the result at command completion. The host acknowledged the data frame when successfully received. Setting the funcID equal to 0 in the data frame disable the callback handling. An example could be the API call **ZW_SetDefault**.

CONFIDENTIAL



Data frame from host, which is acknowledged by ZW when successfully received. A data frame (RES) is returned by ZW with the result at command completion. The host acknowledges the data frame when successfully received. An example could be the API call **ZW_GetControllerCapabilities**.

CONFIDENTIAL



Data frame with callback function enabled from host, which is acknowledged by ZW when successfully received. A data frame (RES) is returned by ZW with the status at command initiation. The host acknowledges the data frame when successfully received. A data frame (callback) is returned by ZW with the result at command completion. The host acknowledges the data frame when successfully received. An example could be the API call **ZW_RequestNodeNeighborUpdate**.

7.9.2.3 Error handling

A number of scenarios exist, which can impede the normal frame flow between the host and the Z-Wave module running the Serial API embedded sample code (ZW).

A LRC checksum failure is the only case there is de-acknowledged by a NAK frame in the current Serial API embedded sample code. When a host receives a NAK frame can it either retry transmission of the frame or abandon the task. A task is defined as the whole frame flow associated with the execution of a specific Serial API function call. If a NAK frame is received by the Z-Wave module in response to a just transmitted frame, then the frame in question is retransmitted (max 2 retries).

Frames with an illegal length are ignored without any notification. Frames with an illegal type (only REQ and RES exists) are ignored without any notification

The Serial API embedded sample code can only perform one host-initiated task at a time. A data frame will be dropped without any notification (no ACK/NAK frame transmitted) by the ZW if it is not ready to execute a new host-initiated task. As of Serial API version 4 a CAN frame is transmitted by the ZW when a received data frame is dropped.

CONFIDENTIAL

If no CAN frame is received the host detect the missing ACK/NAK by implementing a timeout mechanism in the receive function. The host timeout must correspond to the timeout defined in ZW. A reasonable timeout in the host is 2 seconds because the current Serial API embedded sample code has a default timeout of 1.5 seconds. The timeout in the Serial API (as of SerialAPI version 4) can also be set by using the FUNC_ID_SERIAL_API_SET_TIMEOUTS Serial API function:

Serial API:

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

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

RXACKTimeout is the max no. of 10ms ticks the ZW waits for an ACK before timeout. RXBYTETimeout is the max no. of 10ms ticks the ZW waits for a new byte before timeout; this is only valid when a frame has been detected and is being collected.

In case the host expect an ACK but instead receive another data frame then it must read the whole data frame and ACK/NAK accordingly, it will probably also receive a CAN frame to indicate that the ZW has dropped the host transmitted data frame. Afterwards can the host restart transmission of the pending frame ZW never ACK'ed or possibly CAN'ed.

Communication between ZW and other Z-Wave nodes can also result in deviations from the normal frame flow. A get command on application level can for example result in multiple reports coming back and ZW will just pass on the reports to the host. This can happen in case the Z-Wave node did not hear ZW acknowledge the report and therefore it is retransmitted. To handle such scenarios requires a relaxed state machine on application level to handle multiple reports. The same apply for set and get commands.

7.9.2.4 Restrictions on functions using buffers

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

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

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

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

7.9.2.5 Serial API capabilities

As of Serial API protocol version 4 (to determine Serial API protocol version please refer to the Serial API Function described under the Z-Wave API Function **ZW_Version**) it is possible to determine exactly which Serial API functions a specific Serial API Z-Wave Module supports with the FUNC_ID_SERIAL_API_GET_CAPABILITIES Serial API function:

CONFIDENTIAL

Serial API:

HOST->ZW: REQ | 0x07

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

SERIAL_APPL_VERSION is the Serial API application Version number.

SERIAL_APPL_REVISION is the Serial API application Revision number.

SERIALAPI_MANUFACTURER_ID1 is the Serial API application manufacturer_id (MSB).

SERIALAPI_MANUFACTURER_ID2 is the Serial API application manufacturer_id (LSB).

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

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

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

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

FUNCID_SUPPORTED_BITMASK[ ] is a bitmask where every Serial API function ID which is supported has a corresponding bit in the bitmask set to ‘1’. All Serial API function IDs which are not supported have their corresponding bit set to ‘0’. First byte in bitmask corresponds to FuncIDs 1-8 where bit 0 corresponds to FuncID 1 and bit 7 corresponds to FuncID 8. Second byte in bitmask then corresponds to FuncIDs 9-16 and so on.

7.9.2.6 Serial API Softreset

It is possible to make the Z-Wave module do a software reset by using the Serial API function FUNC_ID_SERIAL_API_SOFT_RESET**:**

Serial API:

HOST->ZW: REQ | 0x08

CONFIDENTIAL

7.9.2.7 Build Targets

The Serial API embedded sample code has more targets than the other Z-Wave sample applications. This application builds a serial API for several of the library types in Z-Wave. The targets shown below will be built:

A serial API based on the bridge controller API:

Serialapi_bridge_ZW010x_EU
Serialapi_bridge_ZW010x_US
Serialapi_bridge_ZW020x_EU
Serialapi_bridge_ZW020x_US
Serialapi_bridge_ZW020x_ANZ
Serialapi_bridge_ZW020x_HK
Serialapi_bridge_ZW030x_EU
Serialapi_bridge_ZW030x_US
Serialapi_bridge_ZW030x_ANZ
Serialapi_bridge_ZW030x_HK

A serial API based on the static controller API:

Serialapi_ctrl_static_ZW010x_EU
Serialapi_ctrl_static_ZW010x_US
Serialapi_ctrl_static_ZW020x_EU
Serialapi_ctrl_static_ZW020x_US
Serialapi_ctrl_static_ZW020x_ANZ
Serialapi_ctrl_static_ZW020x_HK
Serialapi_ctrl_static_ZW030x_EU
Serialapi_ctrl_static_ZW030x_US
Serialapi_ctrl_static_ZW030x_ANZ
Serialapi_ctrl_static_ZW030x_HK

A serial API based on the controller API:

Serialapi_ctrl_ZW010x_EU
Serialapi_ctrl_ZW010x_US
Serialapi_ctrl_ZW020x_EU
Serialapi_ctrl_ZW020x_US
Serialapi_ctrl_ZW020x_ANZ
Serialapi_ctrl_ZW020x_HK
Serialapi_ctrl_ZW030x_EU
Serialapi_ctrl_ZW030x_US
Serialapi_ctrl_ZW030x_ANZ
Serialapi_ctrl_ZW030x_HK

A serial API based on the installer API:

Serialapi_inst_ZW010x_EU
Serialapi_inst_ZW010x_US
Serialapi_inst_ZW020x_EU
Serialapi_inst_ZW020x_US
Serialapi_inst_ZW020x_ANZ
Serialapi_inst_ZW020x_HK
Serialapi_inst_ZW030x_EU
Serialapi_inst_ZW030x_US
Serialapi_inst_ZW030x_ANZ
Serialapi_inst_ZW030x_HK

CONFIDENTIAL

A serial API based on the slave library:

Serialapi_slave_ZW010x_EU
Serialapi_slave_ZW010x_US
Serialapi_slave_ZW020x_EU
Serialapi_slave_ZW020x_US
Serialapi_slave_ZW020x_ANZ
Serialapi_slave_ZW020x_HK
Serialapi_slave_ZW030x_EU
Serialapi_slave_ZW030x_US
Serialapi_slave_ZW030x_ANZ
Serialapi_slave_ZW030x_HK

7.9.2.8 Serial API Files

Makefile

This file is part of the make job. It creates the directory structure and defines targets. The following compiler control line defines are used in the makefiles:

US	: Build US frequency target.
EU	: Build EU frequency target.
ANZ	: Build ANZ frequency target.
HK	: Build HK frequency target.
LOW_FOR_ON	: Not used.
SIMPLELED	: Not used.

Makefile.serialapi_bridge_ZW010x_EU
Makefile.serialapi_ctrl_static_ZW010x_EU
Makefile.serialapi_ctrl_ZW010x_EU
Makefile.serialapi_installer_ZW010x_EU
Makefile.serialapi_slave_ZW010x_EU

Makefile.serialapi_bridge_ZW020x_EU
Makefile.serialapi_ctrl_static_ZW020x_EU
Makefile.serialapi_ctrl_ZW020x_EU
Makefile.serialapi_installer_ZW020x_EU
Makefile.serialapi_slave_ZW020x_EU

Makefile.serialapi_bridge_ZW030x_EU
Makefile.serialapi_ctrl_static_ZW030x_EU
Makefile.serialapi_ctrl_ZW030x_EU
Makefile.serialapi_installer_ZW030x_EU
Makefile.serialapi_slave_ZW030x_EU

This file is part of the make job. Is the makefile used for making the EU version of the Serial API build (bridge, controller, installer and slave version) with description of which modules is included in the application and which application specific defines is used in the compilation of the modules.

Makefile.serialapi_bridge_ZW010x_US
Makefile.serialapi_ctrl_static_ZW010x_US
Makefile.serialapi_ctrl_ZW010x_US
Makefile.serialapi_installer_ZW010x_US
Makefile.serialapi_slave_ZW010x_US

Makefile.serialapi_bridge_ZW020x_US
Makefile.serialapi_ctrl_static_ZW020x_US

CONFIDENTIAL

Makefile.serialapi_ctrl_ZW020x_US
Makefile.serialapi_installer_ZW020x_US
Makefile.serialapi_slave_ZW020x_US

Makefile.serialapi_bridge_ZW030x_US
Makefile.serialapi_ctrl_static_ZW030x_US
Makefile.serialapi_ctrl_ZW030x_US
Makefile.serialapi_installer_ZW030x_US
Makefile.serialapi_slave_ZW030x_US

This file is part of the make job. Is the makefile used for making the US version of the Serial API build (bridge, controller, installer and slave version) with description of which modules is included in the application and which application specific defines is used in the compilation of the modules.

Makefile.serialapi_bridge_ZW020x_ANZ
Makefile.serialapi_ctrl_static_ZW020x_ANZ
Makefile.serialapi_ctrl_ZW020x_ANZ
Makefile.serialapi_installer_ZW020x_ANZ
Makefile.serialapi_slave_ZW020x_ANZ

Makefile.serialapi_bridge_ZW030x_ANZ
Makefile.serialapi_ctrl_static_ZW030x_ANZ
Makefile.serialapi_ctrl_ZW030x_ANZ
Makefile.serialapi_installer_ZW030x_ANZ
Makefile.serialapi_slave_ZW030x_ANZ

This file is part of the make job. Is the makefile used for making the ANZ version of the Serial API build (bridge, controller, installer and slave version) with description of which modules is included in the application and which application specific defines is used in the compilation of the modules.

Makefile.serialapi_bridge_ZW020x_HK
Makefile.serialapi_ctrl_static_ZW020x_HK
Makefile.serialapi_ctrl_ZW020x_HK
Makefile.serialapi_installer_ZW020x_HK
Makefile.serialapi_slave_ZW020x_HK

Makefile.serialapi_bridge_ZW030x_HK
Makefile.serialapi_ctrl_static_ZW030x_HK
Makefile.serialapi_ctrl_ZW030x_HK
Makefile.serialapi_installer_ZW030x_HK
Makefile.serialapi_slave_ZW030x_HK

This file is part of the make job. Is the makefile used for making the HK version of the Serial API build (bridge, controller, installer and slave version) with description of which modules is included in the application and which application specific defines is used in the compilation of the modules.

UART_buf_io.c / UART_buf_io.h

Low level routines for handling buffered transmit/receive of data through the UART.

conhandle.c / conhandle.h

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

serialappl.c

This module implements the handling of Serial API protocol. That is, parses the frames, calls the appropriate Z-Wave API library functions and returns results etc. to the PC.

CONFIDENTIAL

7.10 Smoke Detector Sample Code

The developer’s Kit contains sample code for a Smoke Detector sample application. This device an example of how a battery operated smoke detector system could be build. The Smoke Detector uses the Sensor Net Routing Slave where it powers up the radio for a short period every 1 second and if it receives a command it will power up entirely and turn on the LED’s.

The Smoke Detector is based on the Sensor Net Routing Slave library and it has its generic device class set to Binary Switch and the specific device class set to none. The Smoke Sensor supports the following command classes

- Binary Switch command class
- Version command class

NOTE: Due to the limited user interface that can be made with one button the Smoke Detector does not by default support being added to a normal Z-Wave network. However the source code for rigs functionality is in the application it can just not be activated using the button.

7.10.1 User interface

The button on the Z-Wave module has the following functionality in the Smoke Detector:

Press shortly once	Enter bind mode for 2 sec.
Press shortly 2 times	Send an alarm
Press and hold button for 1 sec.	Reset node to factory default

NOTE: When 2 Smoke Detectors are bound the first time one of the nodes will become master in the system and must be used to add additional Smoke Detectors to the system. The node that becomes master will stay in bind mode for an additional 8 seconds after the binding is complete.

The LEDs on the Z-Wave module has the following meaning:

LED 0	LED 1	LED 2	Description
Off	Off	Off	The Smoke Detector is in powerdown mode (Frequently listening mode)
On	Off	Off	The Smoke Detector is awake
On	On	Off	The Smoke Detector is in bind mode
On	Of	On	The Smoke Detector is reset
On	On	On	The Smoke Detector is sending out an alarm
Blink	Blink	Blink	The Smoke Detector has received an alarm

7.10.2 Smoke Detector Files

The Product\SmokeSensor directory contains the source code and makefiles for the application.

Mk.bat

Batch file to start compiling the sample application

Makefile

This file defines the targets that can be built in this directory.

CONFIDENTIAL

Makefile.smokesensor_common

This makefile defines what source files that should be compiled for a specific target and it calls the appropriate makefiles in the Product\Common directory.

SmokeSensor.c + SmokeSensor.h

This file contains the source code for the Smoke Detector sample application

CONFIDENTIAL

7.10.3 PC based Controller Sample Application

The PC\PcController directory contains sample application source code in C# that implements a PC based Controller using the development tool Visual Studio 2005.

For further information about the features of the PC based Controller, see [6].

7.10.4 PC based Installer Tool Sample Application

The PC based Installer Tool directory contains a sample application that illustrates the functions in the Z-Wave Installer API.

For further information about the features of the PC based Installer Tool, see [7].

ReadMe.txt

Readme file for the project.

InstallerTool.dsw + InstallerTool.dsp

Project files for loading the project in Microsoft Visual C++ v6.0.

InstallerTool.cpp + InstallerTool.h

This is the main application class CInstallerTool. The files are as created by the MFC AppWizard.

InstallerTool.rc

This is a listing of all of the Microsoft Windows resources that the program uses. It defines the layout of the dialog.

Resource.h

Defines all resource IDs.

InstallerToolDlg.cpp + InstallerToolDlg.h

Implements the CInstallerToolDlg class, which is where the actual Serial API application is.

StdAfx.cpp + StdAfx.h

These files are used to build a precompiled header file. MFC AppWizard provides the files.

res\InstallerTool.ico

Icon file which is used as the applications icon.

res\InstallerTool.rc2

This file contains resources that are not edited by the Microsoft Visual C++ resource editor. Not used in this project.

CONFIDENTIAL

7.10.5 PC based Z-Wave Bridge Sample Application

The PC\Z-WaveBridge directory contains sample application source code in C# that implements a PC based Z-Wave to UPnP Bridge using the development tool Visual Studio 2005.

For further information about the features of the PC based Z-Wave to UPnP Bridge, see [8].

CONFIDENTIAL

8 REQUIRED DEVELOPMENT COMPONENTS

8.1 Software development components

There is an additional 3rd party software tool that is required to develop Z-Wave applications that is not supplied with the Z-Wave Developer's Kit. That is the Keil PK51 Professional Developer's Kit for the 8051 microcontroller:

A51 Macro Assembler	V7.10
C51 C compiler	V7.50
LX51 Linker/Locater	V3.65b
LIBX51 Librarian Manager	V4.24

Z-Wave libraries and sample applications are build and tested on above versions but newer version should also apply according to Keil's recommendations.

The Keil Developer's Kits can be purchased directly from Keil or from one of their local distributors. Please visit **www.keil.com** for details. Alternatively can it be purchased from Zensys.

Keil Software, Inc. 1501 10th Street, Suite 110 Plano, TX 75074 USA	
Toll Free:	**800-348-8051**
Phone:	972-312-1107
Fax:	972-312-1159
Sales:	**sales.us@keil.com**
Support:	**support.us@keil.com**

Keil Elektronik GmbH Bretonischer Ring 15 D-85630 Grasbrunn Germany	
Toll Free:	**-**
Phone:	(49) (089) 45 60 40 0
Fax:	(49) (089) 46 81 62
Sales:	**sales.intl@keil.com**
Support:	**support.intl@keil.com**

8.2 ZW0102/ZW0201/ZW0301 single chip programmer

This Z-Wave Developer's Kit comes with the Z-Wave Programmer included. The Z-Wave Programmer is used for downloading new firmware to the ZW0x0x Single Chip. For a detailed description refer to [14].

The Z-Wave Programmer is also used when programming the external EEPROM on the Z-Wave module. For further details refer to paragraph 8.7.

CONFIDENTIAL

8.3 Hardware development components for ZW0102

The ZW0102 based static controller serial API and all slave sample applications are designed for the ZW0102 Controller/Slave Unit, which is an assembly of the ZW0x0x Interface Module [2] and the ZM1220 Z-Wave Module [4].

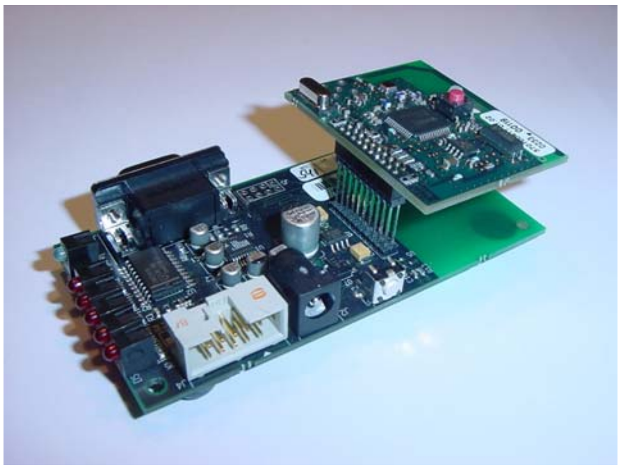


Figure 14 ZW0102 Controller/Slave Unit

The ZW0102 development controller sample application is designed for the ZW0102 Development Controller Unit, which is an assembly of the ZW0x0x Development Module [3] and the ZM1220 Z-Wave Module [4].

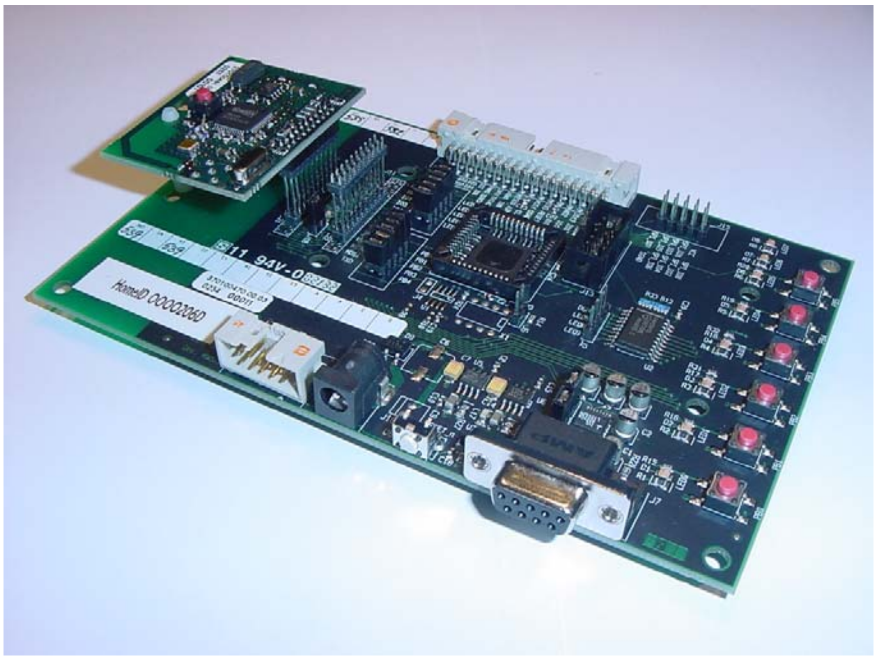


Figure 15 ZW0102 Development Controller Unit

CONFIDENTIAL

8.4 Hardware development components for ZW0201

The ZW0201 based static controller serial API and all slave sample applications are designed for the ZW0201 Controller/Slave Unit, which is an assembly of the ZW0x0x Interface Module [2], ZMxx06 Converter Module [17], and the ZM2106C Module (incl. ZM2102) [18]. Alternatively can it be an assembly of the ZW0x01 Interface Module [2], and the ZM2120C Module (incl. ZM2102) [23].

The ZW0201 development controller sample application is designed for the ZW0201 Development Controller Unit, which is an assembly of the ZW0x0x Development Module [3], ZMxx06 Converter Module [17] and the ZM2106C Module (incl. ZM2102) [18]. Alternatively can it be an assembly of the ZW0x0x Development Module [3], and the ZM2120C Module (incl. ZM2102) [23].

8.5 Hardware development components for ZW0301

The ZW0301 based static controller serial API and all slave sample applications are designed for the ZW0301 Controller/Slave Unit, which is an assembly of the ZW0x0x Interface Module [2], ZMxx06 Converter Module [17] and the ZM3106C Module (incl. ZM3102) [22]. Alternatively can it be an assembly of the ZW0x0x Interface Module [2], and the ZM3120C Module (incl. ZM3102) [24].

The ZW0301 development controller sample application is designed for the ZW0301 Development Controller Unit, which is an assembly of the ZW0x0x Development Module [3], ZMxx06 Converter Module [17] and the ZM3106C Module (incl. ZM3102) [22]. Alternatively can it be an assembly of the ZW0x0x Development Module [3], and the ZM3120C Module (incl. ZM3102) [24].

CONFIDENTIAL

8.6 ZW0102/ZW0201/ZW0301 lock bit settings

The ZW0102/ZW0201/ZW0301 lock bits related to protection of the flash contents should during development be set as follows:

Table 11. Lock bits settings during development

Lock bits	Value	Description
SPIRE	1	It is allowed to read the flash data via the SPI interface
BSIZE[2..0]	111	Boot sector size set to 0 bytes
BOBLOCK	1	Page 0 is writeable

This allows the developer to read contents of the flash. The possibility to read flash contents should be disabled in the end product to avoid copy production. The ZW0102/ZW0201/ZW0301 lock bits in end products should be set as follows:

Table 12. Lock bits settings in end products

Lock bits	Value	Description
SPIRE	0	It is not allowed to read the flash data via the SPI interface
BSIZE[2..0]	111	Boot sector size set to 0 bytes
BOBLOCK	1	Page 0 is writeable

Regarding a detailed description about flash programming and lock bits for ZW0102, ZW0201 and ZW0301, refer to [16], and [10] respectively.

CONFIDENTIAL

8.7 External EEPROM initialization

When creating a controller on a new ZW0102/ZW0201/ZW0301 based Z-Wave module a home ID must be allocated. The home ID is stored in the external EEPROM on the Z-Wave module. Beside the home ID the remaining part of the external EEPROM must be zeroed. The controller requires an initialized external EEPROM as describe above to operate correct. With respect to an enhanced slave then the whole external EEPROM must be zeroed before it can operate correct. The external EEPROM must only be initialized once when creating a controller or enhanced slave on a new Z-Wave module.

In the binary controller directories …\Product\Bin\ supplied on the Developer's Kit CD are the image files extern_eep.hex to be downloaded found. The 32-bit home ID (xxxxxxxx) is located in byte 8, 9, 10 and 11 (when counting from 0) in the file. Byte 8 is the most significant byte and byte 11 is the least significant.

```
:200000005A654E7359730000xxxxxxxx000000000000000000000000000000000000093
```

The procedure to initialize the external EEPROM on the Z-Wave module is described in [14].

The external EEPROM on the ZM1206 module can **only** be updated by the steps described below:

8. Use the Z-Wave Programmer [14] to download eeploader_ZW0102.hex file to the module.
9. Connect a RS-232 serial cable directly from the PC to the Z-Wave interface module. Notice that download of the extern_eep.hex file is not done via the Z-Wave programmer.
10. Go to the directory …\Tools\Eeprom_loader\ and open a command prompt (DOS box). The eeploader.exe program in this directory is used to download the extern_eep.hex file via the COM port.
11. To download the extern_eep.hex in the Development Controllers binary directory …\Product\Bin\Dev_Ctrl run **eeploader ..\..\Product\Bin\Dev_Ctrl\extern_eep.hex COMx**, where x is equal to the COM port number the cable is attached to.
12. The PC application displays download progress and finally download status.

CONFIDENTIAL

9 APPLICATION NOTE: SUC/SIS IMPLEMENTATION

9.1 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.

9.2 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.

9.2.1 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.

9.2.1.1 Request For Becoming a SUC Node ID Server (SIS)

Enabling assignment and requesting the SIS capabilities is done in a similar manner as for the SUC. The capability parameter in **ZW_EnableSUC** and **ZW_SetSUCNodeID** is used to indicate that a SIS is wanted and thereby accept becoming a SIS in the network.

NOTE: There can only be one SIS in a network, but there can be many static controllers that are enabled for an assignment of the SIS capabilities in a network. Even if the SIS functionality is enabled for an assignment in the static controller then the primary controller can still choose only to activate the basic SUC functionality.

CONFIDENTIAL

9.2.2 Updates From The Primary Controller

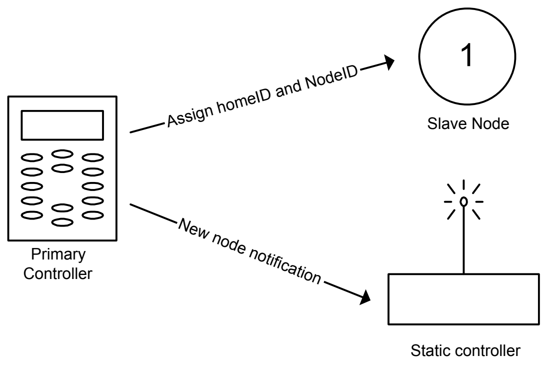


Figure 16 Inclusion of a node having a SUC in the network

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.

9.2.3 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.

9.2.4 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.

9.2.5 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.

9.3 The Primary Controller

The primary controller is responsible for choosing what static controller in the network that should act as a SUC and it will also send notifications to the SUC about all changes in the network topology. The application in a primary controller is responsible for choosing the static controller that should be the SUC. There is no fixed strategy for how to choose the static controller, so it is entirely up to the application to choose the controller that should become SUC. Once a static controller has been selected the

CONFIDENTIAL

application must use the **ZW_SetSUCNodeID** to request that the static controller becomes SUC. The capabilities parameter in the **ZW_SetSUCNodeID** call will determine if the primary controller enables the ID Server functionality in the SUC.

Once a SUC has been selected the protocol in the primary controller will automatically send notifications to the SUC about all changes in the network topology.

NOTE: A static controller can decline the role as SUC and in that case the callback function from **ZW_SetSUCNodeID** will return with a FAILED status. The static controller can also refuse to become SIS if that was what the primary controller requested, but accept to become a SUC.

9.4 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.

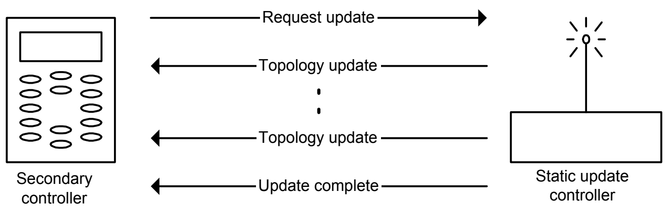


Figure 17 Requesting network updates from a SUC in the network

9.4.1 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.

9.4.2 Asking For And Receiving Updates

If the secondary controller knows the SUC it can ask for updates from the SUC. Asking for updates is done using the **ZW_RequestNetWorkUpdate** function. If the call was successful the update process will start and the controller application will be notified about any changes in the network through calls to **ApplicationControllerUpdate**). Once the update process is completed the callback function provided in **ZW_RequestNetWorkUpdate** will be called.

If the callback functions returns with the status ZW_SUC_UPDATE_OVERFLOW then it means that there has been more that 64 changes made to the network since the last update of this secondary controller and it is therefore necessary to do a controller replication to get this secondary controller updated.

NOTE: The SUC can refuse to update the secondary controller for several reasons, and if that happens the callback function will return with a value explaining why the update request was refused.

WARNING: Consider carefully how often the topology of the network changes and how important it is for the application that the secondary controller is updated with the latest.

CONFIDENTIAL

9.5 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 9.4 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.

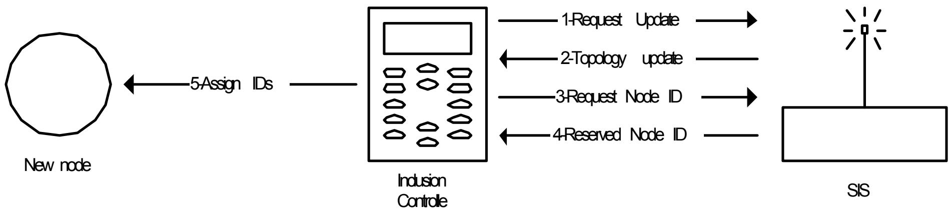


Figure 18 Inclusion of a node having a SIS in the network

9.6 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. Further a primary static controller can also help in case no SUC is present in the network. The lost slave initiates the re-discovery process because it will be the first to recognize that it is unable to reach the configured destinations and therefore can the application call **ZW_RediscoveryNeeded** to request help from other nodes in the network.

The lost battery operated routing slave start to send “I’m lost” frames to each node beginning with node ID = 1. It continue until it find a routing slave which can help it, i.e. the helping routing slave can obtain contact with a SUC or primary controller (require it’s static and mains powered). Scanning through the node ID’s is done on application level. Other strategies to send the “I’m lost” frame can be implemented on the application level.

CONFIDENTIAL

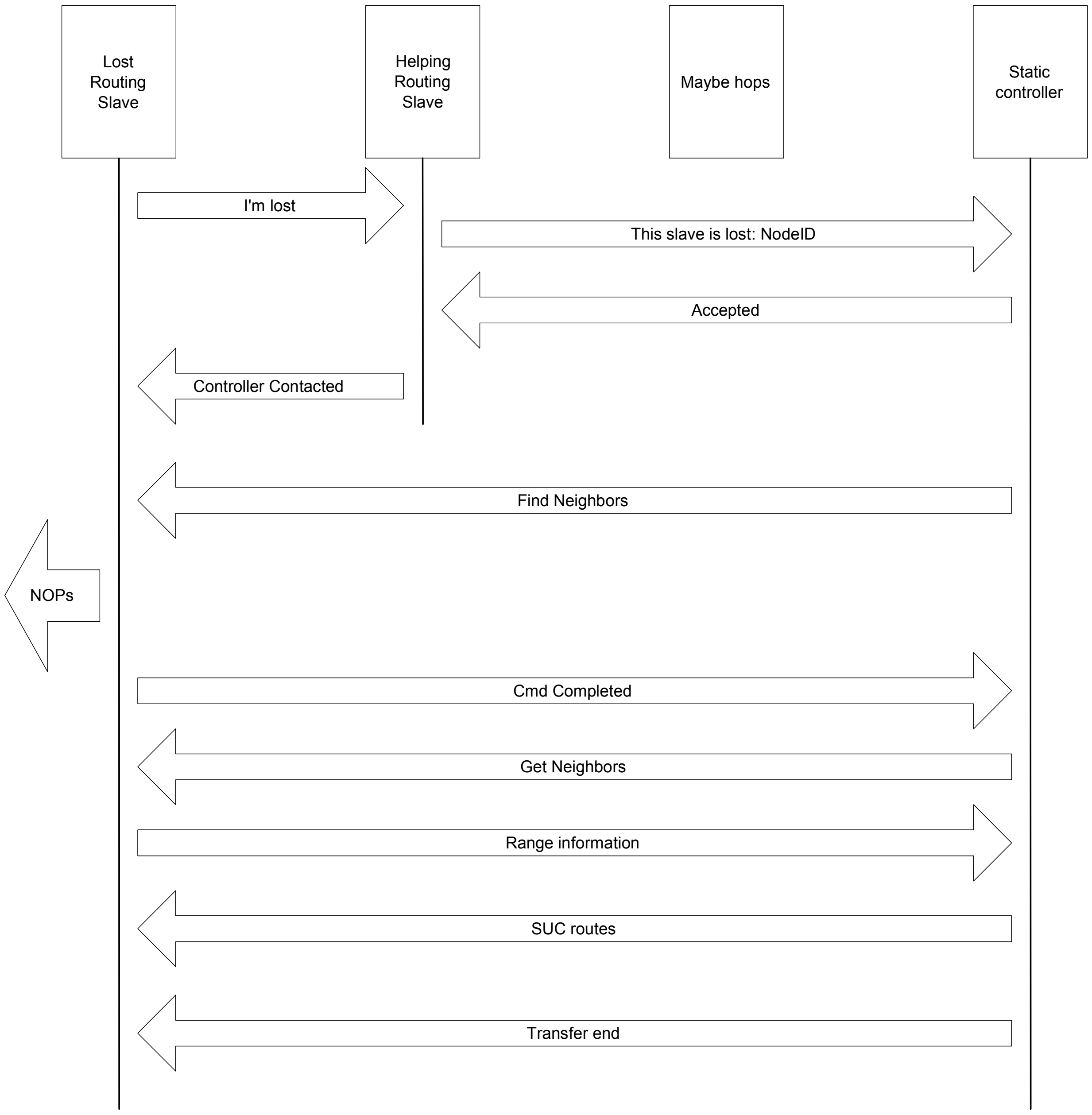


Figure 19 Lost routing slave frame flow

The helping routing slave must maximum use three hops to get to the controller, because it is the fourth hop when the controller issues the re-discovery to the lost routing slave. All handling in the helping slave is implemented on protocol level. In case a primary controller is found then it will check if a SUC exists in the network. In case a SUC is available it will be asked to execute the re-discovery procedure. When the controller receive the request “Re-discovery node ID x” it update the routing table with the new neighbor information. This allows the controller to execute a normal re-discovery procedure.

In case the **ZW_RediscoveryNeeded** was successful then the lost routing slave would get a callback with the status ZW_ROUTE_UPDATE_DONE and afterwards must the application call **ZW_RequestNetWorkUpdate** to obtain updated return routes from the SUC. See the Bin_Sensor_Battery sample code for an example of usage.

CONFIDENTIAL

10 APPLICATION NOTE: INCLUSION/EXCLUSION IMPLEMENTATION

This note describes the API calls the application layer needs to use when including new nodes to the network or excluding nodes from the network.

10.1 Including new nodes to the network

The API calls required by the including controller and the devices that is included are described. The callbacks as well as the steps the protocol takes without any application level involvement is also described. Finally it illustrates the frame flow between the two devices during the inclusion process.

The Z-Wave API calls **ZW_AddNodeToNetwork** and **ZW_SetLearnMode** are used to include nodes in a Z-Wave network. The primary/inclusion controller use the API call **ZW_AddNodeToNetwork** when including a node to the network and **ZW_SetLearnMode** is used by the controller or slave node that is to be included.

For the primary/inclusion controller that is including a node the **ZW_AddNodeToNetwork** is called with either:

ADD_NODE_ANY	Add any type of node to the network
ADD_NODE_SLAVE	Only add a node based on slave libraries
ADD_NODE_CONTROLLER	Only add a node based on controller libraries
ADD_NODE_EXISTING	Node is already in the network

To avoid the need to differentiate on the user interface whether it is a controller or slave the ADD_NODE_ANY can be used. The application can decide which actions to take based on the callback values.

ADD_NODE_SLAVE and ADD_NODE_CONTROLLER are available to support backward compatibility in case they are used on devices with separate slave and controller inclusion procedures.

ADD_NODE_EXISTING is useful when the controller application want the Node Information frame from a node already included in the network.

The figure below illustrates the inclusion process between a primary/inclusion controller and a node that the user wishes to include in the network.

CONFIDENTIAL

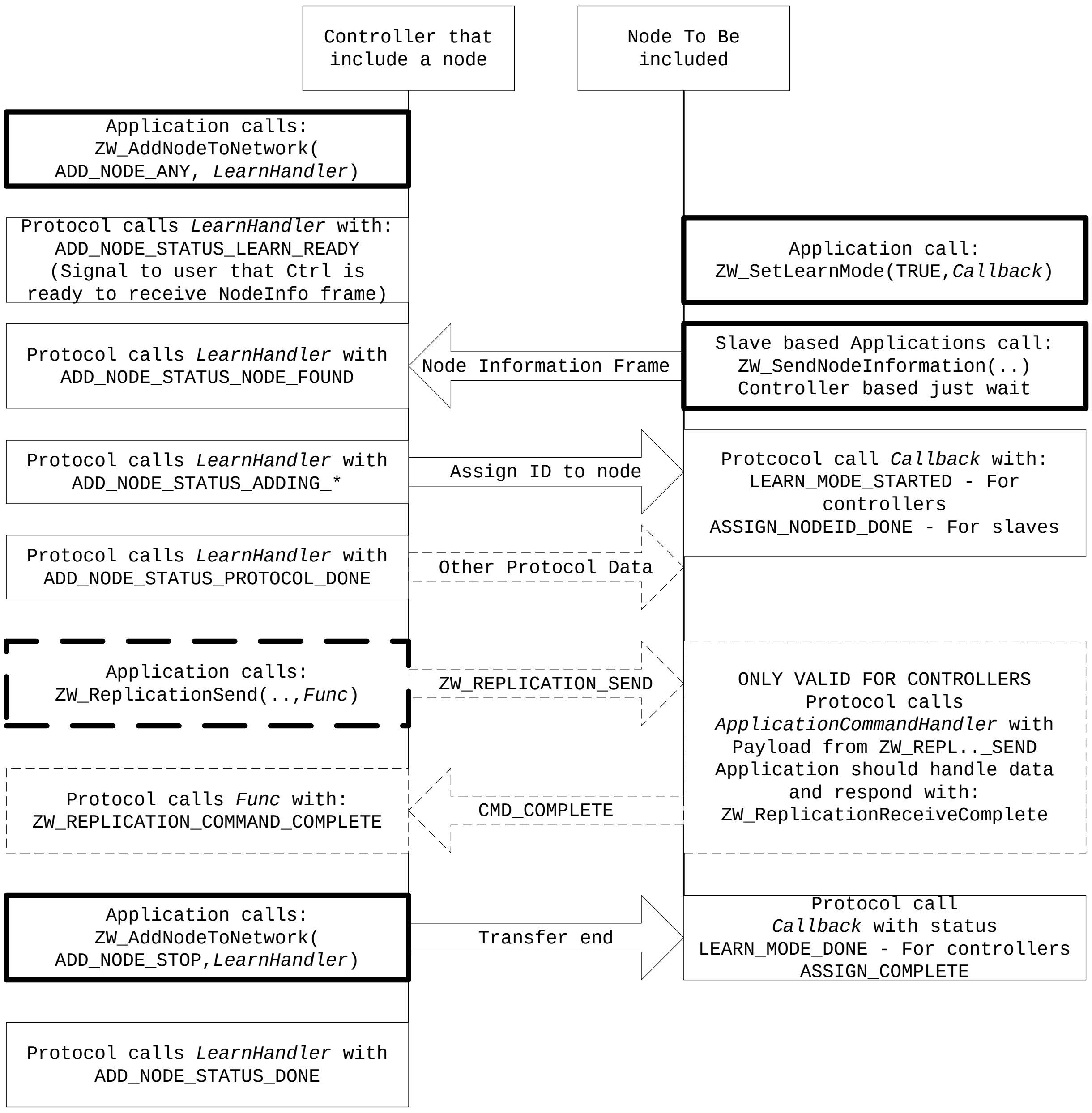

Controller that include a node
Node To Be included
Application calls: ZW_AddNodeToNetwork(ADD_NODE_ANY, LearnHandler)
Protocol calls LearnHandler with: ADD_NODE_STATUS_LEARN_READY (Signal to user that Ctrl is ready to receive NodeInfo frame)
Application call: ZW_SetLearnMode(TRUE,Callback)
Protocol calls 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

Figure 20 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

CONFIDENTIAL

can be included into a network. Furthermore the node must enter a learn mode state in order to accept assignment of new ID's. That state is communicated from the node by sending out a node information frame as described. The primary/inclusion controller can now assign a Home and Node ID to the node to be included in the Z-Wave network. In case the node is already included to a network then the primary/inclusion controller refuses to include it.

During "Other protocol data" the network topology is discovered and updated. The primary/inclusion controller request the new node to check which of the current nodes in the network it can communicate directly with. In case a SUC/SIS is present in the network then the new node is informed about its presence and SUC return routes are transferred automatically. In case the SUC/SIS is created at a later stage, then the API call **ZW_AssignSUCReturnRoutes** can be used to allow the node to communicate with the SUC/SIS.

In case a controller is included then it's optional to transfer groups and scenes on application level using the Controller Replication command class [1]. This option is very handy, as it will save the user a lot of time reconfiguring the groups and scenes in the new controller. The Controller Replication command class must only be used in conjunction with a controller shift or when including a new controller to the network. The API call **ZW_ReplicationSend** must be used by the sending controller when transferring the group and scene command classes to another controller. The API call **ZW_ReplicationReceiveComplete** must be used by the receiving controller as acknowledge on application level because the data must first be stored in 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**.

CONFIDENTIAL

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 stop the process with */
     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)
  {
     /* Add node is done. Application can move on Now is a good time to check if the
     added node should be set as SUC or SIS */
  }
}
```

CONFIDENTIAL

The following code samples show how an application typically implement the code needed in order to be able to include itself in an existing network.

Sample code for controller based devices:

```
/* Call to be performed when a controller wants to be include in the network */
ZW_SetLearnMode (TRUE, InclusionHandler);
/*Controller based devices just wait for the learn process to start*/

/*============================ InclusionHandler ============================
**                  Callback function to ZW_SetLearnMode
**------------------------------------------------------------------------*/
void InclusionHandler(
LEARN_INFO *learnNodeInfo)
{
  if ((*learnNodeInfo).bStatus == LEARN_MODE_STARTED)
  {
     /* The user should no longer be able to exit learn mode.
     ApplicationCommandHandler should be ready to handle ZW_REPLICATION_SEND_DATA
     frames if it supports transferring of Application specific data* /
  }
  else if ((*learnNodeInfo).bStatus == LEARN_MODE_FAILED)
  {
     /* Something went wrong - Signal to user */
  }
  else if ((*learnNodeInfo).bStatus == LEARN_MODE_DONE)
  {
     /* All data have been transmitted. Capabilities may have changed. Might be a
     good idea to read ZW_GET_CONTROLLER_CAPABILITIES() and to check that
     associations still are valid in order to check if the controller have been
     included or excluded from network*/
  }
}
```

CONFIDENTIAL

Sample code for slave based devices:

```
/* Call to be performed when a slave wants to be include in the network */
ZW_SetLearnMode(TRUE, InclusionHandler);
ZW_SendNodeInformation(NODE_BROADCAST, TRANSMIT_OPTION_LOW_POWER,....);

/*============================ InclusionHandler ============================
**                  Callback function to ZW_SetLearnMode
**-----------------------------------------------------------------------*/
void InclusionHandler
   BYTE bStatus /* IN Current status of Learnmode*/
   BYTE nodeID) /* IN resulting nodeID - If 0x00 the node was removed from network*/
{
  if(bStatus == ASSIGN_RANGE_INFO_UPDATE)
  {
     /* Application should make sure that it does not send out NodeInfo now that we
     are updating range */
  }
  if(bStatus == ASSIGN_COMPLETE)
  {
     /* Assignment was complete. Check if it was inclusion or exclusion and maybe
     tell user we are done */
     if (nodeID != 0)
     {
        /* Node was included in a network*/

     }
     else
     {
        /* Node was excluded from a network. Reset any associations */
     }

  }
  else if (bStatus == ASSIGN_NODEID_DONE)
  {
     /* ID is assigned. Protocol will call with bStatus=ASSIGN_COMPLETE when done */
  }
}
```

10.2 Excluding nodes from the network

The API calls required by the controller that exclude and the device that is to be excluded is described. The callbacks as well as the steps the protocol takes without any application level involvement is also described. Finally it illustrates the frame flow between the two devices during the exclusion process.

The Z-Wave API calls **ZW_RemoveNodeFromNetwork** and **ZW_SetLearnMode** are used to exclude nodes from a Z-Wave network. The primary/inclusion controller use the API call ZW_RemoveNodeFromNetwork when removing a node from a network and **ZW_SetLearnMode** is used by the controller or slave node that is to be removed.

For the primary/inclusion controller that is including a node the **ZW_RemoveNodeFromNetwork** is called with either:

REMOVE_NODE_ANY	- Remove any type of node from the network
REMOVE_NODE_SLAVE	- Only remove a node based on slave libraries
REMOVE_NODE_CONTROLLER	- Only remove a node based on controller libraries

CONFIDENTIAL

To avoid the need to differentiate on the user interface whether it is a controller or slave the REMOVE_NODE_ANY can be used. The application can decide which actions to take based on the callback values.

REMOVE_NODE_SLAVE and REMOVE_NODE_CONTROLLER are available to support backward compatibility in case they are used on devices with separate slave and controller exclusion procedures.

The figure below illustrates the exclusion process between a primary/inclusion controller and a node that the user wishes to exclude from the network.

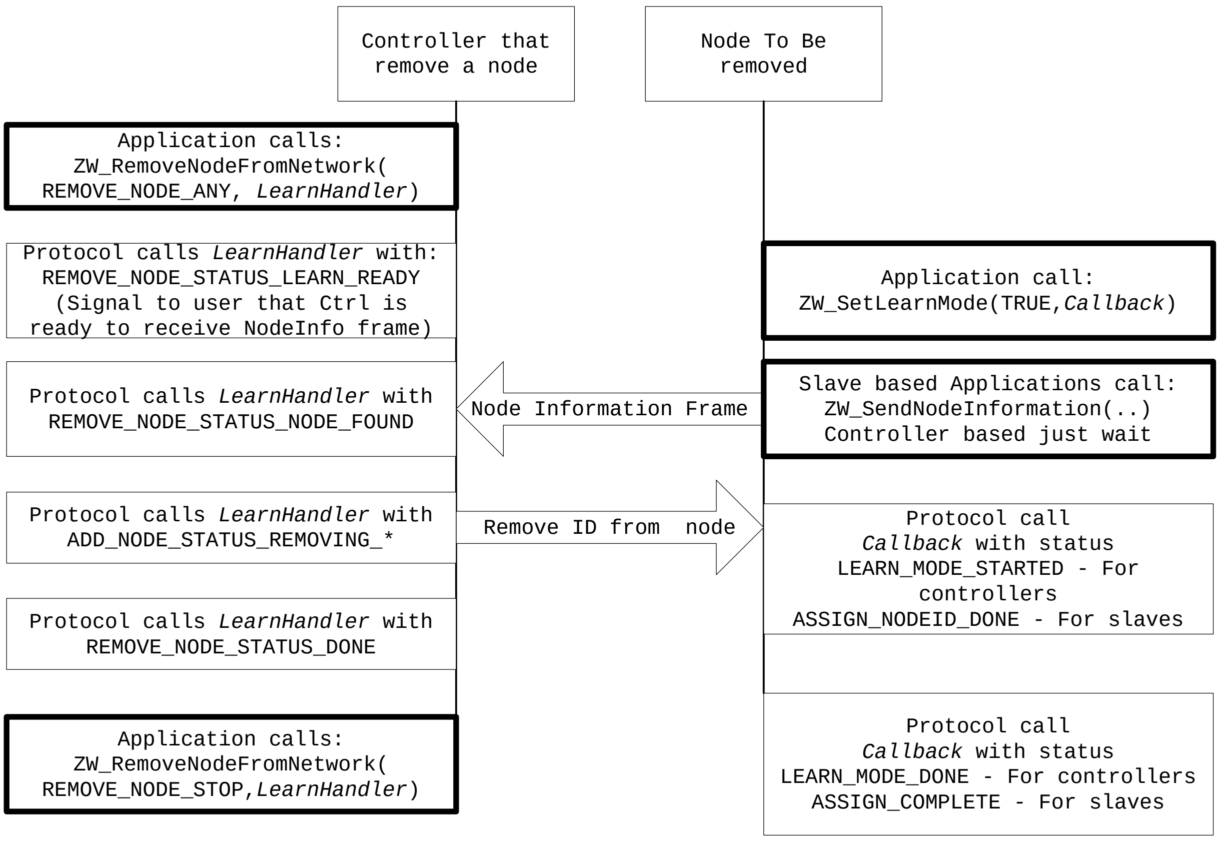

Controller that remove a node
Node To Be removed
Application calls: ZW_RemoveNodeFromNetwork(REMOVE_NODE_ANY, LearnHandler)
Protocol calls LearnHandler with: REMOVE_NODE_STATUS_LEARN_READY (Signal to user that Ctrl is ready to receive NodeInfo frame)
Application call: ZW_SetLearnMode(TRUE,Callback)
Protocol 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 21 Node exclusion frame flow

CONFIDENTIAL

The following code sample shows how remove node functionality is implemented on a controller capable of removing nodes from the network:

```
/* Call to be performed when user/application wants to remove a node from the network
*/
ZW_RemoveNodeFromNetwork(REMOVE_NODE_ANY, LearnHandler);

/*===========================  LearnHandler ==============================
**    Function description
**        Callback function to ZW_RemoveNodeFromNetwork
**------------------------------------------------------------------------*/
void LearnHandler(LEARN_INFO  *learnNodeInfo)
{
  if (learnNodeInfo->bStatus == REMOVE_NODE_STATUS_LEARN_READY)
  {
     /* Application should now signal to the user that we are ready to remove a node.
     User may still choose to abort */
  }
  else if (learnNodeInfo->bStatus == REMOVE_NODE_STATUS_NODE_FOUND)
  {
     /* Protocol is busy removing node. User interaction should be disabled */
  }
  else if (learnNodeInfo->bStatus == REMOVE_NODE_STATUS_REMOVING_SLAVE)
  {
     /* Protocol is still busy, this is just an information that it is a slave based
     unit that is being removed*/
  }
  else if (learnNodeInfo->bStatus == REMOVE_NODE_STATUS_REMOVING_CONTROLLER)
  {
     /* Protocol is still busy, this is just an information that it is a controller
     based unit that is being removed */
  }
  else if (learnNodeInfo->bStatus == REMOVE_NODE_STATUS_ DONE)
  {
     /* Node is no longer part of the network*/

     /* When done - stop the process with */
     ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP, NULL);
  }
  else if (learnNodeInfo->bStatus == ADD_NODE_STATUS_FAILED)
  {
     /* Remove node failed - Application should indicate this to user */
     ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP, NULL);
  }
}
```

For the device that is excluded, the process is no different from an inclusion See paragraph 10 for sample code.

Applications based on Controller libraries should most likely check which capabilities the application should enable once the learn process is over. This includes reading **ZW_GetControllerCapabilities**.

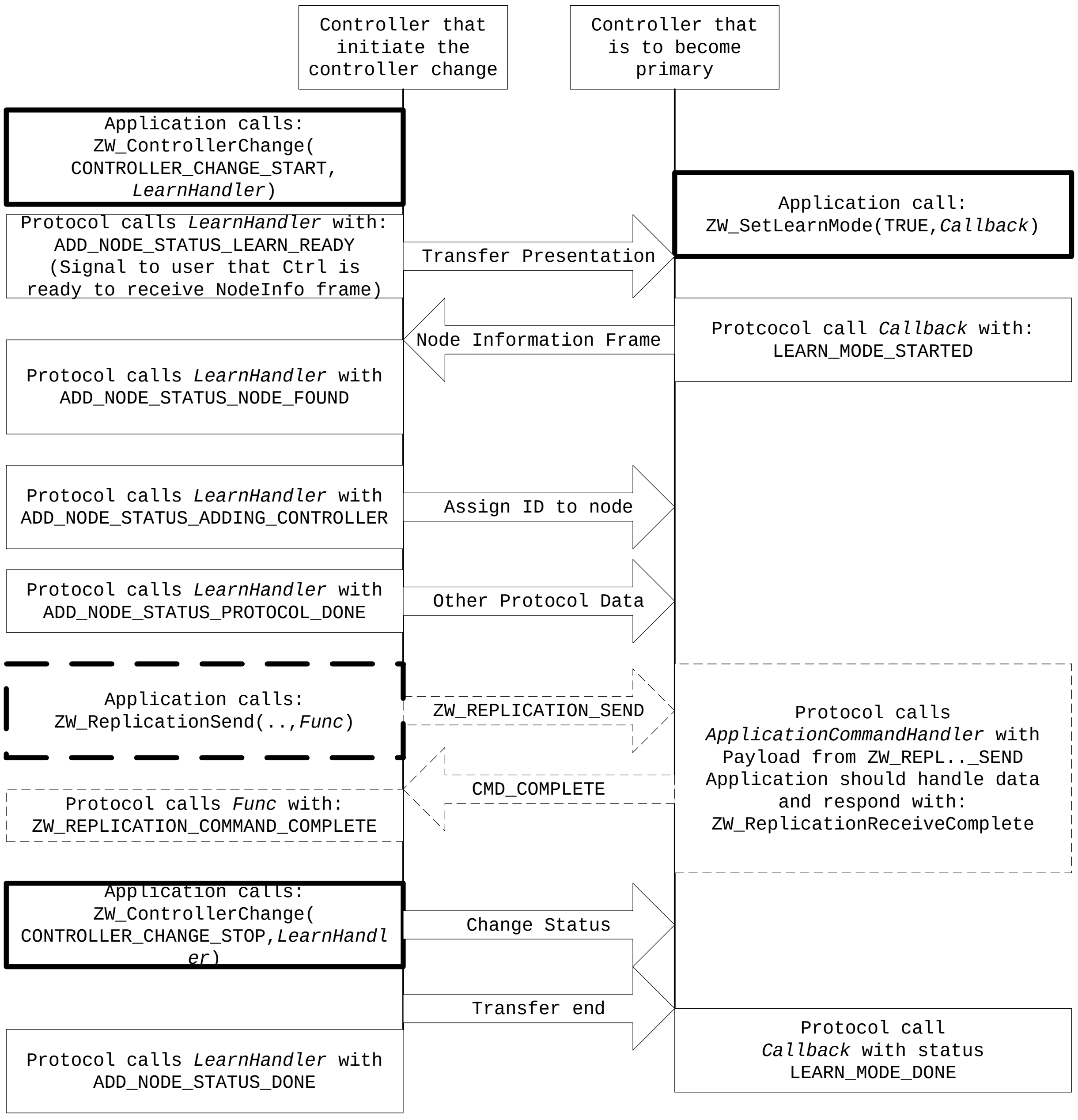
Applications based on slave libraries should check the node ID returned to the callback function during the learn process if this node ID is zero the device is being excluded from the network and the application should most likely remove its network specific settings, such as associations.

CONFIDENTIAL

11 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)
Protocol calls LearnHandler with: ADD_NODE_STATUS_LEARN_READY (Signal to user that Ctrl is ready to receive NodeInfo frame)
Transfer Presentation
Application call: ZW_SetLearnMode(TRUE,Callback)
Node Information Frame
Protcocol call Callback with: LEARN_MODE_STARTED
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 22 Controller shift frame flow

CONFIDENTIAL

12 APPLICATION NOTE: ZENSOR NET BINDING AND FLOODING

Zensor Net Binding is the process of associating a number of Zensor Net Routing Slaves.

It is possible to bind up to 16 nodes to each other; using one arbitrarily chosen node as “Bind master”. The bind master choses a random 16 bit Zensor Net ID. During bind, every other node receives the same Zensor Net ID and a unique Zensor node ID. The Zensor node ID is in the range 1..16.

When creating a system of networked smoke detectors, the Zensor node IDs will not be used. In case of fire, the first node to detect smoke issues a frame with the following properties

- Zensor beam
- Singlecast frame
- Destination address = broadcast
- Carrying a special Zensor Net flooding message

In all nodes receiving a Zensor Net flooding message, the following events take place:

1. Lock on preample; look for Start of Frame (SOF)

2. Look for Zensor start of Frame (ZOF)
   Step 1 is identical to classic Z-Wave.
   Nodes running firmware before Z-Wave v5.0 will interpret the ZOF as the first byte of the home ID.
   Home IDs having the ZOF value as the first byte are reserved. Thus, there is no risk that any legacy Z-Wave networks are influenced by Zensor Net traffic.
   Nodes running Z-Wave v5.0 or later will recognize the ZOF value and resume preamble tracking unless they are Zensor Net Routing Slaves. In this case, decoding goes on:

3. Look for header type = 4 (Zensor flooding)
   (Nodes running firmware before Z-Wave v5.0 would skip interpretation here if it wasn’t for the fact that they never come here because of the home ID mismatch)
   When carrying header type =4 content in a Z-Wave frame, the payload field carries a Zensor Net header extension before the actual payload

4. Look up Zensor Net ID and the Zensor node ID in the Zensor Net header extension.
   Filtering accepts the frame if the Zensor node ID equals the actuals node’s ID or the broadcast address

5. Look up the loop token.
   If the token is not in the recent list of tokens, the token is added to the list, the frame is forwarded and a copy is delivered to the application.
   If the token IS in the recent list of tokens, the frame is discarded.

CONFIDENTIAL

13 REFERENCES

[1] Zensys, SDS10242, Software Design Specification, Z-Wave Device Class Specification
[2] Zensys, DSH10086, Datasheet, ZW0x0x Z-Wave Interface Module
[3] Zensys, DSH10087, Datasheet, ZW0x0x Z-Wave Development Module
[4] Zensys, DSH10033, Datasheet, ZM1220 Z-Wave Module
[5] Zensys, DSH10034, Datasheet, ZM1206 Z-Wave Module
[6] Zensys, INS10240, Instruction, PC Based Controller User Guide
[7] Zensys, INS10241, Instruction, PC Installer Tool Application User Guide
[8] Zensys, INS10245, Instruction, Z-Wave Bridge User Guide
[9] Zensys, INS10029, Instruction, ZW0102 Single Chip Implementation Guideline
[10] Zensys, APL10312, Application Note, Programming the 200 and 300 Series Z-Wave Single Chip Flash
[11] Zensys, INS10336, Instruction, Z-Wave Reliability Test Guideline
[12] Zensys, INS10249, Instruction, Z-Wave Zniffer User Guide
[13] Zensys, INS10250, Instruction, Z-Wave DLL User’s Manual
[14] Zensys, INS10679, Instruction, Z-Wave Programmer User Guide
[15] Zensys, INS10236, Instruction,ZW0102 Development Controller sample application
[16] Zensys, INS10579, Instruction,Programming the ZW0102 Flash and Lock Bits
[17] Zensys, DSH10088, Datasheet ZMxx06 Converter Module
[18] Zensys, DSH10230, Datasheet, ZM2106C Z-Wave Module
[19] Zensys, INS10326, Instruction, ZW0201 Single Chip Implementation Guidelines
[20] Zensys, SRN10793, ZW0102/ZW0201/ZW0301 Developer’s Kit v5.00
[21] Zensys, APL10512, Application Note, Battery Operated Applications Using the ZW0201/ZW0301
[22] Zensys, DSH10856, Datasheet, ZM3106C Z-Wave Module
[23] Zensys, DSH10275, Datasheet, ZM2120C Z-Wave Module
[24] Zensys, DSH10857, Datasheet, ZM3120C Z-Wave Module
[25] Zensys, APL10292, Application Note, ZW0102 Triac Controller Guideline
[26] Zensys, APL10370, Application Note, ZW0201/ZW0301 Triac Controller Guideline
[27] Zensys, APL10514, Application Note, The ZW0201/ZW0301 ADC

CONFIDENTIAL

INDEX

A

ADC_Buf (ZW0201/ZW0301 only) 123
ADC_GetRes 129
ADC_GetSamplingRate (ZW0102 only) 128
ADC_Init 118
ADC_Int 127
ADC_IntFlagClr 127
ADC_Off 116
ADC_On (ZW0102 only) 116
ADC_SelectPin 122
ADC_SetAZPL (ZW0201/ZW0301 only) 124
ADC_SetResolution (ZW0201/ZW0301 only) 124
ADC_SetThres 126
ADC_SetThresMode (ZW0201/ZW0301 only) 125
ADC_Start 116
ADC_Stop 117
App_RFSetup.a51 .... 206, 207
ApplicationCommandHandler 60
ApplicationControllerUpdate 47, 73, 179, 258, 259
ApplicationControllerUpdate (All controller libraries) 65
ApplicationInitHW 56
ApplicationInitSW 57
ApplicationNodeInformation 61
ApplicationPoll 59
ApplicationRfNotify (ZW0301 only) 68
ApplicationSlaveCommandHandler (Bridge Controller library only) 66
ApplicationSlaveNodeInformation (Bridge Controller library only) 67
ApplicationSlaveUpdate 199
ApplicationSlaveUpdate (All slave libraries) 64
ApplicationTestPoll 58
Assembler 252

C

C compiler 252
ClockCmp 142
ClockGet 142
ClockSet 141
Command prompt 212

D

Doload.bat file 23
DOS box 212

E

Eeploader.exe 256
Enhanced Slave 177
Equinox Epsilon5 programmer 22
Extern_eep.hex file 256
External EEPROM 37, 256

CONFIDENTIAL

F

FCC compliance test 206, 207
FLASH_APPL_FREQ_OFFS 206, 207
FLASH_APPL_LOW_POWER_OFFS 206, 207
FLASH_APPL_MAGIC_VALUE_OFFS........ 206, 207
FLASH_APPL_NORM_POWER_OFFS........ 206, 207
FLASH_APPL_PLL_STEPUP_OFFS 68, 207
FLASH_APPL_RX_MATCH_OFFS 206
FLASH_APPL_TX_MATCH_OFFS........ 206
FUNC_ID_SERIAL_API_GET_CAPABILITIES........ 243
FUNC_ID_SERIAL_API_SET_TIMEOUTS........ 243
FUNC_ID_SERIAL_API_SOFT_RESET........ 244

I

I/O pins 208
Inclusion controller........ 46, 260
Interrupt 36
Interrupt service routines 36

K

Keil........ 252
KEIL_LOCAL_PATH 212
KEILPATH........ 212

L

Librarian........ 252
Linker 252
Listening flag........ 61
Lock bits........ 255

M

Make files........ 211
Memory optimization........ 53
MemoryGetBuffer 113
MemoryGetByte........ 111
MemoryGetID 110
MemoryPutBuffer........ 114
MemoryPutByte 112
MK.BAT 211

N

Node Information Frame........ 61, 158
NON_ZERO_SIZE........ 32
NON_ZERO_START_ADDR........ 32

O

Optional Functionality flag 61

P

PIN_GET 209
PIN_IN 208
PIN_OFF........ 209
PIN_ON 209

CONFIDENTIAL

PIN_OUT 208
PIN_TOGGLE 210
Power on reset 78, 79
Primary controller 39, 46, 47, 257
Production test 10, 11, 58, 229
PVT and RF regulatory measurements 26
PWR_Clk_PD (ZW0102 only) 131
PWR_Clk_PUp (ZW0102 only) 131
PWR_Select_Clk (ZW0102 only) 132
PWR_SetStopMode (ZW0102 only) 130

R

Response route 99
Return route 91, 95
RF frequency 206, 207
RF match capacity array values 206
RF settings 206
RF transmit power levels 206, 207
Routing slave 260
Routing Slave 177
RTCTimerCreate 143
RTCTimerDelete 145
RTCTimerRead 144
RXmatch 206

S

SerialAPI_ApplicationNodeInformation 62
SerialAPI_ApplicationSlaveNodeInformation 67
SIS 52, 73, 260
Sleep mode 78, 130, 140, 141
Static update controller 45, 47, 51, 257
Stop mode 80, 130
SUC 51, 73
SUC ID Server 46, 52
SUC/SIS node 177

T

Timer 0 36
Timer 1 36
Timer 2 36
Timer 3 36
TimerCancel 105
TimerRestart 105
TimerStart 104
TOOLSDIR 212
TRIAC_Init 101
TRIAC_Off 103
TRIAC_SetDimLevel 103
TXmatch 206
TXnormal Power 206, 207

U

UART_ClearRx (ZW0201/ZW0301 only) 138
UART_ClearTx (ZW0201/ZW0301 only) 137
UART_Disable (ZW0201/ZW0301 only) 137

CONFIDENTIAL

UART_Enable (ZW0201/ZW0301 only) 137
UART_Init 133
UART_Read (ZW0201/ZW0301 only) 139
UART_RecByte 134
UART_RecStatus 134
UART_SendByte 135
UART_SendNL 136
UART_SendNum 136
UART_SendStatus 135
UART_SendStr 136
UART_Write (ZW0201/ZW0301 only) 138
uninitialized RAM bytes 222

W

Watchdog 88
Wut fast mode 80
Wut mode 80

Z

ZM1206 module 22, 229, 230
ZM1220 module 229
ZW_ADC_BUFFER_DISABLE (Macro) 123
ZW_ADC_BUFFER_ENABLE (Macro) 123
ZW_ADC_CLR_FLAG (Macro) 127
ZW_ADC_GET_READING (Macro) 129
ZW_ADC_INIT (Macro) 118
ZW_ADC_INT_DISABLE (Macro) 127
ZW_ADC_INT_ENABLE (Macro) 127
ZW_ADC_OFF (Macro) 116
ZW_ADC_ON (Macro) 116
ZW_ADC_RESOLUTION_12 (Macro) 124
ZW_ADC_RESOLUTION_8 (Macro) 124
ZW_ADC_SAMPLE_RATE (Macro) 128
ZW_ADC_SELECT_AD1 (Macro) 122
ZW_ADC_SELECT_AD2 (Macro) 122
ZW_ADC_SELECT_AD3 (Macro) 122
ZW_ADC_SELECT_AD4 (Macro) 122
ZW_ADC_SET_AZPL (Macro) 124
ZW_ADC_SET_THRESHOLD (Macro) 126
ZW_ADC_START (Macro) 116
ZW_ADC_STOP (Macro) 117
ZW_ADC_THRESHOLD_LO (Macro) 125
ZW_ADC_THRESHOLD_UP (Macro) 125
ZW_ADD_NODE_TO_NETWORK (Macro) 149
ZW_AddNodeToNetwork 149, 262
ZW_ARE_NODES_NEIGHBOURS(Macro) 170
ZW_AreNodesNeighbours 170
ZW_ASSIGN_RETURN_ROUTE (Macro) 162
ZW_ASSIGN_SUC_RETURN_ROUTE (Macro) 177
ZW_AssignReturnRoute 162
ZW_AssignSUCReturnRoute 177
ZW_CLOCK_CMP (Macro) 142
ZW_CLOCK_GET (Macro) 142
ZW_CLOCK_SET (Macro) 141
ZW_CONTROLLER_CHANGE (Macro) 153
ZW_ControllerChange 153, 270

CONFIDENTIAL

ZW_CREATE_NEW_PRIMARY_CTRL (Macro) 181
ZW_CreateNewPrimaryCtrl 181
ZW_DEBUG_CMD_INIT (Macro) 205
ZW_DEBUG_CMD_POLL (Macro) 205
ZW_DEBUG_INIT 139
ZW_DEBUG_SEND_BYTE 140
ZW_DEBUG_SEND_NUM 140
ZW_DebugInit 205
ZW_DebugPoll 205
ZW_DELETE_RETURN_ROUTE (Macro) 163
ZW_DELETE_SUC_RETURN_ROUTE (Macro) 178
ZW_DeleteReturnRoute 163
ZW_DeleteSUCReturnRoute 178
ZW_EEPROM_INIT (Macro) 115
ZW_EepromInit 115
ZW_ENABLE_SUC (Macro) 180
ZW_EnableSUC 173, 180
ZW_GET_CONTROLLER_CAPABILITIES (Macro) 157
ZW_GET_NEIGHBOR_COUNT (Macro) 169
ZW_GET_NODE_STATE (Macro) 158
ZW_GET_PROTOCOL_STATUS (Macro) 87
ZW_GET_ROUTING_INFO (Macro) 172
ZW_GET_SUC_NODE_ID (Macro) 173
ZW_GET_VIRTUAL_NODES (Macro) 190
ZW_GetControllerCapabilities 157
ZW_GetNeighborCount 169
ZW_GetNodeProtocolInfo 158
ZW_GetProtocolStatus 87
ZW_GetRoutingInfo 172
ZW_GetSUCNodeID 173
ZW_GetVirtualNodes 190
ZW_IS_FAILED_NODE_ID (Macro) 165
ZW_IS_NODE_WITHIN_DIRECT_RANGE (Macro) 198
ZW_IS_VIRTUAL_NODE (Macro) 189
ZW_IS_WUT_KICKED (Macro) 133
ZW_isFailedNode 165
ZW_IsNodeWithinDirectRange 198
ZW_IsPrimaryCtrl 166
ZW_IsVirtualNode 189
ZW_LOCK_RESPONSE_ROUTE (Macro) 99
ZW_LockRoute 99
ZW_MEM_FLUSH (Macro) 115
ZW_MEM_GET_BUFFER (Macro) 113
ZW_MEM_GET_BYTE (Macro) 111
ZW_MEM_PUT_BUFFER (Macro) 114
ZW_MEM_PUT_BYTE (Macro) 112
ZW_MEMORY_GET_ID (Macro) 110
ZW_MemoryFlush 115
ZW_NODE_MASK_BITS_IN (Macro) 147
ZW_NODE_MASK_CLEAR (Macro) 147
ZW_NODE_MASK_CLEAR_BIT (Macro) 146
ZW_NODE_MASK_NODE_IN (Macro) 148
ZW_NODE_MASK_SET_BIT (Macro) 146
ZW_NodeMaskBitsIn 147
ZW_NodeMaskClear 147
ZW_NodeMaskClearBit 146
ZW_NodeMaskNodeIn 148

CONFIDENTIAL

ZW_NodeMaskSetBit 146
ZW_Poll 69
ZW_POLL (Macro)........ 69
ZW_PRIMARYCTRL (Macro)........ 166
ZW_PWM_CLEAR_INTERRUPT (Macro) 109
ZW_PWM_INT_ENABLE (Macro)........ 109
ZW_PWM_PRESCALE (Macro)........ 108
ZW_PWM_SETUP (Macro)........ 106
ZW_PWMClearInterrupt 109
ZW_PWMEnable 109
ZW_PWMPrescale 108
ZW_PWMSetup 106
ZW_PWR_CLK_POWERDOWN (Macro) 131
ZW_PWR_CLK_POWERUP (Macro)........ 131
ZW_PWR_SELECT_CLK (Macro) 132
ZW_PWR_SET_STOP_MODE (Macro)........ 130
ZW_Random........ 69
ZW_RANDOM (Macro)........ 69
ZW_REDISCOVERY_NEEDED (Macro) 72
ZW_RediscoveryNeeded........ 162, 177, 260
ZW_RediscoveryNeeded (Not Slave and Bridge Controller Library) 72
ZW_REMOVE_FAILED_NODE_ID (Macro) 164
ZW_REMOVE_NODE_FROM_NETWORK (Macro) 151
ZW_RemoveFailedNodeID........ 164
ZW_RemoveNodeFromNetwork 151, 267
ZW_REPLACE_FAILED_NODE (Macro)........ 166
ZW_ReplaceFailedNode 166
ZW_REPLICATION_COMMAND_COMPLETE (Macro)........ 161
ZW_REPLICATION_SEND_DATA (Macro)........ 160
ZW_ReplicationReceiveComplete........ 161
ZW_ReplicationSend........ 160
ZW_REQUEST_NETWORK_UPDATE (Macro) 73
ZW_REQUEST_NEW_ROUTE_DESTINATIONS (Macro) 197
ZW_REQUEST_NODE_INFO (Macro) 179, 199
ZW_REQUEST_NODE_NEIGHBOR_UPDATE (Macro)........ 171
ZW_RequestNetWorkUpdate 65, 177, 180, 259, 260
ZW_RequestNetWorkUpdate (Not Slave Library)........ 73
ZW_RequestNewRouteDestinations........ 197
ZW_RequestNodeInfo 179, 199, 263
ZW_RequestNodeNeighborUpdate........ 171
ZW_RF_POWERLEVEL_GET (Macro)........ 71
ZW_RF_POWERLEVEL_REDISCOVERY_SET (Macro)........ 75
ZW_RF_POWERLEVEL_SET (Macro) 70
ZW_RFPowerLevelGet........ 71
ZW_RFPowerlevelRediscoverySet 75
ZW_RFPowerLevelSet 70
ZW_RTC_CREATE (Macro)........ 143
ZW_RTC_DELETE (Macro) 145
ZW_RTC_READ (Macro) 144
ZW_SEND_CONST (Macro) 100
ZW_SEND_DATA (Macro) 91
ZW_SEND_DATA_ABORT (Macro)........ 99
ZW_SEND_DATA_META (Macro) 95
ZW_SEND_DATA_MULTI (Macro) 97
ZW_SEND_NODE_INFO (Macro)........ 81
ZW_SEND_SLAVE_NODE_INFO (Macro) 185
ZW_SEND_SUC_ID (Macro) 176

CONFIDENTIAL

ZW_SEND_TEST_FRAME (Macro) 82
ZW_SendConst 100
ZW_SendData 91
ZW_SendDataAbort 99
ZW_SendDataMeta 95
ZW_SendDataMulti 97
ZW_SendNodeInformation 81
ZW_SendSlaveData 183
ZW_SendSlaveNodeInformation 185
ZW_SendSUCID 176
ZW_SendTestFrame 82
ZW_SET_DEFAULT (Macro) 159, 195
ZW_SET_EXT_INT_LEVEL (Macro) 90
ZW_SET_LEARN_MODE (Macro) 155, 194
ZW_SET_PROMISCUOUS_MODE (Macro) 76
ZW_SET_RX_MODE (Macro) 77
ZW_SET_SLAVE_LEARN_MODE (Macro) 186
ZW_SET_SLEEP_MODE (Macro) 79
ZW_SET_SUC_NODE_ID (Macro) 173
ZW_SET_WUT_TIMEOUT (Macro) 132
ZW_SetDefault 159, 195
ZW_SetExtIntLevel (ZW0201/ZW0301 only) 90
ZW_SetLearnMode 155, 194, 262, 267
ZW_SetPromiscuousMode (Not Bridge Controller library) 76
ZW_SetRFReceiveMode 77
ZW_SetSlaveLearnMode 186
ZW_SetSleepMode 78
ZW_SetSUCNodeID 173, 180
ZW_SetWutTimeout (ZW0201/ZW0301 only) 132, 133
ZW_SLAVE_SEND_DATA (Macro) 183
ZW_SLEEP (Macro) 78
ZW_STORE_HOME_ID (Macro) 193
ZW_STORE_NODE_INFO (Macro) 192
ZW_StoreHomeID 193
ZW_StoreNodeInfo 192
ZW_SUPPORT9600_ONLY(Macro) 196
ZW_Support9600Only (ZW0201/ZW0301 only) 196
ZW_TIMER_CANCEL (Macro) 105
ZW_TIMER_RESTART (Macro) 105
ZW_TIMER_START (Macro) 104
ZW_TRIAC_DIM_SET_LEVEL (Macro) 103
ZW_TRIAC_INIT (Macro) 101, 102
ZW_TRIAC_INIT_2_WIRE (Macro) 101, 102
ZW_TRIAC_LIGHT_SET_LEVEL (Macro) 103
ZW_TRIAC_OFF (Macro) 103
ZW_TX_COUNTER (Macro) 191
ZW_Type_Library 84
ZW_TYPE_LIBRARY (Macro) 84
ZW_UART_CLEAR_RX (Macro) 138
ZW_UART_CLEAR_TX (Macro) 137
ZW_UART_DISABLE (Macro) 137
ZW_UART_ENABLE (Macro) 137
ZW_UART_INIT (Macro) 133
ZW_UART_READ_RX (Macro) 139
ZW_UART_REC_BYTE (Macro) 134
ZW_UART_REC_STATUS (Macro) 134
ZW_UART_SEND_BYTE (Macro) 135

CONFIDENTIAL

ZW_UART_SEND_NL (Macro) 136
ZW_UART_SEND_NUM (Macro).......... 136
ZW_UART_SEND_STATUS (Macro).......... 135
ZW_UART_SEND_STRING (Macro) 136
ZW_UART_WRITE_TX (Macro).......... 138
ZW_Version 85
ZW_VERSION (Macro) 85
ZW_VERSION_BETA (Macro) 86
ZW_VERSION_MAJOR (Macro).......... 86
ZW_VERSION_MAJOR / ZW_VERSION_MINOR / ZW_VERSION_BETA.......... 86
ZW_VERSION_MINOR (Macro) 86
ZW_WATCHDOG_DISABLE (Macro).......... 88
ZW_WATCHDOG_ENABLE (Macro) 88
ZW_WATCHDOG_KICK (Macro).......... 89
ZW_WatchdogDisable 88
ZW_WatchdogEnable.......... 88
ZW_WatchdogKick 89
ZW_ZensorGetID.......... 202
ZW_ZensorNetBind 200
ZW_ZensorSendDataFlood 201
ZW_ZensorSetDefault 202
ZW0102 Controller/Slave Unit 253
ZW0102 Development Controller Unit.......... 253
ZW0201 Controller/Slave Unit 254
ZW0201 Development Controller Unit.......... 254
ZW0301 Controller/Slave Unit 254
ZW0301 Development Controller Unit.......... 254
Z-Wave Programmer 22, 252
zwTransmitCount.......... 191

CONFIDENTIAL