



Instruction

Z-Wave ZW0201/ZW0301 Appl. Prg. Guide v4.53.00

| | |
|----------------------|--|
| Document No.: | INS11095 |
| Version: | 9 |
| Description: | Guideline for developing ZW0201 and ZW0301 based applications using the application programming interface (API) based on Developer's Kit v4.5x |
| Written By: | JFR;JSI;PSH;DDA |
| Date: | 2011-11-08 |
| Reviewed By: | JFR;JSI;CHL;CST;PSH;JKA |
| Restrictions: | Partners Only |

Approved by:

| | | | | |
|------------|----------|----------|---------------|------------------|
| Date | CET | Initials | Name | Justification |
| 2011-11-08 | 17:26:21 | JFR | Jørgen Franck | on behalf of NTJ |

This document is the property of Sigma Designs Inc. The data contained herein, in whole or in part, may not be duplicated, used or disclosed outside the recipient for any purpose. 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 | 20080315 | JFR | 3.4.7 | PVT & RF regulatory files supports now also 40 kbps |
| 1 | 20080315 | JFR | 5.3.3.1 | Added missing description of hop failures in TxStatus in ZW_SendData API call |
| 1 | 20080319 | JFR | 3.4.7 | Added PVT & RF regulatory files using 40kbps |
| 1 | 20080417 | JFR | 3 | Updated directory structure and file locations/descriptions |
| 1 | 20080519 | JBU | 5.9.1 | Names of ZW_RequestNewRouteDestinations() callback function parameters corrected |
| 1 | 20080519 | JBU | 5.4.1 5.4.18 | Clarified ZW_AddNodeToNetwork() and ZW_RemoveNodeFromNetwork () should use NULL callback when * _NODE_STOP commands are given. |
| 1 | 20080519 | JBU | 11.1 11.2 | Updated figures and sample code to use recommended ZW_AddNodeToNetwork()/ZW_RemoveNodeFromNetwork () _STOP commands. |
| 1 | 20080624 | JSI | 5.4.1 | Added description of ADD_NODE_OPTION_NETWORK_WIDE option bit |
| 1 | 20080825 | JSI | 5.3.3 | Added ZW_SendData_Generic, ZW_SendData_Bridge, ZW_SendDataMeta_Generic and ZW_SendDataMeta_Bridge descriptions |
| 1 | 20080925 | JBU | 5.1.1.2 5.1.1.3 | Static controller libraries without repeater functionality cannot provide help to or forward Lost requests. |
| 1 | 20080929 | JBU | 5.3.1.5 | Added promiscuous mode and destNode description to function syntax and SerialAPI description. |
| 1 | 20080929 | JBU | 5.3.2.12 | Added reference to sec 5.3.1.5. |
| 1 | 20081126 | JFR | All 3 & 7 3, 4 & 5 3 & 7 3 & 7 5 3.4.7 | Added MY Frequency. Removed doorbell and smoke sensor. Removed slave sensor library. Updated Bridge and Static Controller library/applications with respect to SUC/SIS and repeater support. Extended serial API to support slave, enhanced slave and routing slave. Remove ZW_lockRoute. ANZ, HK and MY support added to PVT and RF regulatory hex files. |
| 1 | 20081201 | JSI | 5.3.1.5 5.3.1.9 | Bridge Controller do not use the ApplicationCommandHandler The ApplicationSlaveCommandHandler and the ApplicationCommandHandler in the Bridge Controller has been unified in the ApplicationCommandHandler_Bridge with full Multicast support |
| 1 | 20081201 | PSH | 5.4.24 5.8.1 | Changed mode parameter for ZW_SetLearnMode() |
| 1 | 20081201 | PSH | 7.4 7.8.3 | Added short description of network wide inclusion at startup in sample application description. |
| 1 | 20081202 | SAR JFR | 3.1, 3.5, 7.12.3, 7.12.4 & 7.12.5 | PC applications and libraries updated |
| 1 | 20081204 | JSI | 5.3.3 5.3.3.7 5.6 | Updated parameter descriptions and SerialAPI definitions. Added ZW_SendDataMulti_Bridge descriptions. Removed ZW_SendSlaveData descriptions. |
| 2 | 20081216 | JFR | 5.3.3.1 | Clarified transmit options behavior |
| 2 | 20090114 | JFR | 5.3.2.19 | Watchdog disabled by default |
| 2 | 20090114 | PSH | 5.3.1 5.3.2.1 | Limitations regarding application execution ZW_Poll description updated |
| 2 | 20090119 | PSH | 7 | Added description og triple press and network wide inclusion implementation in sample applications |
| 2 | 20090130 | JSI | 5.3.2.10 | Updated serialapi flow description |
| 2 | 20090201 | JFR | 5.4.27 3 & 7 5.9.3 5.4.27 & 5.4.12 | Do not recommend to call ZW_SetSUCNodeID using low RF power Doorbel application added A non-SUC primary static controller do not respond to a rediscovery needed when there is no SUC present in the network ZW_SetRoutingMAX and ZW_GetRoutingMAX added |
| 2 | 20090219 | JFR | 5.3.4.1 & 5.3.4.2 5.3.6 | TRIAC_Init and TRIAC_SetDimLevel description updated. PWM API updated. |
| 2 | 20090223 | JBU | 5.3.3.1 | Discouraged sending to a virtual node from the hosting bridge library. |
| 2 | 20090225 | JSI | 5.4.23 | Added ZW_SetRoutingInfo description. |
| 2 | 20090304 | JSI | 5.3.2.1, 5.3.2.5, 5.3.2.9, 5.8.3, 5.9.1 & 5.9.4 | Added SerialAPI descriptions. |
| 2 | 20090305 | JFR | 5.3.2.9 | Using normal power minus 6dB when doing neighbor search |

REVISION RECORD

| Doc. Rev | Date | By | Pages affected | Brief description of changes |
|----------|----------|------------|---|--|
| 2 | 20090310 | JFR | 5.3.3 | Payload length must be minimum one byte. |
| 2 | 20090318 | JFR | 3.3 & 7 | Security applications added. |
| 3 | 20090529 | JFR | 7 | Sample application behavior due to Node Information Frame clarified |
| 3 | 20090603 | JFR | 3.3.1.19 - 3.3.1.26 | Support of ZW_SET_WUT_TIMEOUT enabled in Slave Enhanced and Slave Routing based serial API targets. |
| 4 | 20090928 | JFR JSI | 3, 4.8, 5 & 7 3 & 7 3.1 & 3.2.4.8 3.1 & 3.2.4.14 7.12.2.7 5.4.27 & 5.4.12 5.4.17 5 3 & 4 3 4 5.4.10 8 | Discontinued slave library. India (IN) frequency added for both applications and tools. Added ZW_slave_enhanced_noflirs_nomr_ZW020xs.lib Added ZW_slave_routing_noflirs_nomr_ZW020xs.lib Serial API watchdog handling added. Serial API impl. of ZW_SetRoutingMAX/ZW_GetRoutingMAX. Added normal/low RF transmission parm. to ZW_ReplaceFailedNode. TRANSMIT_OPTION_RETURN_ROUTE replaced with TRANSMIT_OPTION_AUTO_ROUTE. Enhanced 232 slave library added. New serial API based slave applications added. Details about routing algorithm added to library descriptions. Sensor capabilities documented in Node Information Frame when calling ZW_GetProtocolInfo Tool sample code added |
| 5 | 20091229 | VVI | 8.1.1 | Added description of Z-Wave programmer firmware source files |
| 5 | 20100104 | JFR | 5.9.1 | Missing routing slave supported API call ZW_GetSUCNodeID added |
| 6 | 20100817 | JFR | 5.3.1 | ApplicationRfNotify discontinued |
| 7 | 20100819 | PSH | 5.4.1 | Added ADD_NODE_STATUS_NOT_PRIMARY status to ZW_AddNodeToNetwork() call |
| 7 | 20110902 | DDA | 7.8 | Added explanation of LEDs functioning on ZDP03A |
| 7 | 20111004 | JFR | 4.7 | Warning against USING 0 attribute in ISR's |
| 7 | 20111011 | JSI | 3.3.1, 3.3.2, 3.3.6, 3.3.7, 7.2, 7.3, 7.7, 7.8 | Both None-Secure and Secure versions of Bin_Sensor(Bin_Sensor_Battery) are build using same sourcecode. It is similar for DoorLock and LED_Dimmer having none-secure and secure versions. |
| 8 | 20111101 | JFR | 3.4 | Removed Eeprom_Loader and Equinox |
| 9 | 20111108 | JFR | 7.6 & 7.7 | Updated wakeup intervals used |

Table of Contents

| | | |
|----------|---|----------|
| 1 | ABBREVIATIONS..... | 1 |
| 2 | INTRODUCTION..... | 1 |
| 2.1 | Purpose | 1 |
| 2.2 | Audience and Prerequisites | 2 |
| 3 | SOFTWARE COMPONENTS..... | 3 |
| 3.1 | Directory Structure | 3 |
| 3.2 | Z-Wave..... | 6 |
| 3.2.1 | Common | 7 |
| 3.2.2 | Include | 7 |
| 3.2.3 | I/O Defines..... | 7 |
| 3.2.4 | Libraries..... | 7 |
| 3.2.4.1 | Bridge Controller without SUC, repeater and FLiRS functionality | 7 |
| 3.2.4.2 | Installer Controller | 8 |
| 3.2.4.3 | Portable Controller | 8 |
| 3.2.4.4 | Static Controller without repeater, FLiRS and manual routing functionality..... | 9 |
| 3.2.4.5 | Static Controller without SUC and repeater functionality | 9 |
| 3.2.4.6 | Static Controller without SUC and FLiRS functionality | 9 |
| 3.2.4.7 | Enhanced Slave | 10 |
| 3.2.4.8 | Enhanced Slave without FLiRS and Manual Routing functionality | 10 |
| 3.2.4.9 | Enhanced 232 Slave | 11 |
| 3.2.4.10 | Enhanced 232 Slave without FLiRS and Manual Routing functionality | 11 |
| 3.2.4.11 | Production Test DUT..... | 11 |
| 3.2.4.12 | Production Test Generator..... | 12 |
| 3.2.4.13 | Routing Slave | 12 |
| 3.2.4.14 | Routing Slave without FLiRS and Manual Routing functionality | 12 |
| 3.2.5 | RF Frequency..... | 12 |
| 3.3 | Product..... | 12 |
| 3.3.1 | Bin..... | 13 |
| 3.3.1.1 | Bin_Sensor..... | 13 |
| 3.3.1.2 | Bin_Sensor_Sec..... | 13 |
| 3.3.1.3 | Bin_Sensor_Battery | 13 |
| 3.3.1.4 | Bin_Sensor_Battery_Sec | 14 |
| 3.3.1.5 | Dev_Ctrl | 14 |
| 3.3.1.6 | Dev_Ctrl_AVR_Sec..... | 14 |
| 3.3.1.1 | DoorBell..... | 14 |
| 3.3.1.2 | DoorLock | 15 |
| 3.3.1.3 | DoorLock_Sec..... | 15 |
| 3.3.1.4 | LED_Dimmer..... | 15 |
| 3.3.1.5 | LED_Dimmer_Sec..... | 15 |
| 3.3.1.6 | MyProduct | 15 |
| 3.3.1.7 | Prod_Test_DUT | 16 |
| 3.3.1.8 | Prod_Test_Gen | 16 |
| 3.3.1.9 | SerialAPI_Controller_Bridge_Nosuc_Norep_Noflirs..... | 16 |
| 3.3.1.10 | SerialAPI_Controller_Bridge_Nosuc_Norep_Noflirs_Mr..... | 17 |
| 3.3.1.11 | SerialAPI_Controller_Installer | 18 |
| 3.3.1.12 | SerialAPI_Controller_Installer_Mr..... | 18 |
| 3.3.1.13 | SerialAPI_Controller_Portable | 19 |
| 3.3.1.14 | SerialAPI_Controller_Portable_Mr..... | 19 |
| 3.3.1.15 | SerialAPI_Controller_Static_Norep_Noflirs_Nomr..... | 20 |
| 3.3.1.16 | SerialAPI_Controller_Static_Nosuc_Noflirs | 20 |
| 3.3.1.17 | SerialAPI_Controller_Static_Nosuc_Noflirs_Mr..... | 21 |
| 3.3.1.18 | SerialAPI_Controller_Static_Nosuc_Norep..... | 21 |

| | | |
|----------|--|-----------|
| 3.3.1.19 | SerialAPI_Slave_Enhanced | 22 |
| 3.3.1.20 | SerialAPI_Slave_Enhanced_232 | 22 |
| 3.3.1.21 | SerialAPI_Slave_Enhanced_232_Mr | 23 |
| 3.3.1.22 | SerialAPI_Slave_Enhanced_232_Noflirs | 23 |
| 3.3.1.23 | SerialAPI_Slave_Enhanced_Mr | 24 |
| 3.3.1.24 | SerialAPI_Slave_Enhanced_Noflirs | 24 |
| 3.3.1.25 | SerialAPI_Slave_Routing | 25 |
| 3.3.1.26 | SerialAPI_Slave_Routing_Mr | 25 |
| 3.3.1.27 | SerialAPI_Slave_Routing_Noflirs | 26 |
| 3.3.2 | Binary Sensor | 26 |
| 3.3.3 | Development Controller | 26 |
| 3.3.4 | Secure Development Controller based on serial API and using an AVR as host | 26 |
| 3.3.5 | Doorbell | 26 |
| 3.3.6 | Door Lock | 26 |
| 3.3.7 | LED Dimmer | 27 |
| 3.3.8 | MyProduct | 27 |
| 3.3.9 | Production Test DUT | 27 |
| 3.3.10 | Production Test Generator | 27 |
| 3.3.11 | Serial API | 27 |
| 3.3.12 | Utilities | 27 |
| 3.4 | Tools | 29 |
| 3.4.1 | ERTT | 29 |
| 3.4.2 | IncDep | 30 |
| 3.4.3 | Intel UPnP | 30 |
| 3.4.4 | Make | 30 |
| 3.4.5 | Mergehex | 30 |
| 3.4.6 | Programmer | 30 |
| 3.4.7 | PVT and RF Regulatory | 31 |
| 3.4.8 | Python | 31 |
| 3.4.9 | TextTools | 31 |
| 3.4.10 | XML Editor | 32 |
| 3.4.11 | Zniffer | 32 |
| 3.5 | PC | 33 |
| 3.5.1 | Bin | 33 |
| 3.5.2 | Source | 33 |
| 3.5.2.1 | Libraries | 33 |
| 3.5.2.2 | Sample Application | 35 |
| 4 | Z-WAVE SOFTWARE ARCHITECTURE | 37 |
| 4.1 | Z-Wave System Startup Code | 38 |
| 4.2 | Z-Wave Main Loop | 38 |
| 4.3 | Z-Wave Protocol Layers | 38 |
| 4.4 | Z-Wave Application Layer | 38 |
| 4.5 | Z-Wave Software Timers | 41 |
| 4.6 | Z-Wave Hardware Timers | 42 |
| 4.7 | Z-Wave Hardware Interrupts | 42 |
| 4.8 | Z-Wave Nodes | 43 |
| 4.8.1 | Z-Wave Portable Controller Node | 43 |
| 4.8.2 | Z-Wave Static Controller Node | 45 |
| 4.8.3 | Z-Wave Installer Controller Node | 46 |
| 4.8.4 | Z-Wave Bridge Controller Node | 47 |
| 4.8.5 | Z-Wave Routing Slave Node | 48 |
| 4.8.6 | Z-Wave Enhanced Slave Node | 50 |
| 4.8.7 | Z-Wave Enhanced 232 Slave Node | 51 |
| 4.8.8 | Adding and Removing Nodes to/from the network | 52 |
| 4.8.9 | The Automatic Network Update | 54 |

| | | |
|----------|---|-----------|
| 5 | Z-WAVE APPLICATION INTERFACES..... | 55 |
| 5.1 | Z-Wave Libraries | 55 |
| 5.1.1 | Library Functionality | 55 |
| 5.1.1.1 | Library Functionality without a SUC/SIS | 56 |
| 5.1.1.2 | Library Functionality with a SUC | 57 |
| 5.1.1.3 | Library Functionality with a SIS | 58 |
| 5.1.1.4 | Library Memory Usage | 59 |
| 5.2 | Z-Wave Header Files | 60 |
| 5.3 | Z-Wave Common API | 62 |
| 5.3.1 | Required Application Functions | 62 |
| 5.3.1.1 | ApplicationInitHW | 62 |
| 5.3.1.2 | ApplicationInitSW | 63 |
| 5.3.1.3 | ApplicationTestPoll | 63 |
| 5.3.1.4 | ApplicationPoll | 64 |
| 5.3.1.5 | ApplicationCommandHandler (Not Bridge Controller library) | 65 |
| 5.3.1.6 | ApplicationNodeInformation | 67 |
| 5.3.1.7 | ApplicationSlaveUpdate (All slave libraries) | 70 |
| 5.3.1.8 | ApplicationControllerUpdate (All controller libraries) | 71 |
| 5.3.1.9 | ApplicationCommandHandler_Bridge (Bridge Controller library only) | 73 |
| 5.3.1.10 | ApplicationSlaveNodeInformation (Bridge Controller library only) | 75 |
| 5.3.2 | Z-Wave Basis API | 76 |
| 5.3.2.1 | ZW_ExploreRequestInclusion | 76 |
| 5.3.2.2 | ZW_GetProtocolStatus | 77 |
| 5.3.2.3 | ZW_GetRandomWord | 77 |
| 5.3.2.4 | ZW_Poll | 79 |
| 5.3.2.5 | ZW_Random | 79 |
| 5.3.2.6 | ZW_RFPowerLevelSet | 80 |
| 5.3.2.7 | ZW_RFPowerLevelGet | 81 |
| 5.3.2.8 | ZW_RequestNetWorkUpdate | 82 |
| 5.3.2.9 | ZW_RFPowerlevelRediscoverySet | 84 |
| 5.3.2.10 | ZW_SendNodeInformation | 85 |
| 5.3.2.11 | ZW_SendTestFrame | 86 |
| 5.3.2.12 | ZW_SetExtIntLevel | 87 |
| 5.3.2.13 | ZW_SetPromiscuousMode (Not Bridge Controller library) | 88 |
| 5.3.2.14 | ZW_SetRFReceiveMode | 89 |
| 5.3.2.15 | ZW_SetSleepMode | 89 |
| 5.3.2.16 | ZW_Type_Library | 93 |
| 5.3.2.17 | ZW_Version | 94 |
| 5.3.2.18 | ZW_VERSION_MAJOR / ZW_VERSION_MINOR / ZW_VERSION_BETA | 95 |
| 5.3.2.19 | ZW_WatchDogEnable | 96 |
| 5.3.2.20 | ZW_WatchDogDisable | 96 |
| 5.3.2.21 | ZW_WatchDogKick | 97 |
| 5.3.3 | Z-Wave Transport API | 98 |
| 5.3.3.1 | ZW_SendData | 98 |
| 5.3.3.2 | ZW_SendData_Generic | 103 |
| 5.3.3.3 | ZW_SendData_Bridge | 108 |
| 5.3.3.4 | ZW_SendDataMeta_Generic | 112 |
| 5.3.3.5 | ZW_SendDataMeta_Bridge | 114 |
| 5.3.3.6 | ZW_SendDataMulti | 118 |
| 5.3.3.7 | ZW_SendDataMulti_Bridge | 120 |
| 5.3.3.8 | ZW_SendDataAbort | 122 |
| 5.3.3.9 | ZW_SendConst | 122 |
| 5.3.4 | Z-Wave TRIAC API | 123 |
| 5.3.4.1 | TRIAC_Init | 123 |
| 5.3.4.2 | TRIAC_SetDimLevel | 125 |
| 5.3.4.3 | TRIAC_Off | 125 |

| | | |
|-----------|-----------------------------|-----|
| 5.3.5 | Z-Wave Timer API | 126 |
| 5.3.5.1 | TimerStart | 126 |
| 5.3.5.2 | TimerRestart | 127 |
| 5.3.5.3 | TimerCancel | 127 |
| 5.3.6 | Z-Wave PWM API | 128 |
| 5.3.6.1 | ZW_PWMSetup | 128 |
| 5.3.6.2 | ZW_PWMPrescale | 129 |
| 5.3.6.3 | ZW_PWMClearInterrupt | 130 |
| 5.3.6.4 | ZW_PWMEnable | 130 |
| 5.3.7 | Z-Wave Memory API | 131 |
| 5.3.7.1 | MemoryGetID | 131 |
| 5.3.7.2 | MemoryGetByte | 132 |
| 5.3.7.3 | MemoryPutByte | 133 |
| 5.3.7.4 | MemoryGetBuffer | 134 |
| 5.3.7.5 | MemoryPutBuffer | 135 |
| 5.3.7.6 | ZW_EepromInit | 136 |
| 5.3.7.7 | ZW_MemoryFlush | 136 |
| 5.3.8 | Z-Wave ADC API | 137 |
| 5.3.8.1 | ADC_Off | 137 |
| 5.3.8.2 | ADC_Start | 137 |
| 5.3.8.3 | ADC_Stop | 137 |
| 5.3.8.4 | ADC_Init | 138 |
| 5.3.8.5 | ADC_SelectPin | 140 |
| 5.3.8.6 | ADC_Buf | 141 |
| 5.3.8.7 | ADC_SetAZPL | 142 |
| 5.3.8.8 | ADC_SetResolution | 142 |
| 5.3.8.9 | ADC_SetThresMode | 143 |
| 5.3.8.10 | ADC_SetThres | 144 |
| 5.3.8.11 | ADC_Int | 144 |
| 5.3.8.12 | ADC_IntFlagClr | 145 |
| 5.3.8.13 | ADC_GetRes | 145 |
| 5.3.9 | Z-Wave Power API | 146 |
| 5.3.9.1 | ZW_SetWutTimeout | 146 |
| 5.3.10 | UART interface API | 147 |
| 5.3.10.1 | UART_Init | 147 |
| 5.3.10.2 | UART_RecStatus | 147 |
| 5.3.10.3 | UART_RecByte | 148 |
| 5.3.10.4 | UART_SendStatus | 148 |
| 5.3.10.5 | UART_SendByte | 149 |
| 5.3.10.6 | UART_SendNum | 149 |
| 5.3.10.7 | UART_SendStr | 150 |
| 5.3.10.8 | UART_SendNL | 150 |
| 5.3.10.9 | UART_Enable | 150 |
| 5.3.10.10 | UART_Disable | 151 |
| 5.3.10.11 | UART_ClearTx | 151 |
| 5.3.10.12 | UART_ClearRx | 151 |
| 5.3.10.13 | UART_Write | 152 |
| 5.3.10.14 | UART_Read | 152 |
| 5.3.10.15 | Serial debug output | 153 |
| 5.3.11 | Z-Wave Node Mask API | 154 |
| 5.3.11.1 | ZW_NodeMaskSetBit | 154 |
| 5.3.11.2 | ZW_NodeMaskClearBit | 154 |
| 5.3.11.3 | ZW_NodeMaskClear | 155 |
| 5.3.11.4 | ZW_NodeMaskBitsIn | 155 |
| 5.3.11.5 | ZW_NodeMaskNodeIn | 156 |
| 5.4 | Z-Wave Controller API | 157 |
| 5.4.1 | ZW_AddNodeToNetwork | 157 |

| | | |
|----------|--|------------|
| 5.4.2 | ZW_AreNodesNeighbours..... | 160 |
| 5.4.3 | ZW_AssignReturnRoute..... | 161 |
| 5.4.4 | ZW_AssignSUCReturnRoute..... | 162 |
| 5.4.5 | ZW_ControllerChange..... | 163 |
| 5.4.6 | ZW_DeleteReturnRoute..... | 165 |
| 5.4.7 | ZW_DeleteSUCReturnRoute..... | 166 |
| 5.4.8 | ZW_GetControllerCapabilities..... | 167 |
| 5.4.9 | ZW_GetNeighborCount..... | 168 |
| 5.4.10 | ZW_GetNodeProtocolInfo..... | 169 |
| 5.4.11 | ZW_GetRoutingInfo..... | 170 |
| 5.4.12 | ZW_GetRoutingMAX..... | 171 |
| 5.4.13 | ZW_GetSUCNodeID..... | 171 |
| 5.4.14 | ZW_IsFailedNode..... | 172 |
| 5.4.15 | ZW_IsPrimaryCtrl..... | 172 |
| 5.4.16 | ZW_RemoveFailedNodeID..... | 173 |
| 5.4.17 | ZW_ReplaceFailedNode..... | 175 |
| 5.4.18 | ZW_RemoveNodeFromNetwork..... | 177 |
| 5.4.19 | ZW_ReplicationReceiveComplete..... | 179 |
| 5.4.20 | ZW_ReplicationSend..... | 180 |
| 5.4.21 | ZW_RequestNodeInfo..... | 181 |
| 5.4.22 | ZW_RequestNodeNeighborUpdate..... | 182 |
| 5.4.23 | ZW_SendSUCID..... | 183 |
| 5.4.24 | ZW_SetDefault..... | 184 |
| 5.4.25 | ZW_SetLearnMode..... | 185 |
| 5.4.26 | ZW_SetRoutingInfo..... | 187 |
| 5.4.27 | ZW_SetRoutingMAX..... | 188 |
| 5.4.28 | ZW_SetSUCNodeID..... | 188 |
| 5.5 | Z-Wave Static Controller API..... | 190 |
| 5.5.1 | ZW_CreateNewPrimaryCtrl..... | 190 |
| 5.5.2 | ZW_EnableSUC..... | 192 |
| 5.6 | Z-Wave Bridge Controller API..... | 193 |
| 5.6.1 | ZW_GetVirtualNodes..... | 193 |
| 5.6.2 | ZW_IsVirtualNode..... | 194 |
| 5.6.3 | ZW_SendSlaveNodeInformation..... | 195 |
| 5.6.4 | ZW_SetSlaveLearnMode..... | 196 |
| 5.7 | Z-Wave Installer Controller API..... | 199 |
| 5.7.1 | zwTransmitCount..... | 199 |
| 5.7.2 | ZW_StoreHomeID..... | 200 |
| 5.7.3 | ZW_StoreNodeInfo..... | 201 |
| 5.8 | Z-Wave Slave API..... | 202 |
| 5.8.1 | ZW_SetDefault..... | 202 |
| 5.8.2 | ZW_SetLearnMode..... | 203 |
| 5.8.3 | ZW_Support9600Only..... | 205 |
| 5.9 | Z-Wave Routing and Enhanced Slave API..... | 206 |
| 5.9.1 | ZW_GetSUCNodeID..... | 206 |
| 5.9.2 | ZW_IsNodeWithinDirectRange..... | 206 |
| 5.9.3 | ZW_RediscoveryNeeded..... | 207 |
| 5.9.4 | ZW_RequestNewRouteDestinations..... | 209 |
| 5.9.5 | ZW_RequestNodeInfo..... | 211 |
| 5.10 | Serial Command Line Debugger..... | 212 |
| 5.10.1 | ZW_DebugInit..... | 214 |
| 5.10.2 | ZW_DebugPoll..... | 214 |
| 5.11 | RF Settings in App_RFSetup.a51 file..... | 215 |
| 5.11.1 | ZW0201/ZW0301 RF parameters..... | 215 |
| 6 | HARDWARE SUPPORT DRIVERS..... | 217 |
| 6.1 | Hardware Pin Definitions..... | 217 |

| | | |
|----------|--|------------|
| 7 | APPLICATION SAMPLE CODE..... | 220 |
| 7.1 | Building ZW0x0x Sample Code..... | 220 |
| 7.1.1 | MK.BAT | 221 |
| 7.1.2 | Makefiles..... | 224 |
| 7.2 | Binary Sensor Sample Code | 226 |
| 7.2.1 | Network Wide Inclusion | 227 |
| 7.2.2 | Interface..... | 229 |
| 7.2.3 | Bin_Sensor Files | 229 |
| 7.2.3.1 | Macros for accessing the LED's..... | 230 |
| 7.3 | Binary Sensor Battery Sample Code | 232 |
| 7.3.1 | Network Wide Inclusion | 233 |
| 7.3.2 | Interface..... | 233 |
| 7.3.3 | Bin_Sensor_Battery Files..... | 234 |
| 7.4 | Development Controller Sample Code..... | 235 |
| 7.4.1 | Network Wide Inclusion | 235 |
| 7.4.2 | Dev_Ctrl Files..... | 237 |
| 7.5 | Secure Development Controller (ATmega) Sample Code | 238 |
| 7.5.1 | Dev_Ctrl_AVR_Sec Files..... | 238 |
| 7.6 | Door Bell Sample Code..... | 240 |
| 7.6.1 | Network Wide Inclusion | 240 |
| 7.6.2 | User interface | 240 |
| 7.6.3 | Door Bell Files | 241 |
| 7.7 | Door Lock Sample Code | 242 |
| 7.7.1 | Network Wide Inclusion | 243 |
| 7.7.2 | Interface..... | 243 |
| 7.7.3 | Secure Door Lock Files | 243 |
| 7.7.3.1 | Macros for accessing the Lock/Unlock..... | 244 |
| 7.8 | LED Dimmer Sample Code | 245 |
| 7.8.1 | Network Wide Inclusion | 246 |
| 7.8.2 | Interface..... | 246 |
| 7.8.3 | LED_Dimmer Files | 247 |
| 7.8.3.1 | Macros for accessing the LED's..... | 248 |
| 7.9 | MyProduct Sample Code | 249 |
| 7.9.1 | MyProduct Files..... | 249 |
| 7.10 | Production Test DUT..... | 250 |
| 7.10.1 | Production Test DUT Files | 251 |
| 7.11 | Production Test Generator | 253 |
| 7.11.1 | Production Test Generator Files..... | 255 |
| 7.12 | Serial API Embedded Sample Code | 256 |
| 7.12.1 | Supported API Calls | 256 |
| 7.12.2 | Implementation | 256 |
| 7.12.2.1 | Frame Layout | 256 |
| 7.12.2.2 | Frame Flow | 259 |
| 7.12.2.3 | Error handling..... | 261 |
| 7.12.2.4 | Restrictions on functions using buffers | 262 |
| 7.12.2.5 | Serial API capabilities..... | 262 |
| 7.12.2.6 | Serial API Softreset..... | 263 |
| 7.12.2.7 | Serial API Watchdog | 264 |
| 7.12.2.8 | Serial API Files..... | 265 |
| 7.12.3 | PC based Controller Sample Application | 266 |
| 7.12.4 | PC based Installer Tool Sample Application | 266 |
| 7.12.5 | PC based Z-Wave Bridge Sample Application..... | 266 |
| 8 | TOOL SAMPLE CODE..... | 267 |
| 8.1 | Z-Wave Programmer Firmware..... | 268 |
| 8.1.1 | ATmega_ZWaveProgFW Files..... | 268 |

| | | |
|-------------------|---|------------|
| 9 | REQUIRED DEVELOPMENT COMPONENTS | 270 |
| 9.1 | Software development components | 270 |
| 9.2 | ZW0102/ZW0201/ZW0301 single chip programmer | 270 |
| 9.3 | Hardware development components for ZW0102 | 271 |
| 9.4 | Hardware development components for ZW0201 | 272 |
| 9.5 | Hardware development components for ZW0301 | 272 |
| 9.6 | ZW0102/ZW0201/ZW0301 lock bit settings | 273 |
| 9.7 | External EEPROM initialization | 274 |
| 10 | APPLICATION NOTE: SUC/SIS IMPLEMENTATION..... | 275 |
| 10.1 | Implementing SUC support In All Nodes | 275 |
| 10.2 | Static Controllers | 275 |
| 10.2.1 | Request For Becoming SUC | 275 |
| 10.2.1.1 | Request For Becoming a SUC Node ID Server (SIS) | 275 |
| 10.2.2 | Updates From The Primary Controller | 276 |
| 10.2.3 | Assigning SUC Routes To Routing Slaves | 276 |
| 10.2.4 | Receiving Requests for Network Updates | 276 |
| 10.2.5 | Receiving Requests for new Node ID (SIS only) | 276 |
| 10.3 | The Primary Controller | 276 |
| 10.4 | Secondary Controllers | 277 |
| 10.4.1 | Knowing The SUC | 277 |
| 10.4.2 | Asking For And Receiving Updates | 277 |
| 10.5 | Inclusion Controllers | 278 |
| 10.6 | Routing Slaves | 278 |
| 11 | APPLICATION NOTE: INCLUSION/EXCLUSION IMPLEMENTATION | 280 |
| 11.1 | Including new nodes to the network | 280 |
| 11.2 | Excluding nodes from the network | 285 |
| 12 | APPLICATION NOTE: CONTROLLER SHIFT IMPLEMENTATION..... | 288 |
| 13 | REFERENCES | 289 |
| INDEX..... | | 290 |

List of Figures

| | | |
|------------|--|-----|
| Figure 1. | Software architecture | 37 |
| Figure 2. | Multiple copies of the same Set frame | 39 |
| Figure 3. | Multiple copies of the same Get/Report frame | 40 |
| Figure 4. | Simultaneous communication to a number of nodes | 41 |
| Figure 5. | Portable controller node architecture | 44 |
| Figure 6. | Routing slave node architecture | 48 |
| Figure 7. | Enhanced slave node architecture | 50 |
| Figure 8. | Node Information frame structure on application level | 69 |
| Figure 9. | PWM waveform | 129 |
| Figure 10. | Node Information frame structure without command classes | 169 |
| Figure 11. | Configuring environment variables | 221 |
| Figure 12. | Building sample applications | 222 |
| Figure 13. | Possible sample application targets | 223 |
| Figure 14. | NWI flow diagram for a slave | 228 |
| Figure 15. | NWI flow diagram for a controller | 236 |
| Figure 16. | Prod_Test_DUT test program flow | 250 |
| Figure 17. | ZW0102 Controller/Slave Unit | 271 |
| Figure 18. | ZW0102 Development Controller Unit | 271 |

| | |
|---|-----|
| Figure 19. Inclusion of a node having a SUC in the network | 276 |
| Figure 20. Requesting network updates from a SUC in the network | 277 |
| Figure 21. Inclusion of a node having a SIS in the network | 278 |
| Figure 22. Lost routing slave frame flow..... | 279 |
| Figure 23. Node inclusion frame flow | 281 |
| Figure 24. Node exclusion frame flow | 286 |
| Figure 25. Controller shift frame flow..... | 288 |

List of Tables

| | |
|--|-----|
| Table 1. ZW0102/ZW0201/ZW0301 hardware timer allocation | 42 |
| Table 2. ZW0102/ZW0201/ZW0301 Application ISR availability | 42 |
| Table 3. Controller functionality | 53 |
| Table 4. Library functionality..... | 55 |
| Table 5. Library functionality without a SUC/SIS..... | 56 |
| Table 6. Library functionality with a SUC..... | 57 |
| Table 7. Library functionality with a SIS | 58 |
| Table 8. ApplicationPoll frequency | 64 |
| Table 9. Transmit options behavior for portable and installer libraries..... | 99 |
| Table 10. App_RFSetup.a51 module definitions for ZW0201/ZW0301 | 215 |
| Table 11. Lock bits settings during development | 273 |
| Table 12. Lock bits settings in end products | 273 |

1 ABBREVIATIONS

| Abbreviation | Explanation |
|--------------|---|
| ACK | Acknowledge |
| AES | The Advanced Encryption Standard is a symmetric block cipher algorithm. The AES is a NIST-standard cryptographic cipher that uses a block length of 128 bits and key lengths of 128, 192 or 256 bits. Officially replacing the Triple DES method in 2001, AES uses the Rijndael algorithm developed by Joan Daemen and Vincent Rijmen of Belgium. |
| ANZ | Australia/New Zealand |
| API | Application Programming Interface |
| ASIC | Application Specific Integrated Circuit |
| DLL | Dynamic Link Library |
| DUT | Device Under Test |
| ERTT | Enhanced Reliability Test tool |
| EU | Europe |
| GNU | An organization devoted to the creation and support of Open Source software |
| HK | Hong Kong |
| HW | Hardware |
| IN | India |
| ISR | Interrupt Service Routines |
| LRC | Longitudinal Redundancy Check |
| MY | Malaysia |
| NAK | Not Acknowledged |
| NWI | Network Wide Inclusion |
| PRNG | Pseudo-Random Number Generator |
| PWM | Pulse Width Modulator |
| R&D | Research and Development |
| RF | Radio Frequency |
| SFR | Special Function Registers |
| SIS | SUC ID Server |
| SOF | Start Of Frame |
| SPI | Serial Peripheral Interface |
| SUC | Static Update Controller |
| UPnP | Universal Plug and Play |
| US | United States |
| WUT | Wake Up Timer |
| XML | eXtensible Markup Language |

2 INTRODUCTION

2.1 Purpose

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

2.2 Audience and Prerequisites

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

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:

```
/
  - PC
    - Bin
      - ZwaveDll
      - ZWaveInstaller
      - ZWavePCController
        - Non-secure
        - Secure
      - ZWaveUPnPBridge
    - Source
      - Libraries
        - WinFormsUI
        - ZensysFramework
        - ZensysFrameworkUI
        - ZensysFrameworkUIControls
        - ZWaveCommandClasses
        - ZWaveDll
        - ZWaveHAL
      - SampleApplications
        - ZWaveInstaller
        - ZWavePCController
        - ZWaveUPnPBridge
```

- Product
 - Bin
 - Bin_Sensor
 - Bin_Sensor_Sec
 - Bin_Sensor_Battery
 - Bin_Sensor_Battery_Sec
 - Dev_Ctrl
 - Dev_Ctrl_AVR_Sec
 - DoorBell
 - DoorLock
 - DoorLock_Sec
 - LED_Dimmer
 - LED_Dimmer_Sec
 - Prod_Test_DUT
 - Prod_Test_Gen
 - SerialAPI_Controller_Bridge_Nosuc_Norep_Noflirs
 - SerialAPI_Controller_Bridge_Nosuc_Norep_Noflirs_Mr
 - SerialAPI_Controller_Installer
 - SerialAPI_Controller_Installer_Mr
 - SerialAPI_Controller_Portable
 - SerialAPI_Controller_Portable_Mr
 - SerialAPI_Controller_Static_Norep_Noflirs_Nomr
 - SerialAPI_Controller_Static_Nosuc_Noflirs
 - SerialAPI_Controller_Static_Nosuc_Noflirs_Mr
 - SerialAPI_Controller_Static_Nosuc_Norep
 - SerialAPI_Slave_Enhanced
 - SerialAPI_Slave_Enhanced_232
 - SerialAPI_Slave_Enhanced_232_Mr
 - SerialAPI_Slave_Enhanced_232_Noflirs
 - SerialAPI_Slave_Enhanced_Mr
 - SerialAPI_Slave_Enhanced_Noflirs
 - SerialAPI_Slave_Routing
 - SerialAPI_Slave_Routing_Mr
 - SerialAPI_Slave_Routing_Noflirs
 - Bin_Sensor
 - Bin_Sensor_Sec
 - Dev_Ctrl_AVR_Sec
 - Dev_Ctrl
 - DoorBell
 - DoorLock
 - LED_Dimmer
 - LED_Dimmer_Sec
 - MyProduct
 - Prod_Test_DUT
 - Prod_Test_Gen
 - SerialAPI
 - Util_Func

- Tools
 - ERTT
 - PC
 - Z-Wave_Firmware
 - IncDep
 - Intel_UPnP
 - Make
 - Mergehex
 - Programmer
 - ATmega_ZWaveProgFW
 - PC
 - SD3402_Calibration
 - ZDP02x_Firmware
 - PVT_and_RF_regulatory
 - Python
 - TextTools
 - XML_Editor
 - PC
 - Zniffer
 - PC
 - FileConverter
 - Z-Wave_Firmware

- Z-Wave
 - Common
 - include
 - IO_defines
 - lib
 - controller_bridge_nosuc_norep_noflirs_ZW020x
 - controller_bridge_nosuc_norep_noflirs_ZW030x
 - controller_installer_ZW020x
 - controller_installer_ZW030x
 - controller_portable_ZW020x
 - controller_portable_ZW030x
 - controller_static_norep_noflirs_nomr_ZW020x
 - controller_static_norep_noflirs_nomr_ZW030x
 - controller_static_nosuc_noflirs_ZW020x
 - controller_static_nosuc_noflirs_ZW030x
 - controller_static_nosuc_norep_ZW020x
 - controller_static_nosuc_norep_ZW030x
 - slave_enhanced_232_noflirs_nomr_ZW020x
 - slave_enhanced_232_noflirs_nomr_ZW030x
 - slave_enhanced_232_ZW020x
 - slave_enhanced_232_ZW030x
 - slave_enhanced_ZW020x
 - slave_enhanced_ZW030x
 - slave_enhanced_noflirs_nomr_ZW020x
 - slave_enhanced_noflirs_nomr_ZW030x
 - slave_prodtest_dut_ZW020x
 - slave_prodtest_dut_ZW030x
 - slave_prodtest_gen_ZW020x
 - slave_prodtest_gen_ZW030x
 - slave_routing_noflirs_nomr_ZW020x
 - slave_routing_noflirs_nomr_ZW030x
 - slave_routing_ZW020x
 - slave_routing_ZW030x
- rf_freq

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 9.1.

Note! Recommending leaving 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.

3.2.1 Common

The 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.

3.2.2 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 ZW0201 and ZW0301. Refer to chapter 5.2 regarding a detailed description.

Warning: Disabled linker warning L25 'DATA TYPES DIFFERENT' to allow ZW_classcmd.h updates as device and command class development progress.
Refer to Makefile.common_ZW0x0x_appl files in Common directory regarding linker parameters.

3.2.3 I/O Defines

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

| | |
|-----------------------|---|
| AppRFSetup.a51 | This file is an assembler file that should be used to define which RF setup is to be used (ANZ/EU/HK/IN/MY/US) in the transmissions. Also if special values for capacity match array RX/TX or Normal/Low power transmission levels are needed than these can be defined here. |
| 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. |
| ZW_portdefs.h | This file contains I/O port initialization vectors on the single chips. |

3.2.4 Libraries

The lib directory structure contains all the supported libraries and single chip series.

3.2.4.1 Bridge Controller without SUC, repeater and FLIRS functionality

The lib\controller_bridge_nosuc_norep_noflirs_ZW0x0x directory contains all files needed for building a Z-Wave bridge controller application. The directory contains the following files:

ZW_controller_bridge_nosuc_norep_noflirs_zw020xs.lib
ZW_controller_bridge_nosuc_norep_noflirs_zw030xs.lib

These files are the compiled Z-Wave protocol and API library for the 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.

3.2.4.2 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_ZW020xs.lib
ZW_controller_installer_ZW030xs.lib

These files are the compiled Z-Wave protocol and API library for the 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 EEPROM must only be initialized once.

3.2.4.3 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_zw020xs.lib
ZW_controller_portable_zw030xs.lib

These files are the compiled Z-Wave protocol and API library for the 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.

3.2.4.4 Static Controller without repeater, FLiRS and manual routing functionality

The lib\controller_static_norep_noflirs_nomr_ZW0x0x directory contains all files needed for building a Z-Wave static controller application. The directory contains the following files:

| | |
|--|--|
| ZW_controller_static_norep_noflirs_nomr_zw020xs.lib ZW_controller_static_norep_noflirs_nomr_zw030xs.lib | These files are the compiled Z-Wave protocol and API library for the 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. |

3.2.4.5 Static Controller without SUC and repeater functionality

The lib\controller_static_nosuc_norep_ZW0x0x directory contains all files needed for building a Z-Wave static controller application. The directory contains the following files:

| | |
|--|--|
| ZW_controller_static_nosuc_norep_zw020xs.lib ZW_controller_static_nosuc_norep_zw030xs.lib | These files are the compiled Z-Wave protocol and API library for the 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. |

3.2.4.6 Static Controller without SUC and FLiRS functionality

The lib\controller_static_nosuc_noflirs_ZW0x0x directory contains all files needed for building a Z-Wave static controller application. The directory contains the following files:

| | |
|--|--|
| ZW_controller_static_nosuc_noflirs_zw020xs.lib ZW_controller_static_nosuc_noflirs_zw030xs.lib | These files are the compiled Z-Wave protocol and API library for the 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. |

3.2.4.7 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_ZW020xs.lib
ZW_slave_enhanced_ZW030xs.lib

These files are the compiled Z-Wave protocol and API library for the 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.

3.2.4.8 Enhanced Slave without FLIRS and Manual Routing functionality

The lib\slave_enhanced_noflirs_nomr_ZW0x0x directory contains all files needed for building a Z-Wave enhanced slave node application without FLIRS and Manual Routing functionality. The directory contains the following files:

ZW_slave_enhanced_noflirs_nomr_ZW020xs.lib
ZW_slave_enhanced_noflirs_nomr_ZW030xs.lib

These files are the compiled Z-Wave protocol and API library for the 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.

3.2.4.9 Enhanced 232 Slave

The lib\slave_enhanced_232_ZW0x0x directory contains all files needed for building a Z-Wave enhanced 232 slave node application. The directory contains the following files:

ZW_slave_enhanced_232_ZW020xs.lib
ZW_slave_enhanced_232_ZW030xs.lib

These files are the compiled Z-Wave protocol and API library for the ZW0201/ZW0301 based modules that a Z-Wave enhanced 232 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.

3.2.4.10 Enhanced 232 Slave without FLiRS and Manual Routing functionality

The lib\slave_enhanced_232_noflirs_nomr_ZW0x0x directory contains all files needed for building a Z-Wave enhanced 232 slave node application without FLiRS and Manual Routing functionality. The directory contains the following files:

ZW_slave_enhanced_232_noflirs_nomr_ZW020xs.lib
ZW_slave_enhanced_232_noflirs_nomr_ZW030xs.lib

These files are the compiled Z-Wave protocol and API library for the ZW0201/ZW0301 based modules that a Z-Wave enhanced 232 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.

3.2.4.11 Production Test DUT

The lib\slave_proptest_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_proptest_dut_ZW020xs.lib
ZW_slave_proptest_dut_ZW030xs.lib

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

3.2.4.12 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_ZW020xs.lib
ZW_slave_prodtest_gen_ZW030xs.lib

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

3.2.4.13 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_ZW020xs.lib
ZW_slave_routing_ZW030xs.lib

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

3.2.4.14 Routing Slave without FLiRS and Manual Routing functionality

The lib\slave_routing_ZW0x0x directory contains all files needed for building a Z-Wave routing slave node application without FLiRS and Manual Routing functionality. The directory contains the following files:

ZW_slave_routing_noflirs_nomr_ZW020xs.lib
ZW_slave_routing_noflirs_nomr_ZW030xs.lib

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

3.2.5 RF Frequency

The rf_freq directory contains all the object files ZW_rf_XXXX_xx.obj with RF initialization settings for each single chip series and frequency.

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.

Each directory contains the necessary files for creating ANZ (921.42 MHz), EU (868.42 MHz), HK (919.82 MHz), IN (865.22 MHz), MY (868.10 MHz) and US (908.42 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_ZW020x_y.hex | The compiled and linked binary sensor sample application for y = ANZ, EU, HK, IN, MY and US frequency frequency versions of the ZW0201 based module. |
|-------------------------------|--|

| | |
|-------------------------------|--|
| binsensor_ZW030x_y.hex | The compiled and linked binary sensor sample application for y = ANZ, EU, HK, IN, MY and US frequency frequency versions of the ZW0301 based module. |
|-------------------------------|--|

3.3.1.2 Bin_Sensor_Sec

Secure binary sensor sample application binaries not distributed due to export restrictions. Contact support via zensys_support@sigmadesigns.com for further information.

3.3.1.3 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_ZW020x_y.hex | The compiled and linked battery operated binary sensor sample application for y = ANZ, EU, HK, IN, MY and US frequency versions of the ZW0201 based module. |
|---------------------------------------|---|

| | |
|---------------------------------------|---|
| binsensor_Battery_ZW030x_y.hex | The compiled and linked battery operated binary sensor sample application for y = ANZ, EU, HK, IN, MY and US frequency versions of the ZW0301 based module. |
|---------------------------------------|---|

3.3.1.4 Bin_Sensor_Battery_Sec

Secure battery operated binary sensor sample application binaries not distributed due to export restrictions. Contact support via zensys_support@sigmadesigns.com for further information.

3.3.1.5 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_ZW020x_y.hex | The compiled and linked development controller sample application for y = ANZ, EU, HK, IN, MY and US frequency versions of the ZW0201 based module. |
| dev_ctrl_ZW030x_y.hex | The compiled and linked development controller sample application for y = ANZ, EU, HK, IN, MY and US frequency versions of the ZW0301 based module. |

3.3.1.6 Dev_Ctrl_AVR_Sec

Secure development controller sample application binaries not distributed due to export restrictions. The sample application uses an AVR ATmega128 as host on a ZDP02A/ZDP03A Development module. Configure the Z-Wave module on the ZDP02A/ZDP03A Development module with a serial API based portable controller sample application. Contact support via zensys_support@sigmadesigns.com for further information.

3.3.1.1 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_y.hex | The compiled and linked bell sample application for y = ANZ, EU, HK, IN, MY and US frequency versions of the ZW0201 based module. |
| doorbell_bell_ZW030x_y.hex | The compiled and linked bell sample application for y = ANZ, EU, HK, IN, MY and US frequency versions of the ZW0301 based module. |

3.3.1.2 DoorLock

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

| | |
|------------------------------|---|
| doorlock_ZW020x_y.hex | The compiled and linked bell sample application for y = ANZ, EU, HK, IN, MY and US frequency versions of the ZW0201 based module. |
| doorlock_ZW030x_y.hex | The compiled and linked bell sample application for y = ANZ, EU, HK, IN, MY and US frequency versions of the ZW0301 based module. |

3.3.1.3 DoorLock_Sec

Secure Secure door lock sample application binaries not distributed due to export restrictions. Contact support via zensys_support@sigmadesigns.com for further information.

3.3.1.4 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_ZW020x_y.hex | The compiled and linked LED dimmer sample application for y = ANZ, EU, HK, IN, MY and US frequency versions of the ZW0201 based module. |
| leddimmer_ZW030x_y.hex | The compiled and linked LED dimmer sample application for y = ANZ, EU, HK, IN, MY and US frequency versions of the ZW0301 based module. |

3.3.1.5 LED_Dimmer_Sec

Secure LED dimmer sample application binaries not distributed due to export restrictions. Contact support via zensys_support@sigmadesigns.com for further information.

3.3.1.6 MyProduct

No hexadecimal files available.

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_ZW020x_y.hex The compiled and linked production test DUT sample application for y = ANZ, EU, HK, IN, MY and US frequency versions of the ZW0201 based module.

prod_test_dut_ZW030x_y.hex The compiled and linked production test DUT sample application for y = ANZ, EU, HK, IN, MY and US frequency 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_ZW020x_y.hex The compiled and linked production test generator sample application for y = ANZ, EU, HK, IN, MY and US frequency versions of the ZW0201 based module.

prod_test_gen_ZW030x_y.hex The compiled and linked production test generator sample application for y = ANZ, EU, HK, IN, MY and US frequency versions of the ZW0301 based module.

3.3.1.9 SerialAPI_Controller_Bridge_Nosuc_Norep_Noflirs

The Product\Bin\SerialAPI_Controller_Bridge_Nosuc_Norep_Noflirs directory contains all files needed for running a serial API based bridge controller sample application without SUC/SIS, repeater and FLIRS BEAM functionality 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_nosuc_norep_noflirs_ZW020x_y.hex The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US frequency versions of the ZW0201 based module.

serialapi_controller_bridge_nosuc_norep_noflirs_ZW030x_y.hex The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US frequency versions of the ZW0301 based module.

SupportedFunctions_SerialAPI_bridge_nosuc_norep_noflirs.txt Show enabled (1) and disabled (0) serial API calls of released sample

application.

3.3.1.10 SerialAPI_Controller_Bridge_Nosuc_Norep_Noflirs_Mr

The Product\Bin\SerialAPI_Controller_Bridge_Nosuc_Norep_Noflirs_Mr directory contains all files needed for running a serial API based bridge controller sample application without SUC/SIS, repeater and FLiRS BEAM functionality on a Z-Wave module using manual routing. 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_nosuc_norep_noflirs_mr_ZW020x_y.hex

The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0201 based module.

serialapi_controller_bridge_nosuc_norep_noflirs_mr_ZW030x_y.hex

The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0301 based module.

SupportedFunctions_SerialAPI_bridge_nosuc_norep_noflirs_mr.txt

Show enabled (1) and disabled (0) serial API calls of released sample application.

3.3.1.11 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_ZW020x_y.hex | The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0201 based module. |
| serialapi_controller_installer_ZW030x_y.hex | The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0301 based module. |
| SupportedFunctions_SerialAPI_controller_installer.txt | Show enabled (1) and disabled (0) serial API calls of released sample application. |

3.3.1.12 SerialAPI_Controller_Installer_Mr

The Product\Bin\SerialAPI_Controller_Installer_Mr directory contains all files needed for running a serial API based installer controller sample application on a Z-Wave module using manual routing. 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_mr_ZW020x_y.hex | The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0201 based module. |
| serialapi_controller_installer_mr_ZW030x_y.hex | The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0301 based module. |
| SupportedFunctions_SerialAPI_controller_installer_mr.txt | Show enabled (1) and disabled (0) serial API calls of released sample application. |

3.3.1.13 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_ZW020x_y.hex | The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0201 based module. |
| serialapi_controller_portable_ZW030x_y.hex | The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0301 based module. |
| SupportedFunctions_SerialAPI_controller_portable.txt | Show enabled (1) and disabled (0) serial API calls of released sample application. |

3.3.1.14 SerialAPI_Controller_Portable_Mr

The Product\Bin\SerialAPI_Controller_Portable_Mr directory contains all files needed for running a serial API based portable controller sample application on a Z-Wave module using manual routing. 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_mr_ZW020x_y.hex | The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0201 based module. |
| serialapi_controller_portable_mr_ZW030x_y.hex | The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0301 based module. |
| SupportedFunctions_SerialAPI_controller_portable_mr.txt | Show enabled (1) and disabled (0) serial API calls of released sample application. |

3.3.1.15 SerialAPI_Controller_Static_Norep_Noflirs_Nomr

The Product\Bin\SerialAPI_Controller_Static_Norep_Noflirs_Nomr directory contains all files needed for running a serial API based static controller sample application without repeater, FLiRS BEAM and manual routing functionality 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_norep_noflirs_nomr_ZW020x_y.hex | The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0201 based module. |
| serialapi_controller_static_norep_noflirs_nomr_ZW030x_y.hex | The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0301 based module. |
| SupportedFunctions_SerialAPI_controller_static_norep_noflirs_nomr.txt | Show enabled (1) and disabled (0) serial API calls of released sample application. |

3.3.1.16 SerialAPI_Controller_Static_Nosuc_Noflirs

The Product\Bin\SerialAPI_Controller_Static_Nosuc_Noflirs directory contains all files needed for running a serial API based static controller sample application without SUC/SIS and FLiRS BEAM functionality 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_nosuc_noflirs_ZW020x_y.hex | The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0201 based module. |
| serialapi_controller_static_nosuc_noflirs_ZW030x_y.hex | The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0301 based module. |
| SupportedFunctions_SerialAPI_controller_static_nosuc_noflirs.txt | Show enabled (1) and disabled (0) serial API calls of released sample application. |

3.3.1.17 SerialAPI_Controller_Static_Nosuc_Noflirs_Mr

The Product\Bin\SerialAPI_Controller_Static_Nosuc_Mr directory contains all files needed for running a serial API based static controller sample application without SUC/SIS and FLIRS BEAM functionality on a Z-Wave module using manual routing. 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_nosuc_noflirs_mr_ZW020x_y.hex | The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0201 based module. |
| serialapi_controller_static_nosuc_noflirs_mr_ZW030x_y.hex | The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0301 based module. |
| SupportedFunctions_SerialAPI_controller_static_nosuc_noflirs_mr.txt | Show enabled (1) and disabled (0) serial API calls of released sample application. |

3.3.1.18 SerialAPI_Controller_Static_Nosuc_Norep

The Product\Bin\SerialAPI_Controller_Static_Nosuc_Norep directory contains all files needed for running a serial API based static controller sample application without SUC/SIS and repeater functionality 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_nosuc_norep_ZW020x_y.hex | The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0201 based module. |
| serialapi_controller_static_nosuc_norep_ZW030x_y.hex | The compiled and linked serial API based static controller sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0301 based module. |
| SupportedFunctions_SerialAPI_controller_static_nosuc_norep.txt | Show enabled (1) and disabled (0) serial API calls of released sample application. |

3.3.1.19 SerialAPI_Slave_Enhanced

The Product\Bin\SerialAPI_Slave_Enhanced directory contains all files needed for running a serial API based enhanced slave 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_slave_enhanced_ZW020x_y.hex | The compiled and linked serial API based slave sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0201 based module. |
| serialapi_slave_enhanced_ZW030x_y.hex | The compiled and linked serial API based slave sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0301 based module. |
| SupportedFunctions_SerialAPI_slave_enhanced.txt | Show enabled (1) and disabled (0) serial API calls of released sample application. |

3.3.1.20 SerialAPI_Slave_Enhanced_232

The Product\Bin\SerialAPI_Slave_Enhanced_232 directory contains all files needed for running a serial API based enhanced 232 slave 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_slave_enhanced_232_ZW020x_y.hex | The compiled and linked serial API based enhanced 232 slave sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0201 based module. |
| serialapi_slave_enhanced_232_ZW030x_y.hex | The compiled and linked serial API based enhanced 232 slave sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0301 based module. |
| SupportedFunctions_SerialAPI_slave_enhanced_232.txt | Show enabled (1) and disabled (0) serial API calls of released sample application. |

3.3.1.21 SerialAPI_Slave_Enhanced_232_Mr

The Product\Bin\SerialAPI_Slave_Enhanced_232_Mr directory contains all files needed for running a serial API based enhanced 232 slave sample application on a Z-Wave module using manual routing. 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_slave_enhanced_232_mr_ZW020x_y.hex

The compiled and linked serial API based slave sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0201 based module.

serialapi_slave_enhanced_232_mr_ZW030x_y.hex

The compiled and linked serial API based slave sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0301 based module.

SupportedFunctions_SerialAPI_slave_enhanced_232_mr.txt

Show enabled (1) and disabled (0) serial API calls of released sample application.

3.3.1.22 SerialAPI_Slave_Enhanced_232_Noflirs

The Product\Bin\SerialAPI_Slave_Enhanced_232_Noflirs directory contains all files needed for running a serial API based enhanced 232 slave sample application on a Z-Wave module without FLiRS functionality. 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_slave_enhanced_232_noflirs_ZW020x_y.hex

The compiled and linked serial API based slave sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0201 based module.

serialapi_slave_enhanced_232_noflirs_ZW030x_y.hex

The compiled and linked serial API based slave sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0301 based module.

SupportedFunctions_SerialAPI_slave_enhanced_232_noflirs.txt

Show enabled (1) and disabled (0) serial API calls of released

sample application.

3.3.1.23 SerialAPI_Slave_Enhanced_Mr

The Product\Bin\SerialAPI_Slave_Enhanced_Mr directory contains all files needed for running a serial API based enhanced slave sample application on a Z-Wave module using manual routing. 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_slave_enhanced_mr_ZW020x_y.hex | The compiled and linked serial API based slave sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0201 based module. |
| serialapi_slave_enhanced_mr_ZW030x_y.hex | The compiled and linked serial API based slave sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0301 based module. |
| SupportedFunctions_SerialAPI_slave_enhanced_mr.txt | Show enabled (1) and disabled (0) serial API calls of released sample application. |

3.3.1.24 SerialAPI_Slave_Enhanced_Noflirs

The Product\Bin\SerialAPI_Slave_Enhanced_Noflirs directory contains all files needed for running a serial API based enhanced slave sample application on a Z-Wave module without FLiRS functionality. 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_slave_enhanced_noflirs_ZW020x_y.hex | The compiled and linked serial API based slave sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0201 based module. |
| serialapi_slave_enhanced_noflirs_ZW030x_y.hex | The compiled and linked serial API based slave sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0301 based module. |
| SupportedFunctions_SerialAPI_slave_enhanced_noflirs.txt | Show enabled (1) and disabled (0) serial API calls of released sample application. |

3.3.1.25 SerialAPI_Slave_Routing

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

| | |
|---|---|
| serialapi_slave_routing_ZW020x_y.hex | The compiled and linked serial API based slave sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0201 based module. |
| serialapi_slave_routing_ZW030x_y.hex | The compiled and linked serial API based slave sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0301 based module. |
| SupportedFunctions_SerialAPI_slave_routing_noflirs.txt | Show enabled (1) and disabled (0) serial API calls of released sample application. |

3.3.1.26 SerialAPI_Slave_Routing_Mr

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

| | |
|--|---|
| serialapi_slave_routing_mr_ZW020x_y.hex | The compiled and linked serial API based slave sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0201 based module. |
| serialapi_slave_routing_mr_ZW030x_y.hex | The compiled and linked serial API based slave sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0301 based module. |
| SupportedFunctions_SerialAPI_slave_routing_mr.txt | Show enabled (1) and disabled (0) serial API calls of released sample application. |

3.3.1.27 SerialAPI_Slave_Routing_Noflirs

The Product\Bin\SerialAPI_Slave_Routing_Noflirs directory contains all files needed for running a serial API based routing slave sample application on a Z-Wave module without FLiRS functionality. The directory contains the following files:

| | |
|---|---|
| serialapi_slave_routing_noflirs_ZW020x_y.hex | The compiled and linked serial API based slave sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0201 based module. |
| serialapi_slave_routing_noflirs_ZW030x_y.hex | The compiled and linked serial API based slave sample application for y = ANZ, EU, HK, IN, MY and US versions of the ZW0301 based module. |
| SupportedFunctions_SerialAPI_slave_routing_noflirs.txt | Show enabled (1) and disabled (0) serial API calls of released sample application. |

3.3.2 Binary Sensor

The Product\Bin_Sensor directory contains sample source code for a non-secure/secure binary sensor and non-secure/secure battery operated binary sensor application (see section 7.2 and 7.3).

3.3.3 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].

3.3.4 Secure Development Controller based on serial API and using an AVR as host

The Product\Dev_Ctrl_AVR_Sec directory contains sample source code for the secure development controller. The sample application uses an AVR ATmega128 as host on a ZDP02A/ZDP03A Development module. Configure the Z-Wave module on the ZDP02A/ZDP03A Development module with a serial API based portable controller sample application. For further information refer to section 7.5 and reference [15].

3.3.5 Doorbell

The Product\DoorBell directory contains sample source code for the doorbell application used on the Z-Wave Interface module. Use the Development Controller application to control the doorbell application. For further information, refer to section 7.6.

3.3.6 Door Lock

The Product\DoorLock directory contains sample source code for the non-secure and secure door lock application used on the Z-Wave Interface module. Use the Secure Development Controller application to control the door lock application. For further information, refer to section 7.7.

3.3.7 LED Dimmer

The Product\LED_Dimmer directory contains sample source code for the non-secure and secure dimmer application on a Z-Wave module, which uses the LEDs to simulate a light switch with a built in dimmer. For further information, refer to section 7.8.

3.3.8 MyProduct

The Product\MyProduct directory contains sample source code for a routing slave application on a Z-Wave module. For further information, refer to section 7.9.

3.3.9 Production Test DUT

The Product\Prod_Test_DUT directory contains sample source code for a production test DUT application on a Z-Wave module. For further information, refer to section 7.10.

3.3.10 Production Test Generator

The Product\Prod_Test_Gen directory contains sample source code for a production test generator application on a Z-Wave module. For further information, refer to section 7.11.

3.3.11 Serial API

The Product\SerialAPI directory contains sample source code for the Serial API sample applications. For further information about the Serial API, refer to section 7.12.

3.3.12 Utilities

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

8051_AES_common.a51 8051_AES_core.a51

These files contain shared data and functions for AES128 and functions for AES128 encryption/decryption. Files are not distributed on the Developer's Kit CD due to export restrictions.

The sample applications uses a 3rd party product subjected to intellectual property (IP) rights and licensing issues. The files can be acquired from the rightful IP owner (Stiftung Secure Information and Communication Technologies (SIC) in Austria) for approximately €2.000:

http://jce.iaik.tugraz.at/sic/sales/price_list/hardware_related_products

Alternatively, implement the functions based on Atmel's Application Note "AVR231: AES Bootloader":

http://www.atmel.com/dyn/resources/prod_documents/doc2589.pdf

AES_module.h

This header file contains definitions for implementing secure communication using AES as encrypting/decrypting engine.

association.c association.h

The files contain 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**

The files contain 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.

ctrl_learn.h **ctrl_learn.c**

The files contain sample code for how to handle learn mode on controller nodes.

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.

slave_learn.h **slave_learn.c**

The files contain sample code for how to handle learn mode on slave nodes. These two files are used by all slave based sample code in the ZDK. The sample application should just call StartLearnModeNow() to enter learnmode and transmit nodeinformation. Inclusion uses normal power. The sample application should then wait for the BOOL learnState to go FALSE before doing transmissions.

| | |
|--|---|
| ZW_AES128.h | This header file contains definitions for the security solution on application level. |
| ZW_FLiRS.c ZW_FLiRS.h | The files contain sample code for how to handle FLiRS nodes. |
| ZW_Security_AES_module.c ZW_Security_AES_module.h | The files contain sample code for the functionality supporting secure communication using AES as encryption/decryption mechanism. |
| ZW_TransportLayer.h | Transport layer type selector |
| ZW_TransportNative.h | Implements functions for transporting frames over the native Z-Wave Network. |
| ZW_TransportSecurity.h ZW_TransportSecurity.c | Implements functions for transporting frames over the secure Z-Wave Network. |

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 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_controller_single_ZW020x_ANZ.hex Z-Wave_Firmware\serialapi_controller_single_ZW020x_EU.hex Z-Wave_Firmware\serialapi_controller_single_ZW020x_HK.hex Z-Wave_Firmware\serialapi_controller_single_ZW020x_US.hex | The compiled and linked ERTT application for the ANZ, EU, HK and US frequency versions of the ZW0201 based module. |
| Z-Wave_Firmware\serialapi_controller_single_ZW030x_ANZ.hex Z-Wave_Firmware\serialapi_controller_single_ZW030x_EU.hex Z-Wave_Firmware\serialapi_controller_single_ZW030x_HK.hex Z-Wave_Firmware\serialapi_controller_single_ZW030x_US.hex | The compiled and linked ERTT application for the ANZ, EU, HK and US frequency versions of the ZW0301 based module. |

3.4.2 IncDep

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

3.4.3 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].

3.4.4 Make

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

3.4.5 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.

3.4.6 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 | Programmer application. |
| PC\CP210x_VCP_Win_XP_S2K3_Vista_7.exe | The CP210x USB to UART Bridge Virtual COM Port (VCP) driver. This driver supports Windows XP/2003/Vista(32/64)/7(32/64). |
| ZDP0xA_Firmware\ATMega128_Firmware.hex | The compiled and linked Z-Wave Programmer bootloader for the ATMega128 situated on the ZDP02A/ZDP03A Development Platform. |
| ZDP0xA_Firmware\ZWaveProgrammer_FW.hex | The compiled and linked Z-Wave Programmer firmware for the ATMega128 situated on the ZDP02A/ZDP03A Development Platform. |
| ATmega_ZWaveProgFW\... | Z-Wave Programmer firmware source code for the ATMega128 situated on the ZDP02A/ZDP03A Development Platform. For further details regarding communication protocol interface, refer to [34]. |
| SD3402_Calibration\SD3402_Calibration.hex | SD3402 crystal calibration firmware used by the calibration box, refer to [35]. ZM4101 and ZM4102 are already calibrated during production. |

3.4.7 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:

| | |
|---|---|
| ZW0201_rx_y.hex | Puts the ZW0201 in receive mode for y = ANZ, EU, HK, IN, MY and US frequency versions of the ZW0201 based products. |
| ZW0201_TXcar_y.hex | ZW0201 constantly transmits a carrier y = 921.42MHz (ANZ) or 868.42MHz (EU) or 919.82MHz (HK) or 865.22 MHz (IN) or 868.10MHz (MY) 868.42MHz or 908.42MHz (US). |
| ZW0201_TXmod_y.hex ZW0201_TXmod_40kbps_y.hex | ZW0201 constantly transmits a modulated signal y +/- 25kHz, where y = 921.42MHz (ANZ) or 868.42MHz (EU) or 919.82MHz (HK) or 865.22 MHz (IN) or 868.10MHz (MY) 868.42MHz or 908.42MHz (US). |
| ZW0301_rx_y.hex | Puts the ZW0301 in receive mode. y = ANZ, EU, HK, IN, MY and US frequency versions of the ZW0301 based products. |
| ZW0301_TXcar_y.hex | ZW0301 constantly transmits a carrier y = 921.42MHz (ANZ) or 868.42MHz (EU) or 919.82MHz (HK) or 865.22 MHz (IN) or 868.10MHz (MY) 868.42MHz or 908.42MHz (US). |
| ZW0301_TXmod_y.hex ZW0301_TXmod_40kbps_y.hex | ZW0301 constantly transmits a modulated signal y +/- 25kHz, where y = 921.42MHz (ANZ) or 868.42MHz (EU) or 919.82MHz (HK) or 865.22 MHz (IN) or 868.10MHz (MY) 868.42MHz or 908.42MHz (US). |

3.4.8 Python

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

3.4.9 TextTools

This directory contains the sed stream editor used to modify text strings during the make process.

3.4.10 XML Editor

This directory contains the XML Editor program; the program can be used to define approved Z-Wave device and command classes used by the application layer of the Z-Wave protocol. The XML file can be used by the Zniffer for interpretation of the device and command classes. The customer can also define device and command classes under development or proprietary command class structures enabling interpretation by the Zniffer.

Beside a XML file containing all the information, it is also possible to generate a C# class file and C header file as foundation for Z-Wave application development. For further details, refer to [28].

The XML Editor directory contains the following files:

| | |
|---------------------|-------------------------|
| PC\setup.exe | XML Editor application. |
| PC\Setup.msi | |

3.4.11 Zniffer

This directory contains the Zniffer program; the program is a development tool for capturing Z-Wave RF communication and presenting the frames in a graphical user interface on a PC. The tool shows the node ID of the Source and Destination for the communication, the type of frame being sent, and the application content, i.e. the specific command, which is being sent.

The Zniffer tool is a passive "listener" to the Z-Wave network traffic, and will only display the RF communications taking place within direct RF range. Be also aware that Zniffer can occasionally miss RF communication even from Z-Wave nodes within direct range.

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 | Zniffer application supporting Windows |
| PC\ZnifferSetup.msi | XP/2003/Vista(32/64)/7(32/64) |
| PC\FileConverter\setup.exe | FileConverter enable Zniffer to automatically |
| PC\FileConverter\FileConverterSetup.msi | convert old file formats *.znf to latest *.zlf when |
| | opening file. |
| Z-Wave_Firmware\sniffer_ZW040x.hex | Zniffer application supporting ANZ, EU, HK, IN, |
| | JP, MY and US versions on a 400 Series based |
| | module. |
| Z-Wave_Firmware\sniffer_ZW030x_y.hex | Zniffer application supporting y = ANZ, EU, HK, IN, |
| | MY and US frequency versions on a ZW0301 |
| | based module. |
| Z-Wave_Firmware\sniffer_ZW020x_y.hex | Zniffer application supporting y = ANZ, EU, HK, IN, |
| | MY and US frequency versions on a ZW0201 |
| | 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.

| | |
|---|---|
| ZWaveDll\setup.exe ZWaveDll\ZWaveSetup.msi | Executables of the Z-Wave DLL used by the PC sample applications. |
| ZWaveInstaller\setup.exe ZWaveInstaller\ZWaveInstallerToolSetup.msi | Executables of the Z-Wave Installer Tool sample application. |
| ZWavePCController\Non-secure\setup.exe ZWavePCController\Non-secure\ZWaveControllerSetup.msi | Executables of the Z-Wave Non-secure PC Controller sample application. |
| ZWavePCController\Secure\setup.exe ZWavePCController\Secure\ZWaveControllerSetup.msi | Executables of the Z-Wave Secure PC Controller sample application. Binaries not distributed due to export restrictions. Contact support via zensys_support@sigmadesigns.com for further information. |
| ZWaveUPnPBridge\setup.exe ZWaveUPnPBridge\ZWaveUPnPBridgeSetup.msi | Executables of the Z-Wave to UPnP Bridge sample application. |

3.5.2 Source

The PC\Source directory contains the C# source code of the PC sample applications using the Microsoft Visual Studio 2008 environment.

3.5.2.1 Libraries

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

3.5.2.1.1 WinForms UI

The PC\Source\Libraries\WinFormsUI directory contains C# source code of the windows docking library.

| | |
|--------------------------|--|
| WinFormsUI.csproj | Microsoft Visual Studio 2008 project file containing information at the project level and used to build the project. |
|--------------------------|--|

3.5.2.1.2 Zensys Framework

The PC\Source\Libraries\ZensysFramework directory contains C# source code of the additional functions, formatters, helpers.

ZensysFramework.csproj Microsoft Visual Studio 2008 project file containing information at the project level and used to build the project.

3.5.2.1.3 Zensys Framework UI

The PC\Source\Libraries\ZensysFrameworkUI directory contains C# source code of the completed Z-Wave UI elements that can be reused in applications:

- Associations View Control;
- Bridged UPnP Device View Control;
- Controller View Control;
- Node View Control;
- UPnP Binary Light Device View Control;
- UPnP Device Scanner View Control;
- UPnP Media Renderer View Control.

ZensysFrameworkUI.csproj Microsoft Visual Studio 2008 project file containing information at the project level and used to build the project.

3.5.2.1.4 Zensys Framework UI Controls

The PC\Source\Libraries\ZensysFrameworkUIControls directory contains C# source code of the additional UI elements such as:

- ListDataView;
- TreeDataView;
- BitBox;
- ThreadSafeLabel.

ZensysFrameworkUIControls.csproj Microsoft Visual Studio 2008 project file containing information at the project level and used to build the project.

3.5.2.1.5 Z-Wave Command Class

The PC\Source\Libraries\ZWWaveCommandClasses directory contains C# source code for the XML parser, which enables parsing of Z-Wave frames by the Zniffer and generating frames by the PC based applications.

ZWWaveCommandClasses.csproj Microsoft Visual Studio 2008 project file containing information at the project level and used to build the project.

3.5.2.1.6 Z-Wave DLL

The PC\Source\Libraries\ZWWaveDll directory contains C# 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.

SerialZWWaveDll.sln Microsoft Visual Studio 2008 solutions file containing information at the project level and used to build the project.

3.5.2.1.7 Z-Wave HAL

The PC\Source\Libraries\ZWWaveHAL directory contains C# source code of the ZWave High-level Application Layer in terms of ZWave Dll architecture. It contains common functions that are used in Z-Wave enabled PC applications: ZWavePCController, ZWaveProgrammer, ZWaveUPnPBridge etc. Refer to [13] for further details.

SerialZWWaveHAL.sln Microsoft Visual Studio 2008 solutions file containing information at the project level and used to build the project.

3.5.2.2 Sample Application

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

3.5.2.2.1 Z-Wave Installer

The PC\Source\SampleApplications\ZWWaveInstaller directory contains C# sample source code for a PC based installer tool using the Z-Wave DLL etc. Further reading on how to use the PC based Installer Tool see [7].

ZWWaveInstallerTool.sln Microsoft Visual Studio 2008 solutions file containing information at the project level and used to build the project.

3.5.2.2.2 Z-Wave PC Controller

The PC\Source\SampleApplications\ZWWavePCController directory contains C# 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].

ZWWaveController.sln Microsoft Visual Studio 2008 solutions file containing information at the project level and used to build the project.

3.5.2.2.3 Z-Wave UPnP Bridge

The PC\Source\SampleApplications\ZWaveUPnPBridge directory contains C# 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 2008 solutions file containing information at the project level and used to build the project.

4 Z-WAVE SOFTWARE ARCHITECTURE

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

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

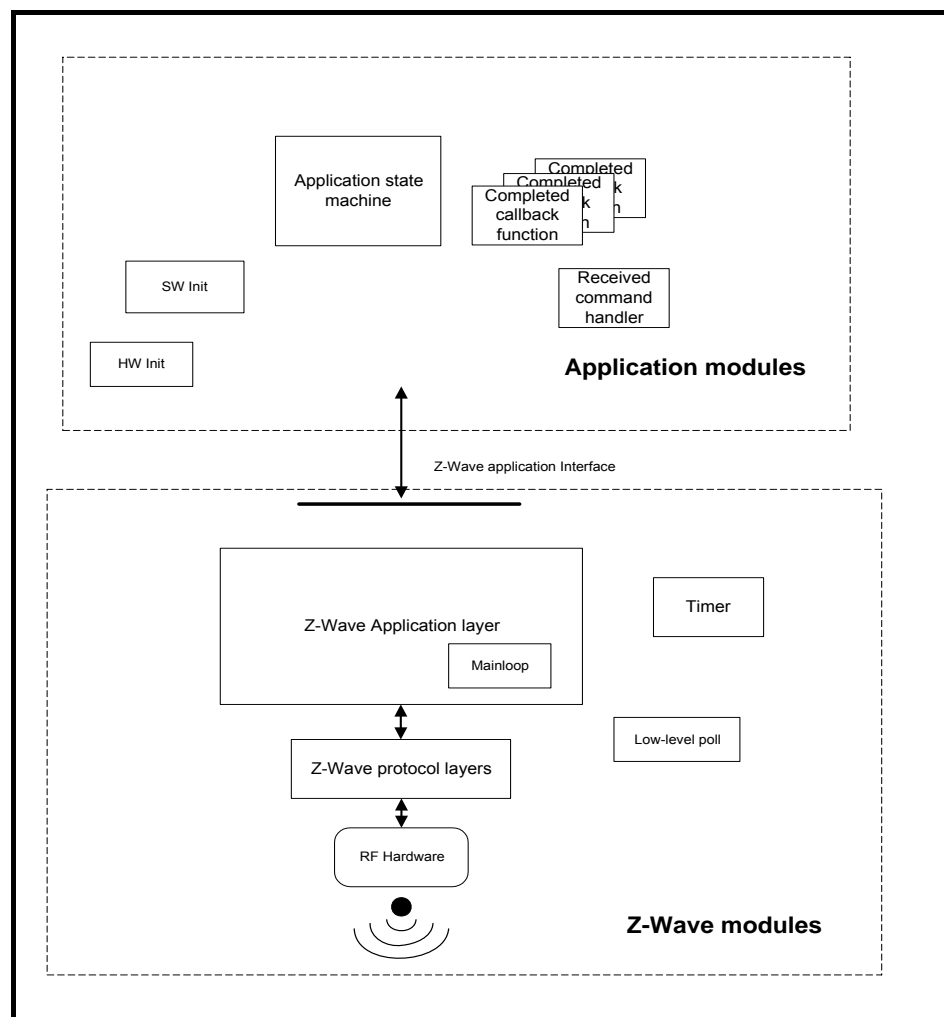


Figure 1. Software architecture

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 initialized 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 `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 and the **ApplicationCommandHandler** function (if a frame was received) in round robin order. The functions must therefore be designed to return to the caller as fast as possible to allow the CPU to do other tasks. Busy loops are not allowed. This will make it possible to receive Z-Wave data, transfer data via the UART and check user-activated buttons “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 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**.

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

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

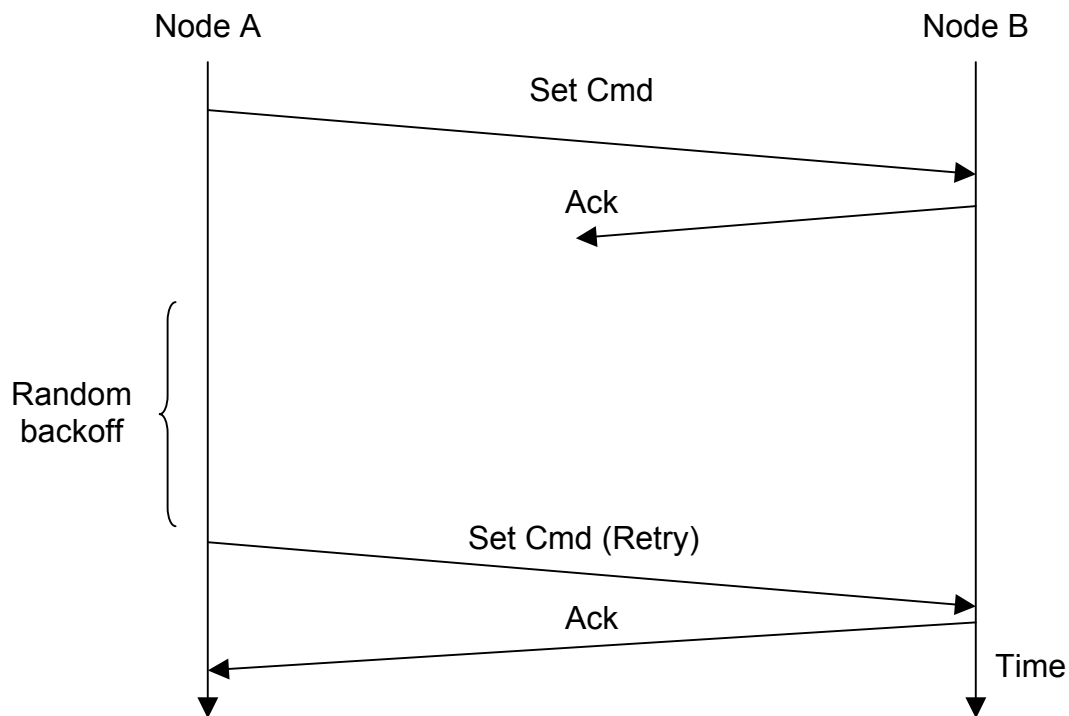


Figure 2. Multiple copies of the same Set frame

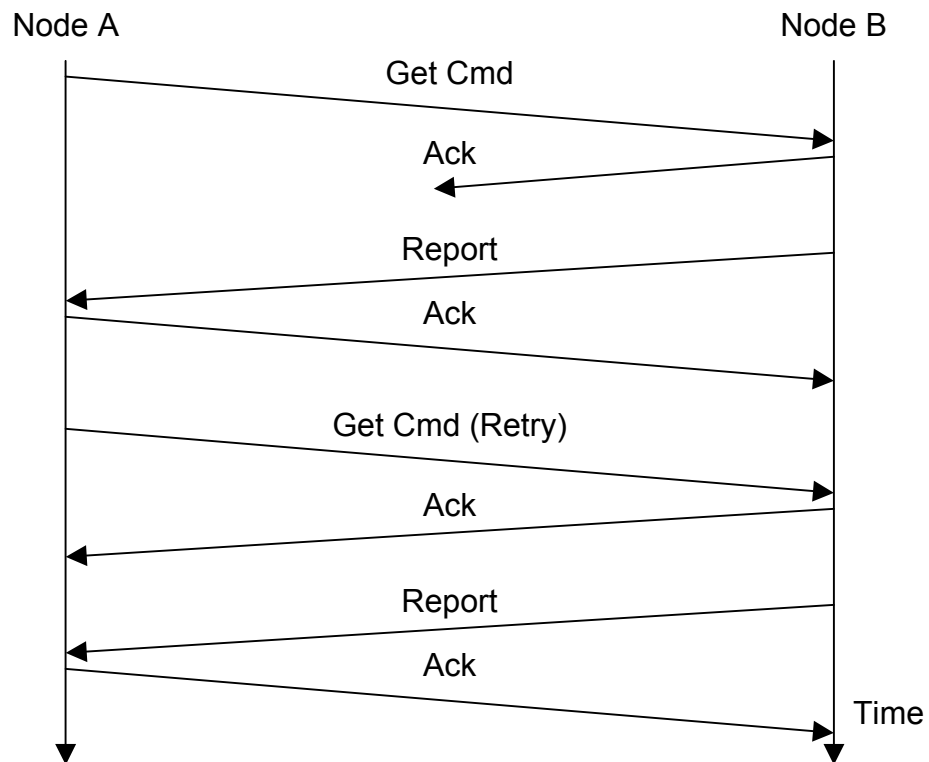


Figure 3. Multiple copies of the same Get/Report frame

A Z-Wave protocol is designed to have low latency on the expense of handling simultaneously communication to a number of nodes in the Z-Wave network. To obtain this is the number of random backoff values limited to 4 (0, 1, 2 and 3). The figure below shows how simultaneous communication to even a small number of nodes easily can block the communication completely.

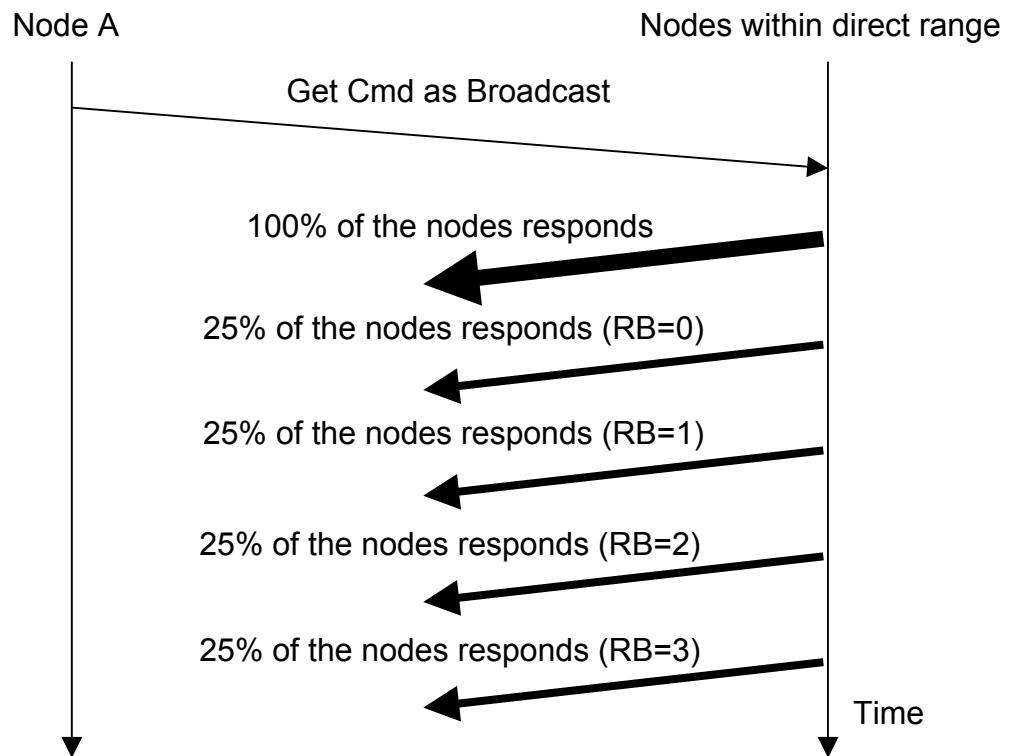


Figure 4. Simultaneous communication to a number of nodes

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

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.

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

Table 2. ZW0102/ZW0201/ZW0301 Application ISR availability

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

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

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, 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".

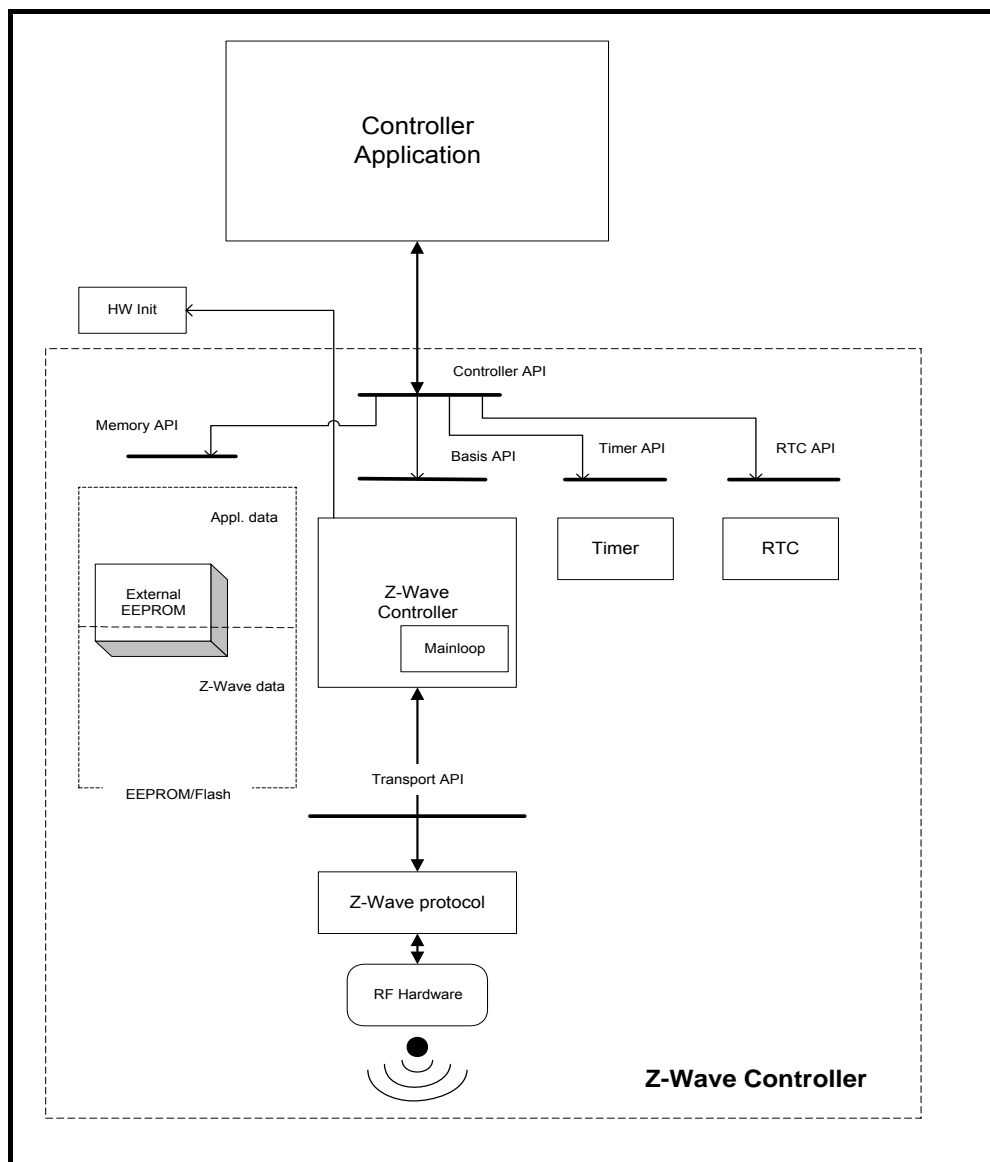


Figure 5. Portable controller node architecture

The Portable Controller node has a unique home ID number assigned, which is stored in the Z-Wave basis area of the external EEPROM. Care must be taken, when reprogramming the external EEPROM, that different controller nodes do not get the same home ID number. Refer to paragraph 9.7 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.

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

- If last working route do not exist and TRANSMIT_OPTION_ACK set. Try direct with retries.
- If last working route exist and TRANSMIT_OPTION_ACK set. Try direct without retries. In case it fails, try the last working route. In case the last working route also fails, purge it.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set then calculate up to two routing attempts per entry/repeater node. In case TRANSMIT_OPTION_EXPLORE set, a maximum number limits number of tries.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set, then direct with retries.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_EXPLORE are set then issue an explore frame as last resort.

The last working route comprises of 232 destinations having up to one route/direct each, which are stored in non-volatile memory. Last working route can also contain direct attempts. The protocol will update last workings routes in the following situations:

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

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

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

The application must be linked with ZW_CONTROLLER_PORTABLE_ZW*S.LIB
(* = 030X for ZW0301 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 secondary static controller not functioning as SUC can also request network Topology updates.

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

- If last working route do not exist and TRANSMIT_OPTION_ACK set. Try direct when neighbors.
- If last working route exist and TRANSMIT_OPTION_ACK set. Try the last working route. In case the last working route fails, purge it and try direct if neighbor.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set then calculate up to two routing attempts per entry/repeater node. In case TRANSMIT_OPTION_EXPLORE set, a maximum number limits number of tries.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set, then direct with retries.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_EXPLORE are set then issue an explore frame as last resort.

The last working route comprises of 232 destinations having up to one route/direct each, which are stored in non-volatile memory. Last working route can also contain direct attempts. The protocol will update last workings routes in the following situations:

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

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

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

The application must be linked with ZW_CONTROLLER_STATIC_ZW*S.LIB
(* = 030X for ZW0301 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 routing algorithm in an installer controller tries to reach the destination depending on the transmit options as follows:

- If last working route do not exist and TRANSMIT_OPTION_ACK set. Try direct with retries.
- If last working route exist and TRANSMIT_OPTION_ACK set. Try direct without retries. In case it fails, try the last working route. In case the last working route also fails, purge it.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set then calculate up to two routing attempts per entry/repeater node. In case TRANSMIT_OPTION_EXPLORE set, a maximum number limits number of tries.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set, then direct with retries.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_EXPLORE are set then issue an explore frame as last resort.

The last working route comprises of 232 destinations having up to one route/direct each, which are stored in non-volatile memory. Last working route can also contain direct attempts. The protocol will update last workings routes in the following situations:

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

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

The application must be linked with ZW_CONTROLLER_INSTALLER_ZW*S.LIB
(* = 030X for ZW0301 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 essentially a Z-Wave Static Controller node, which incorporates extra functionality that can be used to implement controllers, targeted for bridging between the Z-Wave network and others network (ex. UPnP).

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

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

- If last working route do not exist and TRANSMIT_OPTION_ACK set. Try direct when neighbors.
- If last working route exist and TRANSMIT_OPTION_ACK set. Try the last working route. In case the last working route fails, purge it and try direct if neighbor.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set then calculate up to two routing attempts per entry/repeater node. In case TRANSMIT_OPTION_EXPLORE set, a maximum number limits number of tries.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set, then direct with retries.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_EXPLORE are set then issue an explore frame as last resort.

The last working route comprises of 232 destinations having up to one route/direct each, which are stored in non-volatile memory. Last working route can also contain direct attempts. The protocol will update last workings routes in the following situations:

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

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

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

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

4.8.5 Z-Wave Routing Slave Node

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

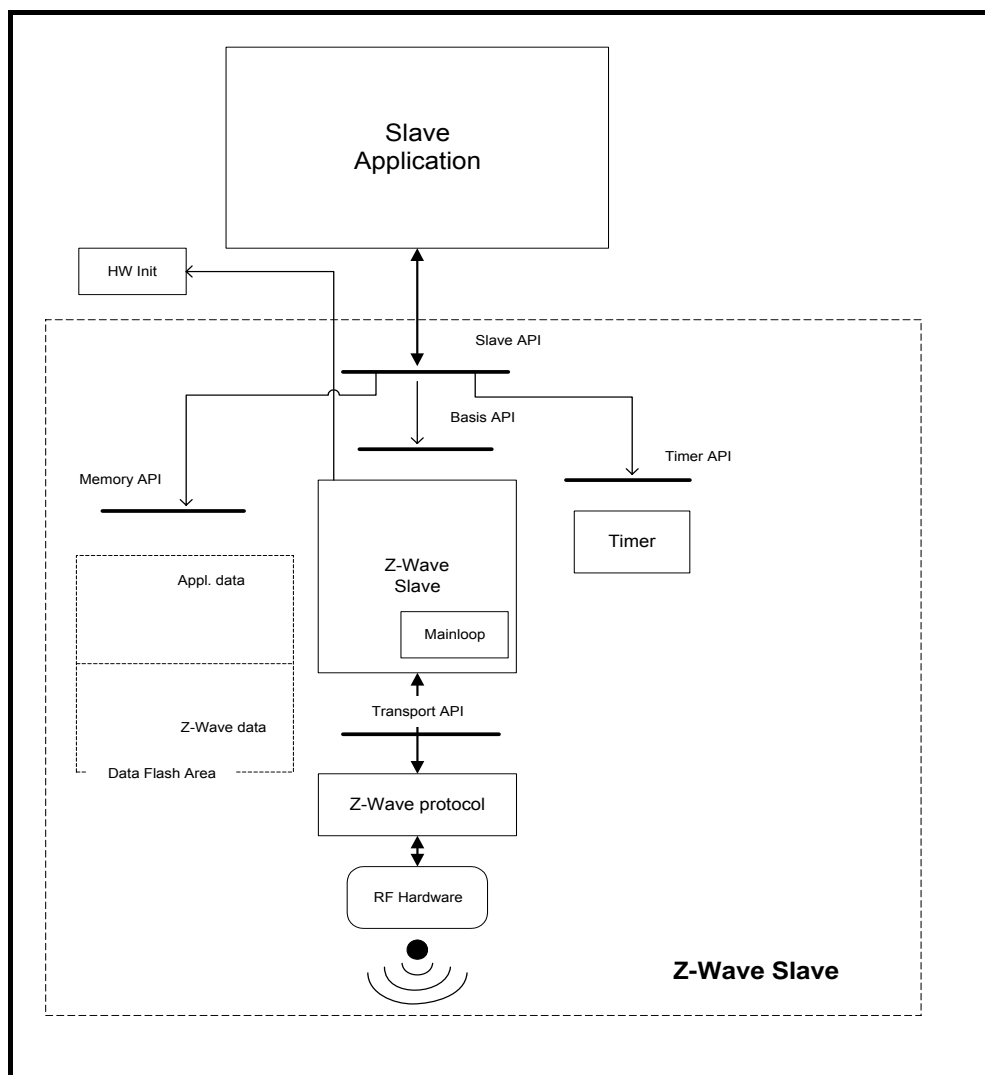


Figure 6. Routing slave node architecture

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

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

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

The routing algorithm in a routing slave tries to reach the destination depending on the transmit options as follows:

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

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

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

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

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

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

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

4.8.6 Z-Wave Enhanced Slave Node

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

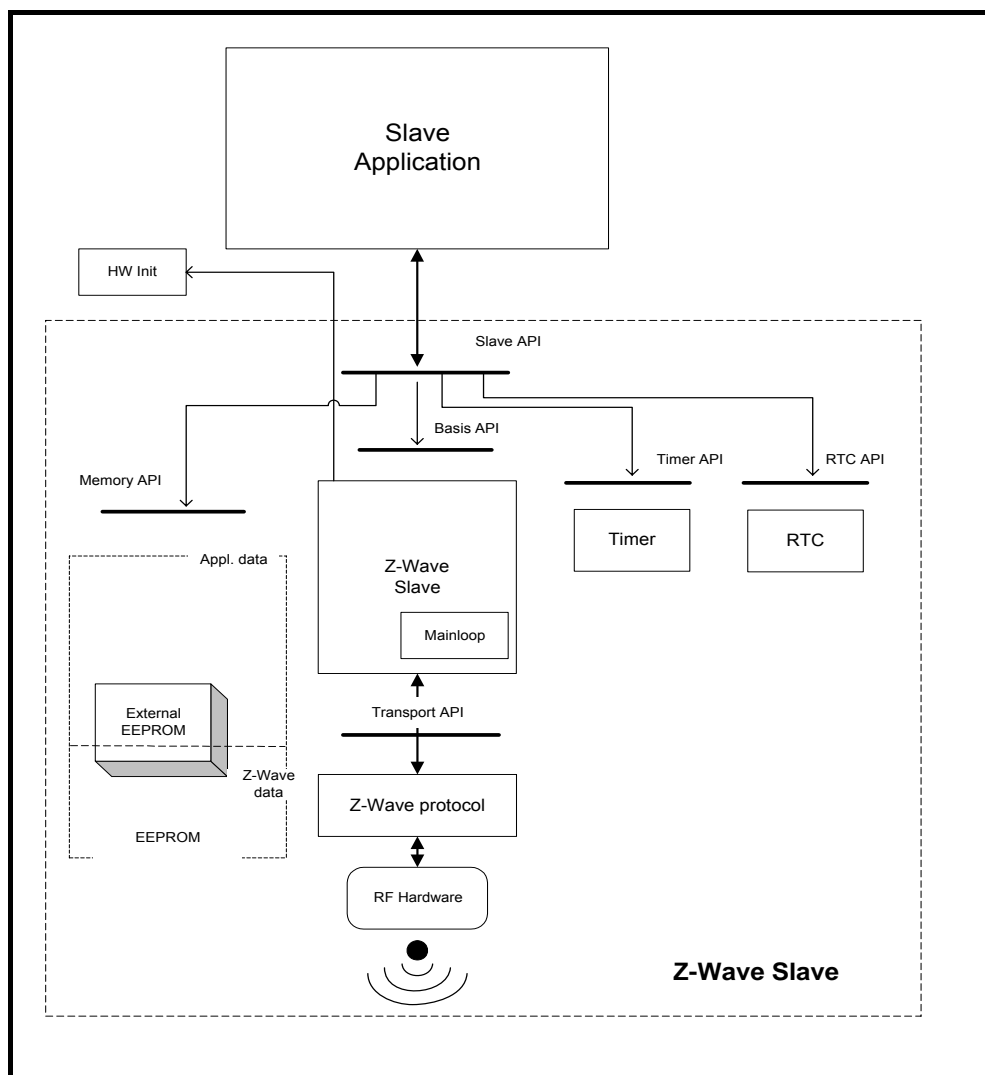


Figure 7. Enhanced slave node architecture

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

The routing algorithm in an enhanced slave tries to reach the destination depending on the transmit options as follows:

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

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

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

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

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

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

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

4.8.7 Z-Wave Enhanced 232 Slave Node

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

The routing algorithm in an enhanced 232 slave tries to reach the destination depending on the transmit options as follows:

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

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

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

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

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

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

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

4.8.8 Adding and Removing Nodes to/from the network

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

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

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

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

Adding a node normally.

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

Adding a new controller and make it the primary controller

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

Create a new primary controller

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

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

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.

4.8.9 The Automatic Network Update

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

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

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

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

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

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

5 Z-WAVE APPLICATION INTERFACES

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

5.1 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

| | Routing Slave | Enhanced Slave | Portable Controller | Static Controller | Installer Controller | Bridge Controller |
|--|----------------|----------------|---------------------|-------------------|----------------------|-------------------|
| Basic Functionality | | | | | | |
| Singlecast | X | X | X | X | X | X |
| Multicast | X | X | X | X | X | X |
| Broadcast | X | X | X | X | X | X |
| UART support | X | X | X | X | X | X |
| SPI support | - | - | - | - | - | - |
| ADC support | X | X | X | X | X | X |
| TRIAC control | X | X | X | X | X | X |
| PWM/HW timer support | X | X | X | X | X | X |
| Power management | X | X | X | - | X | - |
| SW timer support | X | X | X | X | X | X |
| Controller replication | - | - | X | X | X | X |
| Promiscuous mode | - | - | X | - | X | - |
| Random number generator | X | X | X | X | X | X |
| Able to act as NWI center | - | - | X | X | X | X |
| Able to be included via the NWI mechanism | X | X | X | X | X | X |
| Able to issue an explorer frame | X | X | X | X | X | X |
| Able to forward an explorer frame | X | X | - | X | - | - |
| | | | | | | |
| Memory Location | | | | | | |
| Non-volatile RAM in flash | X | - | - | - | - | - |
| Non-volatile RAM in EEPROM | - | X | X | X | X | X |
| | | | | | | |
| Network Management | | | | | | |
| Network router (repeater) | X | X | - | X | - | - |
| Assign routes to routing slave | - | - | X | X | X | X |
| Routing slave functionality | X | X | - | - | - | - |
| Access to routing table | - | - | X | - | X | - |
| Maintain virtual slave nodes | - | - | - | - | - | X ¹ |
| Able to be a FLiRS node | X | X | - | - | - | - |
| Able to beam when repeater | X | X | - | - | - | - |
| Able to create route containing beam | X ² | X ² | X | X | X | - |

¹ Only when secondary controller

² Only when return routes are assigned by a controller capable of creating routes containing beam

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

| | Routing Slave | Enhanced Slave | Portable Controller | Static Controller | Installer Controller | Bridge Controller |
|-------------------------------|---------------|----------------|---------------------|-------------------|----------------------|-------------------|
| Network Management | | | | | | |
| Controller replication | - | - | X | X | X | X |
| Controller shift | - | - | X ³ | X ¹ | X ¹ | X ¹ |
| Create new primary controller | - | - | - | - | - | - |
| Request network updates | - | - | - | - | - | - |
| Request rediscovery of a node | - | - | X ¹ | X ¹ | X ¹ | X ¹ |
| Remove failing nodes | - | - | X ¹ | X ¹ | X ¹ | X ¹ |
| Replace failing nodes | - | - | X ¹ | X ¹ | X ¹ | X ¹ |
| "I'm lost" – cry for help | X | X | - | - | - | - |
| "I'm lost" – provide help | - | - | - | - | - | - |
| Provide routing table info | - | - | X | X | X | X |
| | | | | | | |

³ Only when primary controller

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

| | Routing Slave | Enhanced Slave | Portable Controller | Static Controller | Installer Controller | Bridge Controller |
|-------------------------------|----------------|----------------|---------------------|-------------------|----------------------|-------------------|
| Network Management | | | | | | |
| Controller replication | - | - | X | X | X | X |
| Controller shift | - | - | X ¹ | X ⁴ | X ¹ | X ² |
| Create new primary controller | - | - | - | X ⁵ | - | X ³ |
| Request network updates | X | X | X | X | X | X |
| Request rediscovery of a node | - | - | X ¹ | X ¹ | X ¹ | X ¹ |
| Remove failing nodes | - | - | X ¹ | X ¹ | X ¹ | X ¹ |
| Replace failing nodes | - | - | X ¹ | X ¹ | X ¹ | X ¹ |
| Set static ctrl. to SUC | - | - | X ¹ | X ¹ | X ¹ | X ¹ |
| Work as SUC | - | - | - | X | - | X |
| Work as primary controller | | | X | X | X | X |
| "I'm lost" – cry for help | X | X | - | - | - | - |
| "I'm lost" – provide help | X ⁶ | X ³ | X ³ | X ⁷ | X ³ | X |
| Provide routing table info | - | - | X | X | X | X |
| | | | | | | |

⁴ Only when primary controller and not SUC

⁵ Only when SUC and not primary controller

⁶ Only if "always listening"

⁷ The library without repeater functionality cannot provide help or forward help requests.

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

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

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

¹ Only when primary/inclusion controller

² Only when primary controller

³ Only if "always listening"

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

5.1.1.4 Library Memory Usage

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

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

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

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

| Various header files | Description |
|----------------------|--|
| ZW020x.h | Inventra m8051w SFR and ISR defines for the Z-Wave ZW020x RF transceiver. |
| ZW030x.h | Inventra m8051w SFR and ISR defines for the Z-Wave ZW030x RF transceiver. |
| ZW_adcdriv_api.h | ADC functionality. |
| ZW_appltimer.h | PWM/Timer function |
| ZW_debug_api.h | Debugging functionality via serial port. |
| ZW_eep_addr.h | Define EEPROM_APPL_OFFSET, which is the offset address to store application data in non-volatile memory. Addresses between 0x0 and EEPROM_APPL_OFFSET is used by the protocol. |
| ZW_mem_api.h | EEPROM interface. |
| ZW_nodemask_api.h | Routines for manipulation of node ID lists organized as bit masks. |
| ZW_non_zero.h | Define none zero area of the SRAM (ZW0201/ZW0301 only). See also section 4.1 and 0 |
| ZW_power_api.h | ASIC power management functionality. |
| ZW_RF020x.h | Flash ROM RF table offset for the Z-Wave ZW020x |
| ZW_RF030x.h | Flash ROM RF table offset for the Z-Wave ZW030x |
| ZW_SerialAPI.h | Serial API interface with function ID defines etc. |
| ZW_sysdefs.h | CPU and clock defines. |
| ZW_timer_api.h | Timer functionality. |
| ZW_triac_api.h | TRIAC controller functionality. |
| ZW_typedefs.h | Common used defines (BYTE, WORD...). |
| ZW_uart_api.h | UART functionality. |

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.

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

5.3.1.1 ApplicationInitHW

BYTE ApplicationInitHW(BYTE bWakeupReason)

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

Defined in: ZW_basis_api.h

Return value:

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

Parameters:

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

Serial API (Not supported)

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)

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)

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)

5.3.1.5 ApplicationCommandHandler (Not Bridge Controller library)

In libraries not supporting promiscuous mode (see Table 4):

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

In libraries supporting promiscuous mode:

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

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

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

Defined in: ZW_basis_api.h

Parameters:

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

cmdLength IN Number of Command class bytes.

Serial API:

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

When a foreign frame is received in promiscuous mode:

ZW->HOST: REQ | 0xD1 | rxStatus | sourceNode | cmdLength | pCmd[] | destNode

The destNode parameter is only valid for singlecast frames.

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

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

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

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

Examples:

To set a device as Listening with Optional Functionality:

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

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

```
*deviceOptionsMask = APPLICATION_NODEINFO_NOT_LISTENING;
```

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

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

Defined in: ZW_basis_api.h

Parameters:

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

parmLength OUT Number of Command Class bytes.

Serial API:

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

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

The define **APPL_NODEPARM_MAX** in **serialappl.h** must be modified accordingly to the number of command classes to be notified.

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

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

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

Figure 8. Node Information frame structure on application level

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

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 receiving a Node Information Frame and the protocol is not in a state where it needs the node information.

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

Defined in: ZW_slave_api.h

Parameters:

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

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

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

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

bLen IN The length of the pCmd parameter.

Serial API:

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

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 given a node ID by the primary or an inclusion controller. |
| | | NOTE: At this point, it is not necessarily possible to route to the node because the range information from the node has not been received yet. |
| | | UPDATE_STATE_DELETE_DONE |
| | | A node has been deleted from the network |
| | | UPDATE_STATE_NODE_INFO_RECEIVED |
| | | A node has sent its node info either unsolicited or as a response to a ZW_RequestNodeInfo call |
| | | UPDATE_STATE_SUC_ID |
| | | The SUC node Id was updated |
| bNodeID | IN | The updated node's node ID (1..232). |
| pCmd | IN | Pointer of the updated node's node info. |
| bLen | IN | The length of the pCmd parameter. |

Serial API:

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

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

5.3.1.9 ApplicationCommandHandler_Bridge (Bridge Controller library only)

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

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

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

Defined in: ZW_controller_bridge_api.h

Parameters:

| | | |
|---------------|---|---|
| rxStatus IN | Frame header info: | |
| | RECEIVE_STATUS_ROUTED_BUSY xxxxxx1 | A response route is locked by the application |
| | RECEIVE_STATUS_LOW_POWER xxxxxx1x | Received at low output power level |
| | RECEIVE_STATUS_TYPE_SINGLE xxxx00xx | Received a single cast frame |
| | RECEIVE_STATUS_TYPE_BROAD xxxx01xx | Received a broadcast frame |
| | RECEIVE_STATUS_TYPE_MULTI xxxx10xx | Received a multicast frame |
| destNode IN | Command receiving Node ID. Either Bridge Controller Node ID or virtual slave Node ID. | |
| | If received frame is a MULTICAST frame then destNode is not valid and multi points to a multicast structure containing the destination nodes. | |
| sourceNode IN | Command sender Node ID. | |
| Multi IN | If received frame is, a multicast frame then multi points at the multicast Structure containing the destination Node IDs. | |
| pCmd IN | Payload from the received frame. The command class is the very first byte. | |
| cmdLength IN | Number of Command class bytes. | |

Serial API:

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

5.3.1.10 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[]

5.3.2 Z-Wave Basis API

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

5.3.2.1 ZW_ExploreRequestInclusion

BYTE ZW_ExploreRequestInclusion()

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

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

Defined in: ZW_basis_api.h

Return value:

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

Serial API

HOST->ZW: REQ | 0x5E

ZW->HOST: RES | 0x5E | retVal

5.3.2.2 ZW_GetProtocolStatus

BYTE ZW_GetProtocolStatus(void)

Macro: ZW_GET_PROTOCOL_STATUS()

Report the status of the protocol.

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

Defined in: ZW_basis_api.h

Return value:

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

Serial API

HOST->ZW: REQ | 0xBF

ZW->HOST: RES | 0xBF | retVal

5.3.2.3 ZW_GetRandomWord

BYTE ZW_GetRandomWord(BYTE *randomWord, BOOL bResetRadio)

Macro: ZW_GET_RANDOM_WORD(randomWord, bResetRadio)

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

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

Defined in: ZW_basis_api.h

Return value:

| | | |
|------|-------|---|
| BOOL | TRUE | If possible to generate random number. |
| | FALSE | If not possible e.g. RF not powered down. |

Parameters:

| | |
|----------------|---|
| randomWord OUT | Pointer to word variable, which should receive the random word. |
| bResetRadio IN | If TRUE the RF radio is reinitialized after generating the random word. |

Serial API

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

- Set the RF in powerdown prior to calling the ZW_GetRandomWord the first time, if not possible then return result to HOST.
- Call ZW_GetRandomWord until enough random bytes generated or ZW_GetRandomWord returns FALSE.
- Call ZW_GetRandomWord with bResetRadio = TRUE to reinitialize the radio.
- Call ZW_SetRFReceiveMode with TRUE if the serialAPI hardware is a listening device or with FALSE if it is a non-listening device.
- Return result to HOST.

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

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

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

| | |
|--------------------------|--|
| randomGenerationSuccess | TRUE if random bytes could be generated FALSE if no random bytes could be generated |
| noRandomBytesGenerated | Number of random numbers generated |
| noRandomBytesGenerated[] | Array of generated random bytes |

5.3.2.4 ZW_Poll

void ZW_Poll(void)

Macro: ZW_POLL

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

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

Defined in: ZW_basis_api.h

Serial API (Not supported)

5.3.2.5 ZW_Random

BYTE ZW_Random(void)

Macro: ZW_RANDOM()

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

Defined in: ZW_basis_api.h

Return value:

BYTE Random number (0 – 0xFF)

HOST->ZW: REQ | 0x1D

ZW->HOST: RES | 0x1D | rndNo

5.3.2.6 ZW_RFPowerLevelSet

BYTE ZW_RFPowerLevelSet(BYTE powerLevel)

Macro: ZW_RF_POWERLEVEL_SET(POWERLEVEL)

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

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

Defined in: ZW_basis_api.h

Parameters:

| | | |
|---------------|---|---|
| powerLevel IN | Powerlevel to use in RF transmission, valid values: | |
| | normalPower | Max power possible |
| | minus1dB | Normal power - 1dB (mapped to minus2dB ⁸) |
| | minus2dB | Normal power - 2dB |
| | minus3dB | Normal power - 3dB (mapped to minus4dB) |
| | minus4dB | Normal power - 4dB |
| | minus5dB | Normal power - 5dB (mapped to minus6dB) |
| | minus6dB | Normal power - 6dB |
| | minus7dB | Normal power - 7dB (mapped to minus8dB) |
| | minus8dB | Normal power - 8dB |
| | minus9dB | Normal power - 9dB (mapped to minus10dB) |

Return value:

BYTE The powerlevel set.

Serial API (Serial API protocol version 4):

HOST->ZW: REQ | 0x17 | powerLevel

ZW->HOST: RES | 0x17 | retVal

⁸ 200/300 Series support only -2dB power level steps

5.3.2.7 ZW_RFPowerLevelGet

BYTE ZW_RFPowerLevelGet(void)

Macro: ZW_RF_POWERLEVEL_GET()

Get the current power level used in RF transmitting.

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

Defined in: ZW_basis_api.h

Return value:

BYTE The power level currently in effect during
RF transmissions.

Serial API

HOST->ZW: REQ | 0xBA

ZW->HOST: RES | 0xBA | powerlevel

5.3.2.8 ZW_RequestNetWorkUpdate

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

Macro: ZW_REQUEST_NETWORK_UPDATE (func)

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

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

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

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

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

Defined in: ZW_controller_api.h and ZW_slave_routing_api.h

Return value:

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

Parameters:

completedFunc Transmit complete call back.
IN

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

5.3.2.9 ZW_RFPowerlevelRediscoverySet

void ZW_RFPowerlevelRediscoverySet(BYTE bNewPower)

Macro: ZW_RF_POWERLEVEL_REDISCOVERY_SET(bNewPower)

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

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

Defined in: ZW_basis_api.h

Parameters:

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

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

Serial API:

HOST->ZW: REQ | 0x1E | powerLevel

⁹ 200/300 Series support only -2dB power level steps

5.3.2.10 ZW_SendNodeInformation

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

Macro: ZW_SEND_NODE_INFO(node,option,func)

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

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

NOTE: ZW_SendNodeInformation uses the transmit queue in the API, so using other transmit functions before the complete callback has been called by the API might fail.

Defined in: ZW_basis_api.h

Return value:

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

Parameters:

| | |
|------------------|--|
| destNode IN | Destination Node ID (NODE_BROADCAST == all nodes) |
| txOptions IN | Transmit option flags. (see ZW_SendData) |
| completedFunc IN | Transmit completed call back function |

Callback function Parameters:

| | |
|-------------|---------------------------|
| txStatus IN | (see ZW_SendData) |
|-------------|---------------------------|

Serial API:

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

ZW->HOST: RES | 0x12 | retVal

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

5.3.2.11 ZW_SendTestFrame

```

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

```

Macro: ZW_SEND_TEST_FRAME(nodeID, power, func)

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

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

Defined in: ZW_basis_api.h

Parameters:

| | | |
|---------------|--|--|
| nodeID IN | Node ID on the node ID (1..232) the test frame should be transmitted to. | |
| powerLevel IN | Powerlevel to use in RF transmission, valid values: | |
| | normalPower | Max power possible |
| | minus1dB | Normal power - 1dB (mapped to minus2dB ¹⁰) |
| | minus2dB | Normal power - 2dB |
| | minus3dB | Normal power - 3dB (mapped to minus4dB) |
| | minus4dB | Normal power - 4dB |
| | minus5dB | Normal power - 5dB (mapped to minus6dB) |
| | minus6dB | Normal power - 6dB |
| | minus7dB | Normal power - 7dB (mapped to minus8dB) |
| | minus8dB | Normal power - 8dB |
| | minus9dB | Normal power - 9dB (mapped to minus10dB) |
| func IN | Call back function called when done. | |

Callback function Parameters:

¹⁰ 200/300 Series support only -2dB power level steps

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

5.3.2.12 ZW_SetExtIntLevel

void ZW_SetExtIntLevel(BYTE intSrc, BOOL triggerLevel)

Macro: ZW_SET_EXT_INT_LEVEL(SRC, TRIGGER_LEVEL)

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

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

Defined in: ZW_basis_api.h

Parameters:

intSrc IN The external interrupt valid values:

ZW_INT0 External interrupt 0 (PIN P1_6)

ZW_INT1 External interrupt 1 (PIN P1_7)

triggerLevel IN The external interrupt trigger level:

TRUE Set the interrupt trigger to high level
/Rising edge

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

Serial API

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

5.3.2.13 ZW_SetPromiscuousMode (Not Bridge Controller library)

void ZW_SetPromiscuousMode(BOOL state)

Macro: ZW_SET_PROMISCUOUS_MODE(state)

ZW_SetPromiscuousMode Enable / disable the promiscuous mode.

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

Promiscuously received frames are delivered to the application via the ApplicationCommandHandler callback function (see section 5.3.1.5).

Defined in: ZW_basis_api.h

Parameters:

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

Serial API:

HOST->ZW: REQ | 0xD0 | state

See section 5.3.1.5 for callback syntax when a frame has been promiscuously received.

5.3.2.14 ZW_SetRFReceiveMode

BYTE ZW_SetRFReceiveMode(BYTE mode)

Macro: ZW_SET_RX_MODE(mode)

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

Defined in: ZW_basis_api.h

Return value:

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

Parameters:

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

Serial API

HOST->ZW: REQ | 0x10 | mode

ZW->HOST: RES | 0x10 | retVal

5.3.2.15 ZW_SetSleepMode

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. Notice that ZW_SetSleepMode() doesn't go into sleep mode immediately, it sets a sleep state flag and return. Then at a later point when the protocol is idle (stopped RF transmission etc.) the CPU will power down.

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

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

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.

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.

ZW_FREQUENTLY_LISTENING_MODE

This mode make the module enter a Frequently Listening mode where the module will wakeup for a few milliseconds every 1000ms or 250ms and check for radio transmissions to the module (See 5.3.1.6 for details about selecting wakeup speed). The application will only be woken up if there is incoming RF traffic or if the intEnable or beamCount parameters are used.

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

Serial API

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

5.3.2.16 ZW_Type_Library

BYTE ZW_Type_Library(void)

Macro: ZW_TYPE_LIBRARY()

Get the Z-Wave library type.

Defined in: ZW_basis_api.h

Return value:

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

Serial API

HOST->ZW: REQ | 0xBD

ZW->HOST: RES | 0xBD | retVal

5.3.2.17 ZW_Version

BYTE ZW_Version(BYTE *buffer)

Macro: ZW_VERSION(buffer)

Get the Z-Wave basis API library version.

Defined in: ZW_basis_api.h

Parameters:

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

Z-Wave x.yy

where x.yy is the library version.

Return value:

BYTE Returns the library type as one of the following:

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

Serial API:

HOST->ZW: REQ | 0x15

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

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

HOST->ZW: REQ | 0x02

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

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

Nodes[29] is a node bitmask.

Capabilities flag:

Bit 0: 0 = Controller API; 1 = Slave API

Bit 1: 0 = Timer functions not supported; 1 = Timer functions supported.

Bit 2: 0 = Primary Controller; 1 = Secondary Controller

Bit 3-7: reserved

The chip used can be determined as follows:

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

Timer functions are: TimerStart, TimerRestart and TimerCancel.

5.3.2.18 ZW_VERSION_MAJOR / ZW_VERSION_MINOR / ZW_VERSION_BETA

Macro: ZW_VERSION_MAJOR/ZW_VERSION_MINOR/ ZW_VERSION_BETA

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

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

Defined in: ZW_basis_api.h

Serial API (Not supported)

5.3.2.19 ZW_WatchDogEnable

void ZW_WatchDogEnable(void)

Macro: ZW_WATCHDOG_ENABLE()

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

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

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

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

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

Defined in: ZW_basis_api.h

Serial API

HOST->ZW: REQ | 0xB6

5.3.2.20 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

5.3.2.21 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

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))
```

NOTE: Only libraries without manual routing functionality support ZW_SendData.

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

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

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

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

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

Table 9. Transmit options behavior for portable and installer libraries

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

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

To enable on-demand route resolution a new transmit option TRANSMIT_OPTION_EXPLORE must be appended to the well known send API calls. This instructs the protocol to transmit the frame as an explore frame to the destination node if source routing fails. It is also possible to specify the maximum number of source routing attempts before the explorer frame kicks in using the API call ZW_SetRoutingMAX. Default value is five with respect to maximum number of source routing attempts. A ZDK 4.5 controller uses the routing algorithm from 5.02 to address nodes from ZDK's not supporting explorer frame. The routing algorithm from 5.02 ignores the transmit option TRANSMIT_OPTION_EXPLORE flag and maximum number of source routing attempts value. Notice that an explorer frame cannot wake up FLiRS nodes.

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

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

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

NOTE: Always 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 |
|------|-------|----------------------------|

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. In case it is a singlecast, which piggyback on an explorer frame overhead is 8 bytes leaving minimum 46 bytes for the payload. The payload must be minimum one byte. |
| txOptions IN | Transmit option flags: | |
| | TRANSMIT_OPTION_LOW_POWER | Transmit at low output power level (1/3 of normal RF range). |
| | TRANSMIT_OPTION_NO_ROUTE | Only send this frame directly, even if a response route exist |
| | TRANSMIT_OPTION_ACK | Request acknowledge from destination node. |
| | TRANSMIT_OPTION_AUTO_ROUTE | <u>Controllers:</u> Request retransmission via repeater nodes (at normal output power level). Number of max routes can be set using ZW_SetRoutingMax <u>Routing and Enhanced Slaves:</u> Send the frame to nodeID using the return routes assigned for nodeID to the routing/enhanced slave, if no routes are valid then transmit directly to nodeID (if nodeID = NODE_BROADCAST then the frame will be a BROADCAST). If return routes exists and the nodeID = NODE_BROADCAST then the frame will be transmitted to all assigned return route destinations. If nodeID != NODE_BROADCAST then the frame will be transmitted via the assigned return routes for nodeID. |
| | TRANSMIT_OPTION_EXPLORE | Transmit frame as an explore frame if everything else fails. |

completedFunc Transmit completed call back function

Callback function Parameters:

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

Serial API:

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

ZW->HOST: RES | 0x13 | RetVal

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

5.3.3.2 ZW_SendData_Generic

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

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

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

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

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

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

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

To enable on-demand route resolution a new transmit option TRANSMIT_OPTION_EXPLORE must be appended to the well known send API calls. This instruct the protocol to transmit the frame as an explore frame to the destination node if source routing fails. It is also possible to specify the maximum number of source routing attempts before the explorer frame kicks in using the API call ZW_SetRoutingMAX. Default value is five with respect to maximum number of source routing attempts. A ZDK 4.5 controller uses the routing algorithm from 5.02 to address nodes from ZDK's not supporting explorer frame. The routing algorithm from 5.02 ignores the transmit option TRANSMIT_OPTION_EXPLORE flag and maximum number of source routing attempts value. Notice that an explorer frame cannot wake up FLiRS nodes.

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

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

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

Defined in: ZW_transport_api.h

Return value:

BYTE FALSE If transmit queue overflow

Parameters:

| | | |
|---------------|--|---|
| nodeID IN | Destination node ID (NODE_BROADCAST == all nodes) | The frame will also be transmitted in case the source node ID is equal destination node ID |
| pData IN | Data buffer pointer | |
| dataLength IN | Data buffer length | The maximum size of a frame is 64 bytes. The protocol header and checksum takes 10 bytes in a single cast or broadcast frame leaving 54 bytes for the payload. In case it is a routed singlecast the source routing info takes up to 6 bytes depending on the number of hops leaving minimum 48 bytes for the payload. In case it is a singlecast, which piggyback on an explorer frame overhead is 8 bytes leaving minimum 46 bytes for the payload. The payload must be minimum one byte. |
| txOptions IN | Transmit option flags: | |
| | TRANSMIT_OPTION_LOW_POWER | Transmit at low output power level (1/3 of normal RF range). |
| | TRANSMIT_OPTION_NO_ROUTE | Only send this frame directly, even if a response route exist |
| | TRANSMIT_OPTION_EXPLORE | Transmit frame as an Explore frame if all else fails. |
| | TRANSMIT_OPTION_ACK | Request acknowledge from destination node. |
| | TRANSMIT_OPTION_AUTO_ROUTE | <u>Controllers:</u> Request retransmission via repeater nodes (at normal output power level). Number of max routes can be set using ZW_SetRoutingMax <u>Routing and Enhanced Slaves:</u> Send the frame to nodeID using the return routes assigned for nodeID to the routing/enhanced slave, if no routes are valid then transmit directly to nodeID (if nodeID = NODE_BROADCAST then the frame will be a BROADCAST). |

| | | | | | | |
|---------------|---|---|------------|------------|------------|--|
| | | If return routes exists and the nodeID = NODE_BROADCAST then the frame will be transmitted to all assigned return route destinations. If nodeID != NODE_BROADCAST then the frame will be transmitted via the assigned return routes for nodeID. | | | | |
| pRoute IN | <p>Pointer to byte array (4 bytes) containing up to 4 hop node IDs. Zero indicates end of source route if the route is shorter than 4 hops. NULL indicates that the internal source routing mechanism is used.</p> <p>Format of data:</p> <table><tr><td>Repeater 1</td></tr><tr><td>Repeater 2</td></tr><tr><td>Repeater 3</td></tr><tr><td>Repeater 4</td></tr></table> | Repeater 1 | Repeater 2 | Repeater 3 | Repeater 4 | <p>The call will only try the proposed source route when pRoute is different from NULL. Ignores the transmit options TRANSMIT_OPTION_AUTO_ROUTE if pRoute is different from NULL.</p> <p>NOTE: On controllers the protocol will calculate the speed used for the route and in routing/enhanced slaves the route will always use 9.6kbps</p> |
| Repeater 1 | | | | | | |
| Repeater 2 | | | | | | |
| Repeater 3 | | | | | | |
| Repeater 4 | | | | | | |
| completedFunc | Transmit completed call back function | | | | | |

Callback function Parameters:

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

Serial API:

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

ZW->HOST: RES | 0x19 | RetVal

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

5.3.3.3 ZW_SendData_Bridge

```

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

```

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

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

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

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

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

To enable on-demand route resolution a new transmit option `TRANSMIT_OPTION_EXPLORE` must be appended to the well known send API calls. This instruct the protocol to transmit the frame as an explore frame to the destination node if source routing fails. It is also possible to specify the maximum number of source routing attempts before the explorer frame kicks in using the API call `ZW_SetRoutingMAX`. Default value is five with respect to maximum number of source routing attempts. A ZDK 4.5 controller uses the routing algorithm from 5.02 to address nodes from ZDK's not supporting explorer frame. The routing algorithm from 5.02 ignores the transmit option `TRANSMIT_OPTION_EXPLORE` flag and maximum number of source routing attempts value. Notice that an explorer frame cannot wake up FLIRS nodes.

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

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

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

Defined in: ZW_transport_api.h

Return value:

| | | |
|------|-------|----------------------------|
| BYTE | FALSE | If transmit queue overflow |
|------|-------|----------------------------|

Parameters:

| | | |
|---------------|---|--|
| srcNodeID IN | Source node ID. Valid values: | |
| | NODE_BROADCAST = Bridge Controller NodeID. | |
| | Bridge Controller NodeID. | |
| | Virtual Slave NodeID (only existing Virtual Slave NodeIDs). | |
| destNodeID IN | Destination node ID (NODE_BROADCAST == all nodes) | The frame will also be transmitted in case the source node ID is equal destination node ID |
| pData IN | Data buffer pointer | |
| dataLength IN | Data buffer length | The maximum size of a frame is 64 bytes. The protocol header and checksum takes 10 bytes in a single cast or broadcast frame leaving 54 bytes for the payload. In case it is a routed single cast the source routing info takes up to 6 bytes depending on the number of hops leaving minimum 48 bytes for the payload. In case it is a singlecast, which piggyback on an explorer frame overhead is 8 bytes leaving minimum 46 bytes for the payload. The payload must be minimum one byte. |
| txOptions IN | Transmit option flags: | |
| | TRANSMIT_OPTION_LOW_POWER | Transmit at low output power level (1/3 of normal RF range). |
| | TRANSMIT_OPTION_NO_ROUTE | Only send this frame directly, even if a response route exist |
| | TRANSMIT_OPTION_EXPLORE | Transmit frame as an Explore frame if all else fails |
| | TRANSMIT_OPTION_ACK | Request acknowledge from destination node. |
| | TRANSMIT_OPTION_AUTO_ROUTE | Request retransmission via repeater nodes (at normal output power level). |
| pRoute IN | Pointer to byte array (4 bytes) containing up to 4 hop node IDs. Zero indicates end of source route. NULL indicates that the internal source routing mechanism is used. | The call will only try the proposed source route when pRoute is different from NULL. Ignores the transmit options TRANSMIT_OPTION_AUTO_ROUTE if pRoute is different from NULL. |

completedFunc Transmit completed call back function

Callback function Parameters:

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

Serial API:

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

ZW->HOST: RES | 0xA9 | RetVal

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

5.3.3.4 ZW_SendDataMeta_Generic

```

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

```

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

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

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

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

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

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

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

Defined in: ZW_transport_api.h

Return value:

BYTE FALSE

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

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: | <p>The maximum size of a frame is 64 bytes. The protocol header and checksum takes 10 bytes in a single cast or broadcast frame leaving 54 bytes for the payload. In case it is a routed singlecast the source routing info takes up to 6 bytes depending on the number of hops leaving minimum 48 bytes for the payload. In case it is a singlecast, which piggyback on an explorer frame overhead is 8 bytes leaving minimum 46 bytes for the payload. The payload must be minimum one byte.</p> <p>TRANSMIT_OPTION_LOW_POWER</p> <p>Transmit at low output power level (1/3 of normal RF range).</p> <p>TRANSMIT_OPTION_EXPLORE</p> <p>Transmit frame as an Explore frame if all else fails</p> <p>TRANSMIT_OPTION_ACK</p> <p>Request the destination node to acknowledge the frame</p> <p>TRANSMIT_OPTION_AUTO_ROUTE</p> <p><u>Controllers:</u> Request retransmission via repeater nodes (at normal output power level). Number of max routes can be set using ZW_SetRoutingMax</p> <p><u>Routing and Enhanced Slaves:</u> Send the frame to nodeID using the return routes assigned for nodeID to the routing/enhanced slave, if no routes are valid then transmit directly to nodeID (if nodeID = NODE_BROADCAST then the frame will be a BROADCAST). If return routes exists and the nodeID = NODE_BROADCAST then the frame will be transmitted to all assigned return route destinations. If nodeID != NODE_BROADCAST then the frame will be transmitted via the assigned return routes for nodeID.</p> |
| pRoute IN | Pointer to byte array (4 bytes) containing up to 4 hop node IDs. Zero indicates end of source route. NULL indicates that the internal source routing mechanism is | The call will only try the proposed source route when pRoute is different from NULL. Ignores the transmit options TRANSMIT_OPTION_AUTO_ROUTE if |

used.

pRoute is different form NULL.

completedFunc Transmit completed call back function

Callback function Parameters:

txStatus IN (see **ZW_SendData**)

Serial API (Serial API protocol version 4):

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

ZW->HOST: RES | 0x1A | RetVal

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

5.3.3.5 ZW_SendDataMeta_Bridge

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

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

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

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

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

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

Defined in: ZW_transport_api.h

Return value:

BYTE FALSE

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

Parameters:

| | | |
|---------------|----|--|
| srcNodeID | IN | Source node ID. Valid values: NODE_BROADCAST = Bridge Controller NodeID. Bridge Controller NodeID. Virtual Slave NodeID (only existing Virtual Slave NodeIDs). |
| destNodeID | IN | Destination node ID Node to send Meta data to. Should be 40kbit/s capable |
| pData | IN | Data buffer pointer Pointer to data buffer. |
| dataLength | IN | Data buffer length Length of buffer |
| txOptions | IN | Transmit option flags: The maximum size of a frame is 64 bytes. The protocol header and checksum takes 10 bytes in a single cast or broadcast frame leaving 54 bytes for the payload. In case it is a routed single cast the source routing info takes up to 6 bytes depending on the number of hops leaving minimum 48 bytes for the payload. In case it is a singlecast, which piggyback on an explorer frame overhead is 8 bytes leaving minimum 46 bytes for the payload. The payload must be minimum one byte. TRANSMIT_OPTION_LOW_POWER Transmit at low output power level (1/3 of normal RF range). TRANSMIT_OPTION_EXPLORE Transmit frame as an Explore frame if all else fails TRANSMIT_OPTION_ACK Request the destination node to acknowledge the frame TRANSMIT_OPTION_AUTO_ROUTE Request retransmission on single cast frames via repeater nodes (at normal output power level) |
| pRoute | IN | Pointer to byte array (4 bytes) containing up to 4 hop node IDs. Zero indicates end of source route. NULL indicates that the internal source routing mechanism is used. The call will only try the proposed source route when pRoute is different from NULL. Ignores the transmit options TRANSMIT_OPTION_AUTO_ROUTE if pRoute is different from NULL. |
| completedFunc | | Transmit completed call back function |

Callback function Parameters:

txStatus IN (see **ZW_SendData**)

Serial API (Serial API protocol version 4):

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

ZW->HOST: RES | 0xAA | RetVal

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

5.3.3.6 ZW_SendDataMulti

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

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

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

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

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

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

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

Defined in: ZW_transport_api.h

Return value:

| | | |
|------|-------|----------------------------|
| BYTE | FALSE | If transmit queue overflow |
|------|-------|----------------------------|

Parameters:

| | | |
|---------------|---|---|
| pNodeIDList | IN List of destination node ID's | This is a fixed length bit-mask. |
| Pdata | IN Data buffer pointer | |
| DataLength | IN Data buffer length | The maximum size of a packet is 64 bytes. The protocol header, multicast addresses and checksum takes 39 bytes in a multicast frame leaving 25 bytes for the payload. In case routed single casts follow multicast the source routing info takes up to 6 bytes depending on the number of hops leaving minimum 19 bytes for the payload. In case it is a singlecast, which piggyback on an explorer frame overhead is 8 bytes leaving minimum 17 bytes for the payload. The payload must be minimum one byte. |
| TxOptions | IN Transmit option flags: | |
| | TRANSMIT_OPTION_LOW_POWER | Transmit at low output power level (1/3 of normal RF range). |
| | TRANSMIT_OPTION_EXPLORE | If TRANSMIT_OPTION_ACK is set the will make the node try sending as an Explore frame if all else fails when doing the single cast transmits |
| | TRANSMIT_OPTION_ACK | The multicast frame will be followed by a number of single cast frames to each of the destination nodes and request acknowledge from each destination node. |
| | TRANSMIT_OPTION_AUTO_ROUTE (Controller API only) | Request retransmission on single cast frames via repeater nodes (at normal output power level) |
| completedFunc | Transmit completed call back function | |

Callback function Parameters:

txStatus IN (see **ZW_SendData**)

Serial API:

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

ZW->HOST: RES | 0x14 | RetVal

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

5.3.3.7 ZW_SendDataMulti_Bridge

```

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

```

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

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

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

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

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

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

Defined in: ZW_transport_api.h

Return value:

| | | |
|------|-------|----------------------------|
| BYTE | FALSE | If transmit queue overflow |
|------|-------|----------------------------|

Parameters:

| | | | |
|---------------|----|---|--|
| srcNodeID | IN | Source node ID. Valid values: NODE_BROADCAST = Bridge Controller NodeID. Bridge Controller NodeID. Virtual Slave NodeID (only existing Virtual Slave NodeIDs). | |
| pNodeIDList | IN | List of destination node ID's | This is a fixed length bit-mask. |
| Pdata | IN | Data buffer pointer | |
| DataLength | IN | Data buffer length | The maximum size of a packet is 64 bytes. The protocol header for a multicast depends on the destination node IDs leaving between 25-53 bytes for the payload. The size of the protocol header and checksum for a multicast frame is: $((\text{MaxNodeID} - (\text{MinNodeID} - 1) \& 0xE0) + 7) \gg 3 + 10$ where MaxNodeID is the largest node ID number and MinNodeID is the smallest. |
| TxOptions | IN | Transmit option flags: TRANSMIT_OPTION_LOW_POWER TRANSMIT_OPTION_EXPLORE TRANSMIT_OPTION_ACK TRANSMIT_OPTION_AUTO_ROUTE | Transmit at low output power level (1/3 of normal RF range). If TRANSMIT_OPTION_ACK is set the will make the node try sending as an Explore frame if all else fails when doing the single cast transmits 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. Request retransmission on single cast frames via repeater nodes (at normal output power level) |
| completedFunc | | Transmit completed call back function | |

Callback function Parameters:

txStatus IN (see **ZW_SendData**)

Serial API:

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

ZW->HOST: RES | 0xAB | RetVal

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

5.3.3.8 ZW_SendDataAbort

void ZW_SendDataAbort()

Macro: ZW_SEND_DATA_ABORT

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

Defined in: ZW_transport_api.h

Serial API:

HOST->ZW: REQ | 0x16

5.3.3.9 ZW_SendConst

void ZW_SendConst(void)

Macro: ZW_SEND_CONST()

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

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

Defined in: ZW_transport_api.h

Serial API (Not supported)

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 ZW0201/ZW0301 TRIAC Controller.

5.3.4.1 TRIAC_Init

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

Macros:

ZW_TRIAC_INIT

ZW_TRIAC_INIT_2_WIRE

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

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

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

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

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)

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 (0-100) or sets the light level to lightLevel (0-100) if the ZW_TRIAC_LIGHT_SET_LEVEL macro is used. This is done for the mains frequency selected in the TRIAC_Init function call (50/60Hz).

Defined in: ZW_triac_api.h

Parameters:

dimLevel IN Level (0...100)

Serial API (Not supported)

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)

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 stop 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

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)

5.3.6 Z-Wave PWM API

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

5.3.6.1 ZW_PWMSetup

BYTE ZW_PWMSetup (BYTE bValue)

Macro: ZW_PWM_SETUP(value)

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

Parameters:

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

Serial API (Not supported)

5.3.6.2 ZW_PWMPrescale

BYTE ZW_PWMPrescale(BYTE bValueMSB, BYTE bValueLSB)

Macro ZW_PWM_PRESCALE(msb,lsb)

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

PWM mode:

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

Low time of PWM signal: $t_{lPWM} = (\text{lsb} * \text{prescaler}) / \text{CPU_FREQ}$

Total period of PWM signal: $T_{PWM} = t_{hPWM} + t_{lPWM}$

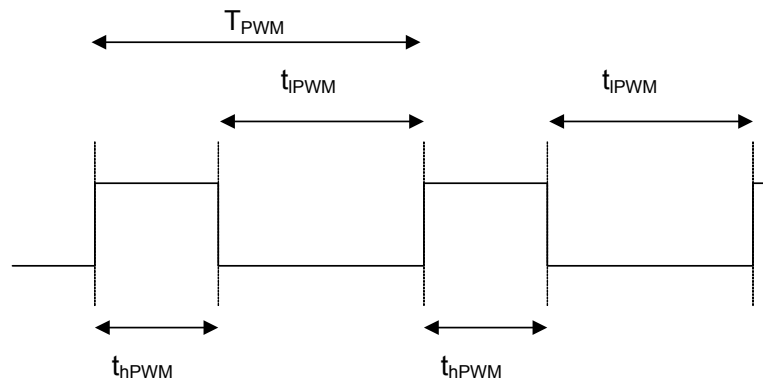


Figure 9. PWM waveform

Timer mode (Interrupt period):

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

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

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)

5.3.6.3 ZW_PWMClearInterrupt

BYTE ZW_PWMClearInterrupt(void)

Macro ZW_PWM_CLEAR_INTERRUPT()

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

Defined in: ZW_apptimer_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 GP Timer interrupt (ISR number: ZW_PWM).

Defined in: ZW_apptimer_api.h

Parameters:

| | | |
|-----------|-------|---------------------|
| bValue IN | TRUE | Interrupt enabled. |
| | FALSE | Interrupt disabled. |

Serial API (Not supported)

5.3.7 Z-Wave Memory API

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

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 access the external EEPROM. The SPI interface related pins CLK, MOSI, and MISO are only free as GPIO for the application on a routing slave.

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

NOTE: The CPU halts while the API is writing to flash memory, so care should be taken not to write to flash too 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 interested in reading the Node ID.

Defined in: ZW_mem_api.h

Parameters:

pHomeID OUT Home-ID pointer

pNodeID OUT Node-ID pointer

Serial API:

HOST->ZW: REQ | 0x20

ZW->HOST: RES | 0x20 | HomeID(4 bytes) | NodeID

5.3.7.2 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 from 0x0000.

Serial API:

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

ZW->HOST: RES | 0x21 | RetVal

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.).

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

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

Defined in: ZW_mem_api.h

Return value:

BYTE FALSE If write buffer full.

Parameters:

offset IN Application area offset from 0x0000.

data IN Data to store

Serial API:

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

ZW->HOST: RES | 0x22 | RetVal

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 from 0x0000.

buffer IN Buffer pointer

length IN Number of bytes to read

Serial API:

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

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

5.3.7.5 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 in the non-volatile memory.

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

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

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

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

Defined in: ZW_mem_api.h

Return value:

BYTE FALSE If the buffer put queue is full.

Parameters:

offset IN Application area offset from 0x0000.

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

length IN Number of bytes to read

func IN Buffer write completed function pointer

Serial API:

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

ZW->HOST: RES | 0x24 | RetVal

ZW->HOST: REQ | 0x24 | funcID

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 and when it is more than 200ms the buffer was accessed. This function can be used to write data immediately to FLASH without waiting for the RF to be idle.

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

Defined in: ZW_mem_api.h

Serial API (Not supported)

5.3.8 Z-Wave ADC API

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

5.3.8.1 ADC_Off

void ADC_Off()

Macro: ZW_ADC_OFF

Call turns the power off to the ADC unit.

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

Defined in: ZW_adcdriv_api.h

Serial API (Not supported)

5.3.8.2 ADC_Start

void ADC_Start()

Macro: ZW_ADC_START

Start the conversion process.

Defined in: ZW_adcdriv_api.h

Serial API (Not supported)

5.3.8.3 ADC_Stop

void ADC_Stop()

Macro: ZW_ADC_STOP

Stop the conversion process.

Defined in: ZW_adcdriv_api.h

Serial API (Not supported)

5.3.8.4 ADC_Init

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

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

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

The ADC unit has 5 multiplexed inputs. The Upper reference voltage can be set to be VCC, internal bandgap 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. |

| | | |
|-----------|--|--|
| 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)

5.3.8.5 ADC_SelectPin

ADC_SelectPin(BYTE adcPin);

Macro:

ZW_ADC_SELECT_AD1 -select pin 1 as the ADC input

ZW_ADC_SELECT_AD2 - select pin 2 as the ADC input

ZW_ADC_SELECT_AD3 - select pin 3 as the ADC input

ZW_ADC_SELECT_AD4 - select pin 4 as the ADC input

Select a pin to use as the active ADC input.

Defined in: ZW_adcdriv_api.h

Parameters:

adcPin IN The pin to use as ADC input:

ADC_PIN_1

ADC_PIN_2

ADC_PIN_3

ADC_PIN_4

Serial API (Not supported)

5.3.8.6 ADC_Buf

ADC_Buf (BYTE enable);

Macro:

ZW_ADC_BUFFER_ENABLE - Enable the input buffer.

ZW_ADC_BUFFER_DISABLE - Disable the input buffer.

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

Defined in: ZW_adcdriv_api.h

Parameters:

enable IN Switch the input buffer on/off:

TRUE

Switch the input buffer on.

FALSE

Switch the input buffer off.

Serial API (Not supported)

5.3.8.7 ADC_SetAZPL

ADC_SetAZPL (BYTE azpl);

Macro: ZW_ADC_SET_AZPL(PERIOD)

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

Defined in: ZW_adcdriv_api.h

Parameters:

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

Serial API (Not supported)

5.3.8.8 ADC_SetResolution

ADC_SetResolution (BYTE reso);

Macro:

ZW_ADC_RESOLUTION_8 - Set the ADC resolution to 8 bit.

ZW_ADC_RESOLUTION_12 - Set the ADC resolution to 12 bit.

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

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)

5.3.8.9 ADC_SetThresMode

ADC_SetThresMode (BYTE thresMode);

Macro:

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

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

Set the ADC threshold type.

Defined in: ZW_adcdriv_api.h

Parameters:

thresMode IN The ADC threshold mode.

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

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

Serial API (Not supported)

5.3.8.10 ADC_SetThres

void ADC_SetThres(WORD threshHold)

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:

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

Serial API (Not supported)

5.3.8.11 ADC_Int

void ADC_Int(BYTE enable)

Macro:

ZW_ADC_INT_ENABLE

ZW_ADC_INT_DISABLE

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

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

Defined in: ZW_adcdriv_api.h

Parameters:

enable IN The start of the ADC interrupt routine.

TRUE Enables ADC interrupt.

FALSE Disables ADC interrupt.

Serial API (Not supported)

5.3.8.12 ADC_IntFlagClr

void ADC_IntFlagClr()

Macro: ZW_ADC_CLR_FLAG

Clear the ADC interrupt flag.

Defined in: ZW_adcdriv_api.h

Serial API (Not supported)

5.3.8.13 ADC_GetRes

WORD ADC_GetRes()

Macro: ZW_ADC_GET_READING

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

Defined in: ZW_adcdriv_api.h

Return value:

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

Serial API (Not supported)

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 ZW0201/ZW0301 ASIC. The API can be used both for slave and controller applications.

5.3.9.1 ZW_SetWutTimeout

void ZW_SetWutTimeout (BYTE wutTimeout)

Macro: ZW_SET_WUT_TIMEOUT(TIME)

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

Defined in: ZW_power_api.h

Parameters:

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

Serial API

HOST->ZW: REQ | 0xB4 | wutTimeout

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)

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)

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)

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)

5.3.10.9 UART_Enable

void UART_Enable(void)

Macro: ZW_UART_ENABLE

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

Defined in: ZW_uart_api.h

Serial API (Not supported)

5.3.10.10 UART_Disable

void UART_Disable(void)

Macro: ZW_UART_DISABLE

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

Defined in: ZW_uart_api.h

Serial API (Not supported)

5.3.10.11 UART_ClearTx

void UART_ClearTx(void)

Macro: ZW_UART_CLEAR_TX

Clear the UART transmit done flag.

Defined in: ZW_uart_api.h

Serial API (Not supported)

5.3.10.12 UART_ClearRx

void UART_ClearRx(void)

Macro: ZW_UART_CLEAR_RX

Clear the UART receiver ready flag.

Defined in: ZW_uart_api.h

Serial API (Not supported)

5.3.10.13 UART_Write

void UART_Write(BYTE txByte)

Macro: ZW_UART_WRITE(TXBYTE)

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

Defined in: ZW_uart_api.h

Parameters:

txByte IN Data to send.

Serial API (Not supported)

5.3.10.14 UART_Read

BYTE UART_Read(void)

Macro: ZW_UART_READ

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

Defined in: ZW_uart_api.h

Return value:

BYTE The contents of the UART receive register.

Serial API (Not supported)

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)

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 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.

5.3.11.1 ZW_NodeMaskSetBit

void ZW_NodeMaskSetBit(BYTE_P pMask, BYTE bNodeID)

Macro: ZW_NODE_MASK_SET_BIT(pMask, bNodeID)

Set the node bit in a node bit mask.

Defined in: ZW_nodemask_api.h

Parameters:

pMask IN Pointer to node mask

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

Serial API (Not supported)

5.3.11.2 ZW_NodeMaskClearBit

void ZW_NodeMaskClearBit(BYTE_P pMask, BYTE bNodeID)

Macro: ZW_NODE_MASK_CLEAR_BIT(pMask, bNodeID)

Clear the node bit in a node bit mask.

Defined in: ZW_nodemask_api.h

Parameters:

PMask IN Pointer to node mask

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

Serial API (Not supported)

5.3.11.3 ZW_NodeMaskClear

void ZW_NodeMaskClear(BYTE_P pMask, BYTE bLength)

Macro: ZW_NODE_MASK_CLEAR(pMask, bLength)

Clear all bits in a node mask.

Defined in: ZW_nodemask_api.h

Parameters:

pMask IN Pointer to node mask

bLength IN Length of node mask

Serial API (Not supported)

5.3.11.4 ZW_NodeMaskBitsIn

BYTE ZW_NodeMaskBitsIn(BYTE_P pMask, BYTE bLength)

Macro: ZW_NODE_MASK_BITS_IN (pMask, bLength)

Number of bits set in node mask.

Defined in: ZW_nodemask_api.h

Return value:

BYTE Number of bits set in node mask

Parameters:

pMask IN Pointer to node mask

bLength IN Length of node mask

Serial API (Not supported)

5.3.11.5 ZW_NodeMaskNodeIn

BYTE ZW_NodeMaskNodeIn (BYTE_P pMask, BYTE bNode)

Macro: ZW_NODE_MASK_NODE_IN (pMask, bNode)

Check if a node is in a node mask.

Defined in: ZW_nodemask_api.h

Return value:

| | | |
|------|----------|---------------------|
| BYTE | ZERO | If not in node mask |
| | NONEZERO | If in node mask |

Parameters:

| | | |
|-------|----|----------------------------|
| pMask | IN | Pointer to node mask |
| bNode | IN | Node to clear in node mask |

Serial API (Not supported)

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

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

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

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

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

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

Defined in: ZW_controller_api.h

Parameters:

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

Callback function Parameters (completedFunc):

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

Serial API:

HOST->ZW: REQ | 0x4A | mode | funcID

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

5.4.2 ZW_AreNodesNeighbours

BYTE ZW_AreNodesNeighbours (BYTE bNodeA, BYTE bNodeB)

Macro: ZW_ARE_NODES_NEIGHBOURS (nodeA, nodeB)

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

Defined in: ZW_controller_api.h

Return value:

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

Parameters:

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

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

Serial API

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

ZW->HOST: RES | 0xBC | retVal

5.4.3 ZW_AssignReturnRoute

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

Macro: ZW_ASSIGN_RETURN_ROUTE(routingNodeID, destNodeID, func)

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

Defined in: ZW_controller_api.h

Return value:

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

Parameters:

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

Callback function Parameters:

| | |
|-------------|---------------------------|
| txStatus IN | (see ZW_SendData) |
|-------------|---------------------------|

Serial API:

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

ZW->HOST: RES | 0x46 | retVal

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

5.4.4 ZW_AssignSUCReturnRoute

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

Macro: ZW_ASSIGN_SUC_RETURN_ROUTE(srcnode,func)

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

Defined in: ZW_controller_api.h

Return value:

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

Parameters:

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

completedFunc IN Transmit complete call back.

Callback function Parameters:

bStatus IN (see ZW_SendData)

Serial API:

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

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

ZW->HOST: RES | 0x51 | retVal

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

5.4.5 ZW_ControllerChange

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

Macro: ZW_CONTROLLER_CHANGE(mode, func)

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

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

Defined in: ZW_controller_api.h

Parameters:

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

Callback function Parameters (completedFunc):

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

Serial API:

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

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

5.4.6 ZW_DeleteReturnRoute

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

Macro: ZW_DELETE_RETURN_ROUTE(nodeID, func)

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

Defined in: ZW_controller_api.h

Return value:

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

Parameters:

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

completedFunc IN Transmit completed call back function

Callback function Parameters:

txStatus IN (see **ZW_SendData**)

Serial API:

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

ZW->HOST: RES | 0x47 | retVal

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

5.4.7 ZW_DeleteSUCReturnRoute

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

Macro: ZW_DELETE_SUC_RETURN_ROUTE (nodeID, func)

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

Defined in: ZW_controller_api.h

Return value:

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

Parameters:

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

completedFunc IN Transmit complete call back.

Callback function Parameters:

txStatus IN (see **ZW_SendData**)

Serial API:

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

ZW->HOST: RES | 0x55 | retVal

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

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

- If request nodeinfo transmission was unsuccessful (no ACK received) then the **ApplicationControllerUpdate** is called with UPDATE_STATE_NODE_INFO_REQ_FAILED (status only available in the Serial API implementation).
- If request nodeinfo transmission was successful there is no indication that it went well apart from the returned Nodeinfo frame which should be received via the **ApplicationControllerUpdate** with status UPDATE_STATE_NODE_INFO_RECEIVED.

5.4.8 ZW_GetControllerCapabilities

BYTE ZW_GetControllerCapabilities (void)

Macro: ZW_GET_CONTROLLER_CAPABILITIES()

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

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

Defined in: ZW_controller_api.h

Return value:

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

Serial API:

HOST->ZW: REQ | 0x05

ZW->HOST: RES | 0x05 | RetVal

5.4.9 ZW_GetNeighborCount

BYTE ZW_GetNeighborCount(BYTE nodeID)

Macro: ZW_GET_NEIGHBOR_COUNT (nodeID)

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

Defined in: ZW_controller_api.h

Return value:

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

Parameters:

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

Serial API

HOST->ZW: REQ | 0xBB | nodeID

ZW->HOST: RES | 0xBB | retVal

5.4.10 ZW_GetNodeProtocolInfo

```
void ZW_GetNodeProtocolInfo(BYTE bNodeID,  
                             NODEINFO, *nodeInfo)
```

Macro: ZW_GET_NODE_STATE(nodeID, nodeInfo)

Return the Node Information Frame without command classes from the non-volatile memory for a given node ID:

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

Figure 10. Node Information frame structure without command classes

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

Defined in: ZW_controller_api.h

Parameters:

| | | |
|--------------|----------------------------------|---|
| bNodeID IN | Node ID | 1..232 |
| nodeInfo OUT | Node info buffer (see Figure 10) | 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 10)

5.4.11 ZW_GetRoutingInfo

```
void ZW_GetRoutingInfo(BYTE bNodeID,
                      BYTE_P pMask,
                      BYTE bRemove)
```

Macro: ZW_GET_ROUTING_INFO(bNodeID, pMask, bRemove)

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

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

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

| pMask[i] ($0 \leq i < (\text{ZW_MAX_NODES}/8)$) | | | | | | | | |
|---|---------|---------|---------|---------|---------|---------|---------|---------|
| Bit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| NodeID | $i*8+1$ | $i*8+2$ | $i*8+3$ | $i*8+4$ | $i*8+5$ | $i*8+6$ | $i*8+7$ | $i*8+8$ |

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

Defined in: ZW_controller_api.h

Parameters:

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

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

bRemove IN GET_ROUTING_INFO_REMOVE_BAD Remove bad routes from the routing info.

GET_ROUTING_INFO_REMOVE_NON_REPS Remove non-repeaters from the routing info.

GET_ROUTING_INFO_REMOVE_9600 Remove 9.6K nodes from the routing info.

Serial API:

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

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

5.4.12 ZW_GetRoutingMAX

BYTE ZW_GetRoutingMAX(void)

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

Defined in: ZW_controller_api.h

Return value:

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

Serial API:

Not implemented

5.4.13 ZW_GetSUCNodeID

BYTE ZW_GetSUCNodeID(void)

Macro: ZW_GET_SUC_NODE_ID()

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

Defined in: ZW_controller_api.h

Return value:

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

Serial API:

HOST->ZW: REQ | 0x56

ZW->HOST: RES | 0x56 | SUCNodeID

5.4.14 ZW_isFailedNode

BYTE ZW_isFailedNode(BYTE nodeID)

Macro: ZW_IS_FAILED_NODE_ID(nodeID)

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

Defined in: ZW_controller_api.h

Return value:

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

Parameters:

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

Serial API:

HOST->ZW: REQ | 0x62 | nodeID

ZW->HOST: RES | 0x62 | retVal

5.4.15 ZW_IsPrimaryCtrl

BOOL ZW_IsPrimaryCtrl (void)

Macro: ZW_PRIMARYCTRL()

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

Defined in: ZW_controller_api.h

Return value:

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

Serial API (Not supported)

5.4.16 ZW_RemoveFailedNodeID

**BYTE ZW_RemoveFailedNodeID(BYTE NodeID,
 BOOL bNormalPower,
 VOID_CALLBACKFUNC(completedFunc)(BYTE txStatus))**

Macro: ZW_REMOVE_FAILED_NODE_ID(node,func)

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

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

Defined in: ZW_controller_api.h

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

| | | |
|------|-------------------------------|------------------------------|
| BYTE | ZW_FAILED_NODE_REMOVE_STARTED | The removing process started |
|------|-------------------------------|------------------------------|

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

| | | |
|------|------------------------------------|---|
| BYTE | ZW_NOT_PRIMARY_CONTROLLER | The removing process was aborted because the controller is not the primary one. |
| | ZW_NO_CALLBACK_FUNCTION | The removing process was aborted because no call back function is used. |
| | ZW_FAILED_NODE_NOT_FOUND | The requested process failed. The nodeID was not found in the controller list of failing nodes. |
| | ZW_FAILED_NODE_REMOVE_PROCESS_BUSY | The removing process is busy. |
| | ZW_FAILED_NODE_REMOVE_FAIL | The requested process failed. Reasons include: <ul style="list-style-type: none"> • Controller is busy • The node responded to a NOP; thus the node is no longer failing. |

Parameters:

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

Callback function Parameters:

txStatus IN Status of removal of failed node:

ZW_NODE_OK

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

ZW_FAILED_NODE_REMOVED

The failed node was removed from the failed nodes list.

ZW_FAILED_NODE_NOT_REMOVED

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

Serial API:

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

ZW->HOST: RES | 0x61 | retVal

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

5.4.17 ZW_ReplaceFailedNode

```

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

```

Macro: ZW_REPLACE_FAILED_NODE(node,func)

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

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

Defined in: ZW_controller_api.h

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

| | |
|--|------------------------------------|
| BYTE ZW_FAILED_NODE_REMOVE_STARTED | The replacing process has started. |
|--|------------------------------------|

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

| | |
|--|--|
| BYTE ZW_NOT_PRIMARY_CONTROLLER | The replacing process was aborted because the controller is not a primary/inclusion/SIS controller. |
| ZW_NO_CALLBACK_FUNCTION | The replacing process was aborted because no call back function is used. |
| ZW_FAILED_NODE_NOT_FOUND | The requested process failed. The nodeID was not found in the controller list of failing nodes. |
| ZW_FAILED_NODE_REMOVE_PROCESS_BUSY | The removing process is busy. |
| ZW_FAILED_NODE_REMOVE_FAIL | The requested process failed. Reasons include: <ul style="list-style-type: none"> • Controller is busy • The node responded to a NOP; thus the node is no longer failing. |

Parameters:

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

Callback function Parameters:

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

Serial API:

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

ZW->HOST: RES | 0x63 | retVal

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

5.4.18 ZW_RemoveNodeFromNetwork

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

Macro: ZW_REMOVE_NODE_FROM_NETWORK(mode, func)

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

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

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

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

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

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

Defined in: ZW_controller_api.h

Parameters:

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

Callback function Parameters (completedFunc):

| | | |
|------------------------------|--|---|
| *learnNodeInfo.bStatus IN | Status of learn mode: | |
| | REMOVE_NODE_STATUS_LEARN_READY | The controller is now ready to remove a node from the network. |
| | REMOVE_NODE_STATUS_NODE_FOUND | A node that wants to be deleted from the network has been found |
| | REMOVE_NODE_STATUS_REMOVING_* | A slave/controller node has been removed from the network. Remove node ID is returned. |
| | REMOVE_NODE_STATUS_DONE | The node has now been removed and the controller is ready to continue normal operation again. |
| | REMOVE_NODE_STATUS_FAILED | The remove process failed |
| *learnNodeInfo.bSource IN | Node id of the removed node | |
| *learnNodeInfo.pCmd IN | Pointer to Application Node information data (see ApplicationNodeInformation - nodeParm). NULL if no information present. | |
| | The pCmd only contain information when bLen is not zero, so the information should be stored when that is the case. Regardless of the bStatus. | |
| *learnNodeInfo.bLen IN | Node info length. | |

Serial API:

HOST->ZW: REQ | 0x4B | mode | funcID

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

5.4.19 ZW_ReplicationReceiveComplete

void ZW_ReplicationReceiveComplete(void)

Macro: ZW_REPLICATION_COMMAND_COMPLETE

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

Defined in: ZW_controller_api.h

Serial API:

HOST->ZW: REQ | 0x44

5.4.20 ZW_ReplicationSend

```

BYTE ZW_ReplicationSend(BYTE destNodeID, BYTE *pData, BYTE dataLength,  

                        BYTE txOptions,  

                        VOID_CALLBACKFUNC(completedFunc)(BYTE txStatus))

```

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

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

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

Defined in: ZW_controller_api.h

Return value:

BYTE FALSE If transmit queue overflow.

Parameters:

| | |
|------------------|--|
| destNode IN | Destination Node ID (not equal NODE_BROADCAST). |
| pData IN | Data buffer pointer |
| dataLength IN | Data buffer length |
| txOptions IN | Transmit option flags. (see ZW_SendData , but avoid using routing!) |
| completedFunc IN | Transmit completed call back function |

Callback function Parameters:

txStatus IN (see **ZW_SendData**)

Serial API:

HOST->ZW: REQ | 0x45 | destNodeID | dataLength | pData[] | txOptions | funcID

ZW->HOST: RES | 0x45 | RetVal

ZW->HOST: REQ | 0x45 | funcID | txStatus

5.4.21 ZW_RequestNodeInfo

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

Macro: ZW_REQUEST_NODE_INFO(NODEID)

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

Defined in: ZW_controller_api.h

Return value:

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

Parameters:

nodeID IN The node ID (1...232) of the node to request the Node Information Frame from.

completedFunc IN Transmit complete call back.

Callback function Parameters:

txStatus IN (see **ZW_SendData**)

Serial API:

HOST->ZW: REQ | 0x60 | NodeID

ZW->HOST: RES | 0x60 | retVal

5.4.22 ZW_RequestNodeNeighborUpdate

**BYTE ZW_RequestNodeNeighborUpdate(NODEID,
VOID_CALLBACKFUNC (completedFunc)(BYTE
bStatus))**

Macro: ZW_REQUEST_NODE_NEIGHBOR_UPDATE(nodeid, func)

Get the neighbors from the specified node. This call can only be called by a primary/inclusion controller. An inclusion controller should call **ZW_RequestNetWorkUpdate** in advance because the inclusion controller may not have the latest network topology.

Defined in: ZW_controller_api.h

Return value:

| | | |
|------|-------|--|
| BYTE | TRUE | The discovery process is started and the function will be completed by the callback |
| | FALSE | The discovery was not started and the callback will not be called. The reason for the failure can be one of the following: <ul style="list-style-type: none"> • This is not a primary/inclusion controller • There is only one node in the network, nothing to update. • The controller is busy doing another update. |

Parameters:

nodeID IN Node ID (1...232) of the node that the controller wants to get new neighbors from.

completedFunc IN Transmit complete call back.

Callback function Parameters:

| | |
|------------|--|
| bStatus IN | Status of command: |
| | REQUEST_NEIGHBOR_UPDATE_STARTED Requesting neighbor list from the node is in progress. |
| | REQUEST_NEIGHBOR_UPDATE_DONE New neighbor list received |
| | REQUEST_NEIGHBOR_UPDATE_FAIL Getting new neighbor list failed |

Serial API:

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

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

5.4.23 ZW_SendSUCID

**BYTE ZW_SendSUCID (BYTE node,
BYTE txOption,
VOID_CALLBACKFUNC (completedFunc)(BYTE txStatus))**

Macro: ZW_SEND_SUC_ID(nodeID, txOption, func)

Transmit SUC node ID from a primary controller or static controller to the controller node ID specified. Routing slaves ignore this command, use instead ZW_AssignSUCReturnRoute.

Defined in: ZW_controller_api.h

Return value:

| | |
|-------|--|
| TRUE | In progress. |
| FALSE | Not a primary controller or static controller. |

Parameters:

| | |
|------------------|---|
| node IN | The node ID (1...232) of the node to receive the current SUC node ID. |
| txOption IN | Transmit option flags. (see ZW_SendData) |
| completedFunc IN | Transmit complete call back. |

Callback function parameters:

| | |
|-------------|---------------------------|
| txStatus IN | (see ZW_SendData) |
|-------------|---------------------------|

Serial API:

HOST->ZW: REQ | 0x57 | node | txOption | funcID

ZW->HOST: RES | 0x57 | RetVal

ZW->HOST: REQ | 0x57 | funcID | txStatus

5.4.24 ZW_SetDefault

void ZW_SetDefault(VOID_CALLBACKFUNC(completedFunc)(void))

Macro: ZW_SET_DEFAULT(func)

This function set the Controller back to the factory default state. Erase all Nodes, routing information and assigned homeID/nodeID from the EEPROM memory. Finally write a new random home ID to the EEPROM memory.

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

Warning: Use this function with care as it could render a Z-Wave network unusable if the primary controller in an existing network is set back to default.

Defined in: ZW_controller_api.h

Parameters:

completedFunc IN Command completed call back function

Serial API:

HOST->ZW: REQ | 0x42 | funcID

ZW->HOST: REQ | 0x42 | funcID

5.4.25 ZW_SetLearnMode

```
void ZW_SetLearnMode (BYTE mode,
                     VOID_CALLBACKFUNC(completedFunc)(LEARN_INFO *learnNodeInfo))
```

Macro: ZW_SET_LEARN_MODE(mode, func)

ZW_SetLearnMode is used to add or remove the controller to a Z-Wave network.

This function is used to instruct the controller to allow it to be added or removed from the network.

When a controller is added to the network the following things will happen:

1. The controller is assigned a valid Home ID and Node ID
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: | |
| | ZW_SET_LEARN_MODE_CLASSIC | Start the learn mode on the controller and only accept being included in direct range. |
| | ZW_SET_LEARN_MODE_NWI | Start the learn mode on the controller and accept routed inclusion. |
| | ZW_SET_LEARN_MODE_DISABLE | Stop learn mode on the controller |
| completedFunc IN | Callback function pointer (Should only be NULL if state is turned off). | |

Callback function Parameters (completedFunc):

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

Serial API:

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

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

5.4.26 ZW_SetRoutingInfo

```
void ZW_SetRoutingInfo(BYTE bNodeID,
                      BYTE bLength,
                      BYTE_P pMask )
```

Macro: ZW_SET_ROUTING_INFO(bNodeID, bLength, pMask)

NOTE: This function is not available in the Bridge Controller library and Static Controller library without repeater and manual routing functionality.

ZW_SetRoutingInfo is a function that can be used to overwrite the current neighbor information for a given node ID in the protocol locally.

The format of the routing info must be organised as follows:

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

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

Defined in: ZW_controller_api.h

Return value:

| | | |
|------|-------|--|
| BOOL | TRUE | Neighbor information updated successfully. |
| | FALSE | Failed to update neighbor information. |

Parameters:

| | |
|------------|---|
| bNodeID IN | Node ID (1...232) to be updated with respect to neighbor information. |
| bLength IN | Routing info buffer length in bytes. |
| pMask IN | Pointer to buffer where routing info should be taken from. The buffer should be at least ZW_MAX_NODES/8 bytes |

Serial API (Only Developer's Kit v4.5x):

HOST->ZW: REQ | 0x1B | bNodeID | NodeMask[29]

ZW->HOST: RES | 0x1B | retVal

5.4.27 ZW_SetRoutingMAX

BOOL ZW_SetRoutingMAX(BYTE maxRouteTries)

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

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

Defined in: ZW_controller_api.h

Parameters:

maxRouteTries IN 1...20 Maximum number of source routing attempts

Serial API:

Not implemented

5.4.28 ZW_SetSUCNodeID

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

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

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

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

Defined in: ZW_controller_api.h

Return value:

TRUE

If the process of configuring the static/bridge controller is started.

FALSE

The process not started because the calling controller is not the master or the

destination node is not a static/bridge controller.

Parameters:

| | | |
|------------------|--|---|
| nodeID IN | The node ID (1...232) of the static controller to configure. | |
| SUCState IN | TRUE | Want the static controller to be a SUC node. |
| | FALSE | If the static/bridge controller should not be a SUC node. |
| bTxOption IN | TRUE | Want to send the frame with low transmission power |
| | FALSE | Want to send the frame at normal transmission power |
| capabilities IN | SUC capabilities that is enabled: | |
| | ZW_SUC_FUNC_BASIC_SUC | Only enables the basic SUC functionality. |
| | ZW_SUC_FUNC_NODEID_SERVER | Enable the node ID server functionality to become a SIS. |
| completedFunc IN | Transmit complete call back. | |

Callback function Parameters:

| | | |
|-------------|----------------------|---------------------------------|
| txStatus IN | Status of command: | |
| | ZW_SUC_SET_SUCCEEDED | The process ended successfully. |
| | ZW_SUC_SET_FAILED | The process failed. |

Serial API:

HOST->ZW: REQ | 0x54 | nodeID | SUCState | bTxOption | capabilities | funcID

ZW->HOST: RES | 0x54 | RetVal

ZW->HOST: REQ | 0x54 | funcID | txStatus

In case **ZW_SetSUCNodeID** is called locally with the controllers own node ID then only the response is returned. In case true is returned in the response then it can be interpreted as the command is now executed successfully.

5.5 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_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 new primary controller to the network. |
| | CREATE_PRIMARY_STOP | Stop the process. |
| | CREATE_PRIMARY_STOP_FAILED | Stop the inclusion and report a failure to the other controller. |
| completedFunc IN | Callback function pointer (Should only be NULL if state is turned off). | |

Callback function Parameters:

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

Serial API:

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

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

5.5.2 ZW_EnableSUC

BYTE ZW_EnableSUC (BYTE state, BYTE capabilities)

Macro: ZW_ENABLE_SUC (state, capabilities)

Used to enable/disable assignment of the SUC/SIS functionality in the controller. Assignment is default enabled. Assignment is done by the API call **ZW_SetSUCNodeID**.

If SUC is enabled then the static controller can store network changes sent from the primary, send network topology updates requested by controllers.

If SUC is disabled, then the static controller will ignore the frames sent from the primary controller after calling **ZW_SetSUCNodeID**. If the primary controller called **ZW_RequestNetWorkUpdate**, then the call back function will return with ZW_SUC_UPDATE_DISABLED.

Defined in: ZW_controller_static_api.h

Return value:

| | | |
|------|-------|---|
| BYTE | TRUE | The SUC functionality was enabled/disabled. |
| | FALSE | Attempting to disable a running SUC, not allowed. |

Parameters:

| | | |
|-----------------|-----------------------------------|--|
| State IN | TRUE | SUC functionality is enabled. |
| | FALSE | SUC functionality is disabled. |
| capabilities IN | SUC capabilities that is enabled: | |
| | ZW_SUC_FUNC_BASIC_SUC | Only enables the basic SUC functionality. |
| | ZW_SUC_FUNC_NODEID_SERVER | Enable the SUC node ID server functionality to become a SIS. |

Serial API:

HOST->ZW: REQ | 0x52 | state | capabilities

ZW->HOST: RES | 0x52 | retVal

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_GetVirtualNodes

VOID ZW_GetVirtualNodes(BYTE *pnodeMask)

Macro: ZW_GET_VIRTUAL_NODES (pnodemask)

Request a buffer containing available Virtual Slave nodes in the Z-Wave network.

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

| pnodeMask[i] ($0 \leq i < (ZW_MAX_NODES/8)$) | | | | | | | | |
|--|---------|---------|---------|---------|---------|---------|---------|---------|
| Bit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| NodeID | $i*8+1$ | $i*8+2$ | $i*8+3$ | $i*8+4$ | $i*8+5$ | $i*8+6$ | $i*8+7$ | $i*8+8$ |

If bit n in pnodeMask[i] is 1, it indicates that node $(i*8)+n+1$ is a Virtual Slave node. If bit n in pnodeMask[i] is 0, it indicates that node $(i*8)+n+1$ is not a Virtual Slave node.

Defined in: ZW_controller_bridge_api.h

Parameters:

pNodeMask IN Pointer to nodemask (29 byte size)
buffer where the Virtual Slave
nodeMask should be copied.

Serial API:

HOST->ZW: REQ | 0xA5

ZW->HOST: RES | 0xA5 | pnodeMask[29]

5.6.2 ZW_IsVirtualNode

BYTE ZW_IsVirtualNode(BYTE nodeID)

Macro: ZW_IS_VIRTUAL_NODE (nodeid)

Checks if “nodeID” is a Virtual Slave node.

Defined in: ZW_controller_bridge_api.h

Return value:

| | | |
|------|-------|--|
| BYTE | TRUE | If “nodeID” is a Virtual Slave node. |
| | FALSE | If “nodeID” is not a Virtual Slave node. |

Parameters:

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

Serial API:

HOST->ZW: REQ | 0xA6 | nodeID

ZW->HOST: RES | 0xA6 | retVal

5.6.3 ZW_SendSlaveNodeInformation

```

BYTE ZW_SendSlaveNodeInformation(BYTE srcNode,
                                BYTE destNode,
                                BYTE txOptions,
                                VOID_CALLBACKFUNC(completedFunc)(BYTE txStatus))

```

Macro: ZW_SEND_SLAVE_NODE_INFO(srcnode, destnode, option, func)

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

NOTE: ZW_SendSlaveNodeInformation uses the transmit queue in the API, so using other transmit functions before the complete callback has been called by the API might fail.

Defined in: ZW_controller_bridge_api.h

Return value:

| | | |
|------|-------|---|
| BYTE | TRUE | If frame was put in the transmit queue. |
| | FALSE | If transmitter queue overflow or if bridge controller is primary or srcNode is invalid then completedFunc will NOT be called. |

Parameters:

| | | |
|------------------|--|--|
| srcNode IN | Source Virtual Slave Node ID | |
| destNode IN | Destination Node ID (NODE_BROADCAST == all nodes) | |
| txOptions | IN Transmit option flags: | |
| | TRANSMIT_OPTION_LOW_POWER | Transmit at low output power level (1/3 of normal RF range). NOTE: The TRANSMIT_OPTION_LOW_POWER option should only be used when the two nodes that are communicating are close to each other (<2 meter). In all other cases this option should not be used. |
| | TRANSMIT_OPTION_ACK | Request acknowledge from destination node. |
| completedFunc IN | Transmit completed call back function | |

Callback function Parameters:

txStatus (see ZW_SendData)

Serial API:

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

ZW->HOST: RES | 0xA2 | retVal

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

5.6.4 ZW_SetSlaveLearnMode

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

Macro: ZW_SET_SLAVE_LEARN_MODE (node, mode, func)

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

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

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

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

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

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

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

Defined in: ZW_controller_bridge_api.h

Return value:

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

Parameters:

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

Callback function Parameters:

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

Serial API:

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

ZW->HOST: RES | 0xA4 | retVal

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

5.7 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)

5.7.2 ZW_StoreHomeID

**void ZW_StoreHomeID(BYTE_P pHomeID,
 BYTE bNodeID)**

Macro: ZW_STORE_HOME_ID(pHomeID, NodeID)

ZW_StoreHomeID is a function that can be used to restore HomeID and NodeID information from a backup.

Defined in: ZW_controller_installer_api.h

Parameters:

pHomeID IN Pointer to HomeID structure to store

bNodeID IN NodeID to store.

Serial API:

HOST->ZW: REQ | 0x84 | pHomeID[0] | pHomeID[1] | pHomeID[2] | pHomeID[3] | bNodeID

5.7.3 ZW_StoreNodeInfo

```

BOOL ZW_StoreNodeInfo(BYTE bNodeID,  

                       BYTE_P pNodeInfo,  

                       VOID_CALLBACKFUNC(func)())

```

Macro: ZW_STORE_NODE_INFO(NodeID,NodeInfo,function)

ZW_StoreNodeInfo is a function that can be used to restore protocol node information from a backup or the like. The format of the node info frame should be identical with the format used by ZW_GET_NODE_STATE.

Defined in: ZW_controller_installer_api.h

Return value:

| | | |
|------|-------|--|
| BOOL | TRUE | If NodeInfo was Stored. |
| | FALSE | If NodeInfo was not Stored. (Illegal NodeId or MemoryWrite failed) |

Parameters:

| | |
|--------------|--|
| bNodeID IN | Node ID (1...232) to store information at. |
| pNodeInfo IN | Pointer to Node Information Frame. |
| func IN | Callback function. Called when data has been stored. |

Serial API:

HOST->ZW: REQ | 0x83 | bNodeID | nodeInfo (nodeInfo is a NODEINFO field) | funcID

ZW->HOST: RES | 0x83 | retVal

ZW->HOST: REQ| 0x83 | funcId

5.8 Z-Wave Slave API

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

5.8.1 ZW_SetDefault

void ZW_SetDefault(void)

Macros: ZW_SET_DEFAULT

This function set the slave back to the factory default state. Erase routing information and assigned homeID/nodeID from the EEPROM memory. Support of 9.6kbps communication only is also cleared and **ZW_Support9600Only** must be called to set it again. Finally write a new random home ID to the EEPROM memory.

Defined in: ZW_slave_api.h

Serial API:

HOST->ZW: REQ | 0x42 | funcID

ZW->HOST: REQ | 0x42 | funcID

5.8.2 ZW_SetLearnMode

```
void ZW_SetLearnMode(BYTE mode,
                     VOID_CALLBACKFUNC(learnFunc)(BYTE bStatus, BYTE nodeID) )
```

Macro: ZW_SET_LEARN_MODE(mode, func)

ZW_SetLearnMode enable or disable home and node ID's learn mode. Use this function to add a new Slave node to a Z-Wave network. Setting the ID's to zero will remove the Slave node from the Z-Wave network, so that it can be moved to another network.

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

When learn mode is enabled, received "Assign ID's Command" are handled as follow:

1. If the current stored ID's are zero, the received ID's will be stored.
2. If the received ID's are zero the stored ID's will be set to zero.

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

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

Defined in: ZW_slave_api.h

Parameters:

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

Callback function Parameters:

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

Serial API:

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

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

5.8.3 ZW_Support9600Only

BOOL ZW_Support9600Only(BOOL bValue)

Macros: ZW_SUPPORT9600_ONLY(value)

The API call **ZW_Support9600Only** can select that non-listening ZW0201/ZW0301 slaves only want to support 9.6kbps communication. Support 9.6kbps only is cleared on RESET (this also includes WUT) and on calling **ZW_SetDefault**. Excluding the Node from the network will not clear support 9.6kbps only. The protocol will before sending still check for any ongoing 9.6 and 40kbps communication.

Important: Place this call only in ApplicationInitSW

Defined in: ZW_slave_api.h

Return value:

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

Parameters:

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

Serial API:

HOST->ZW: REQ | 0x5B | bValue

ZW->HOST: RES | 0x5B | retVal

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_GetSUCNodeID

BYTE ZW_GetSUCNodeID(void)

Macro: ZW_GET_SUC_NODE_ID()

API call used to get the currently registered SUC node ID. A controller must have called **ZW_AssignSUCReturnRoute** before a SUC node ID is registered in the routing or enhanced slave.

Defined in: ZW_slave_routing_api.h

Return value:

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

Serial API:

HOST->ZW: REQ | 0x56

ZW->HOST: RES | 0x56 | SUCNodeID

5.9.2 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:

HOST->ZW: REQ | 0x5D | bNodeID

ZW->HOST: RES | 0x5D | retVal

5.9.3 ZW_RediscoveryNeeded

**BYTE ZW_RediscoveryNeeded (BYTE bNodeID,
VOID_CALLBACKFUNC (completedFunc)(BYTE bStatus))**

Macro: ZW_REDISCOVERY_NEEDED(nodeid, func)

This function can request a SUC/SIS controller to update the requesting nodes neighbors. The function will try to request a neighbor rediscovery from a SUC/SIS controller in the network. In order to reach a SUC/SIS controller it uses other nodes (bNodeID) in the network. The application must implement the algorithm for scanning the bNodeID's to find a node which can help.

If bNodeID supports this functionality (routing slave and enhanced slave libraries), bNodeID will try to contact a SUC/SIS controller on behalf of the node that requests the rediscovery. If the functionality is unsupported by bNodeID ZW_ROUTE_LOST_FAILED will be returned in the callback function and the next node can be tried.

The callback function is called when the request have been processed by the protocol.

Defined in: ZW_slave_routing_api.h

Return value:

| | |
|-------|--|
| FALSE | The node is busy doing another update. |
| TRUE | The help process is started; status will come in the callback. |

Parameters:

bNodeID IN Node ID (1..232) to request help from

completedFunc IN Transmit completed call back function

Callback function parameters:

| | |
|-----------------------|---|
| ZW_ROUTE_LOST_ACCEPT | The node bNodeID accepts to forward the help request. Wait for the next callback to determine the outcome of the rediscovery. |
| ZW_ROUTE_LOST_FAILED | The node bNodeID has responded it is unable to help and the application can try next node if it decides so. |
| ZW_ROUTE_UPDATE_ABORT | No reply was received before the protocol has timed out. The application can try the next node if it decides so. |
| ZW_ROUTE_UPDATE_DONE | The node bNodeID was able to contact a controller and the routing information has been updated. |

Serial API:

HOST->ZW: REQ | 0x59 | bNodeID | funcID

ZW->HOST: RES | 0x59 | retVal

ZW->HOST: REQ | 0x59 | funcID | bStatus

5.9.4 ZW_RequestNewRouteDestinations

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

Macro: ZW_REQUEST_NEW_ROUTE_DESTINATIONS (pdestList, destListLen, func)

Used to request new return route destinations from the SUC/SIS node.

NOTE: No more than the first ZW_MAX_RETURN_ROUTE_DESTINATIONS will be requested regardless of bDestListLen.

Defined in: ZW_slave_routing_api.h

Return value:

| | |
|-------|--|
| TRUE | If the updating process is started. |
| FALSE | If the requesting routing slave is busy or no SUC node known to the slave. |

Parameters:

| | |
|------------------|--|
| pDestList IN | Pointer to a list of new destinations for which return routes is needed. |
| bDestListLen | Number of destinations contained in pDestList. |
| completedFunc IN | Transmit completed call back function |

Callback function parameters:

| | |
|--------------------------|---|
| ZW_ROUTE_UPDATE_DONE | The update process is ended successfully |
| ZW_ROUTE_UPDATE_ABORT | The update process aborted because of error |
| ZW_ROUTE_UPDATE_WAIT | The SUC node is busy |
| ZW_ROUTE_UPDATE_DISABLED | The SUC functionality is disabled |

Serial API:

HOST->ZW: REQ | 0x5C | destList[5] | funcID

ZW->HOST: RES | 0x5C | retVal

ZW->HOST: REQ | 0x5C | funcID | bStatus

5.9.5 ZW_RequestNodeInfo

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

Macro: ZW_REQUEST_NODE_INFO(NODEID)

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

Defined in: ZW_slave_routing_api.h

Return value:

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

Parameters:

nodeID IN The node ID (1...232) of the node to request the Node Information Frame from.

completedFunc IN Transmit complete call back.

Callback function Parameters:

txStatus IN (see **ZW_SendData**)

Serial API:

HOST->ZW: REQ | 0x60 | NodeID

ZW->HOST: RES | 0x60 | retVal

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

- If request nodeinfo transmission was unsuccessful (no ACK received) then the **ApplicationSlaveUpdate** is called with UPDATE_STATE_NODE_INFO_REQ_FAILED (status only available in the Serial API implementation).
- If request nodeinfo transmission was successful there is no indication that it went well apart from the returned Nodeinfo frame which should be received via the **ApplicationSlaveUpdate** with status UPDATE_STATE_NODE_INFO_RECEIVED.

5.10 Serial Command Line Debugger

The debug driver is a simple single line command interpreter, operated via the serial interface (UART – RS232). The command line debugger is used to dump and edit memory, including the memory mapped registers.

For a controller/slave_enhanced node the debugger startup by displaying the following help text on the debug terminal:

```
Z-Wave Commandline debugger Vx.nn
Keyes(VT100): BS; ^,<,> arrows; F1.
H                                Help
D[X|E|F] <addr> [<length>]      Dump memory
E[X|E]   <addr>                 Edit memory (Key: SP)
W[X|E|F] <addr>                 Watch memory location
                                is idata (80-FF is SFR)
  X      is xdata
    E    is External EEPROM
    F    is flash
>
```

For a slave node the debugger startup by displaying the following help text on the debug terminal:

```
Z-Wave Commandline debugger Vx.nn
Keyes(VT100): BS; ^,<,> arrows; F1.
H                                Help
D[X|I|F] <addr> [<length>]      Dump memory
E[X|I]   <addr>                 Edit memory (Key: SP)
W[X|I|F] <addr>                 Watch memory location
                                is idata (80-FF is SFR)
  X      is xdata
    I    is "Internal EEPROM" flash
    F    is flash
>
```

The command debugger is then ready to receive commands via the serial interface.

Special input keys:

- F1 (function key 1) same as the help command line.
- BS (backspace) delete the character left to the curser.
- < (left arrow) move the cursor one character left.
- > (right arrow) move the cursor one character right.
- ^ (up arrow) retrieve last command line.

Commands:

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

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

idata SRAM memory Rnn

xdata SRAM memory Xnn

Internal EEPROM flash Inn (slave only)

External EEPROM Enn (controllers/slave_enhanced only)

Examples:

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


5.10.1 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)

5.10.2 ZW_DebugPoll

void ZW_DebugPoll(void)

Macro: ZW_DEBUG_CMD_POLL

Command line debugger poll function. Collect characters from the debug terminal and execute the commands.

Should be called via the main poll loop (see function **ApplicationPoll**).

By using the debug macros (ZW_DEBUG_CMD_INIT, ZW_DEBUG_CMD_POLL) the command line debugger can be enabled by defining the compile flag "ZW_DEBUG_CMD" under CDEFINES in the makefile as follows:

```
CDEFINES+= EU,\
           ZW_DEBUG_CMD,\
           SUC_SUPPORT,\
           ASSOCIATION,\
           LOW_FOR_ON,\
           SIMPLELED
```

Both the debug output (ZW_DEBUG) and the command line debugger (ZW_DEBUG_CMD) can be enabled at the same time.

Defined in: ZW_debug_api.h

Serial API (Not supported)

5.11 RF Settings in App_RFSetup.a51 file

The Z-Wave libraries are capable of transmitting/receiving on either 921.42MHz (ANZ) or 868.42MHz (EU) or 919.82MHz (HK) or 865.22 MHz (IN) or 868.10MHz (MY) or 908.42MHz (US). The default frequency is set to US.

5.11.1 ZW0201/ZW0301 RF parameters

To allow for the selection of frequency, and transmit power levels (normal and low power) every application must have the App_RFSetup.a51 module linked in to define the const block placed in Flash memory. The ZW_RF020x.h / ZW_RF030x.h file contains various definitions such as default values:

Table 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 | 0x00-0x05 | 0x00 = EU 0x01 = US 0x02 = ANZ 0x03 = HK 0x04 = MY 0x05 = IN 0xFF = use default. |
| 2 | FLASH_APPL_NORM_POWER_OFFS | 0xFF | | If 0xFF the default lib value is used: US = 0x2A EU = 0x2A ANZ = 0x2A HK = 0x2A IN = 0x2A MY = 0x2A |
| 4 | FLASH_APPL_LOW_POWER_OFFS | 0xFF | | If 0xFF the default lib value is used: 0x14 |
| 5 | FLASH_APPL_PLL_STEPUP_OFFS | 0xFF | 0x00 | Only supported on ZW0301 |

An application programmer can select RF frequency (921.42MHz (ANZ) or 868.42MHz (EU) or 919.82MHz (HK) or IN - 865.22 MHz or 868.10MHz (MY) or 908.42MHz (US)), and the values for normal and low power transmissions by changing the const block defined in the App_RFSetup.a51 module. The RF frequency to use can be set by defining either ANZ or EU or HK or IN or MY or US in the application makefile. The TXnormal Power needs to be adjusted when making the FCC compliance test. According to the FCC part 15, the output radiated power shall not exceed 94dBuV/m. This radiated power is the result of the module output power and your product antenna gain. As the antenna gain is different from product to product, the module output power needs to be adjusted to comply with the FCC regulations.

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

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

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.

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 */
```

PIN_TOGGLE(pin)

Toggle output pin.

Parameters:

pin IN Z-Wave pin name

Example:

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

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 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.

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/IN/MY/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:

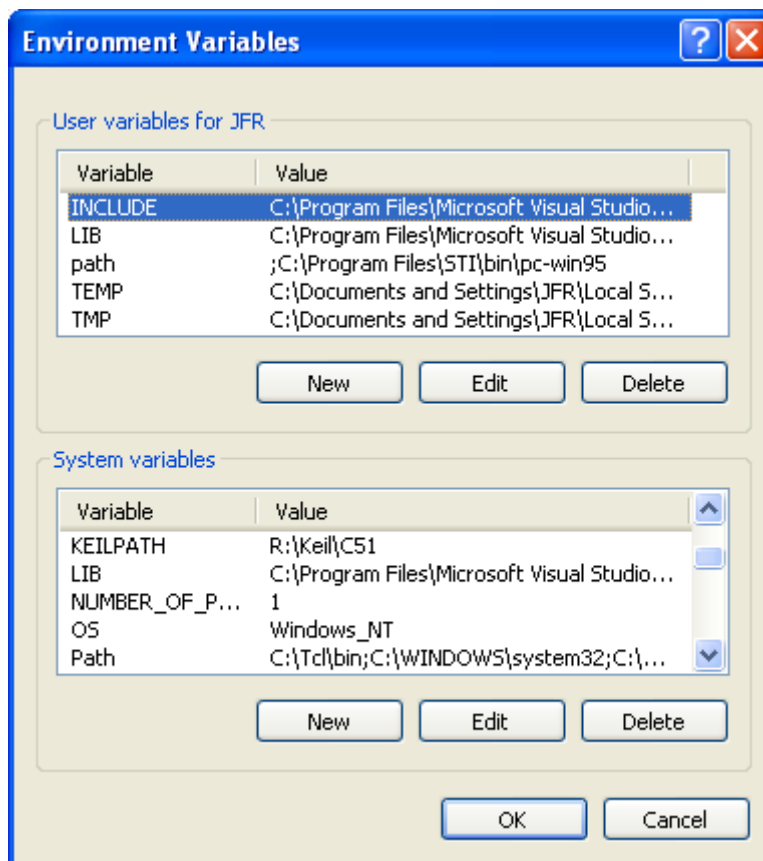


Figure 11. Configuring environment variables

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 **TOOLSDIR** (use capital letters because Windows XP is case sensitive)
8. 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.

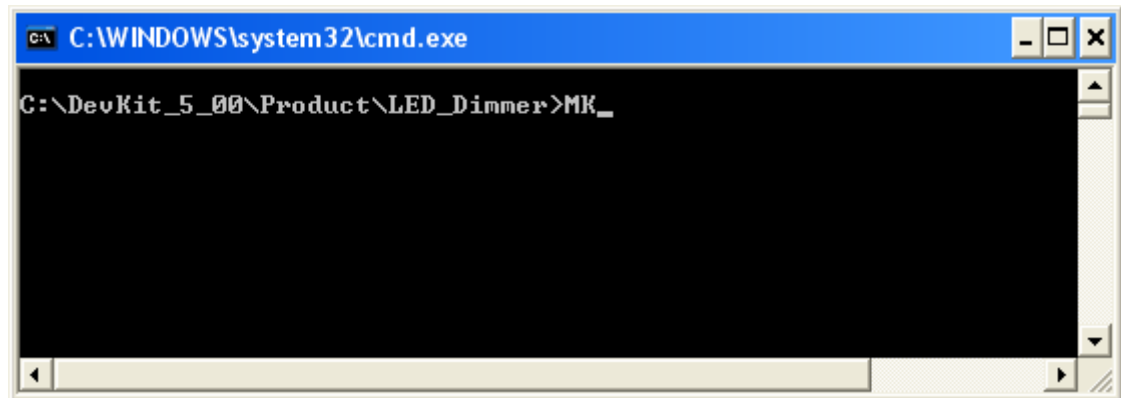


Figure 12. Building sample applications

Remember to use upper case in **KEILPATH**, **KEIL_LOCAL_PATH** and **TOOLS DIR** 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.

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.

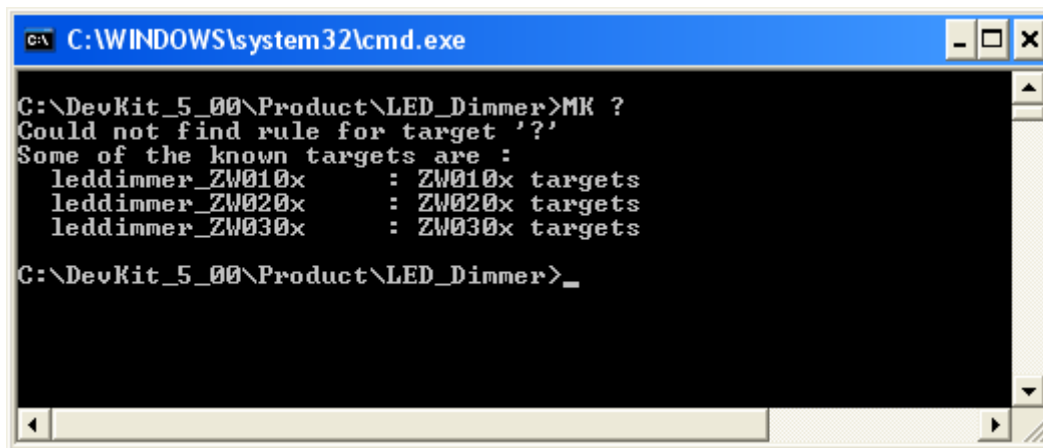


Figure 13. Possible sample application targets

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 ZW0201 targets:

- <application>
 - build
 - <application>_ZW020x_ANZ
 - list
 - Rels
 - <application>_ZW020x_ANZ.hex
 - <application>_ZW020x_EU
 - list
 - Rels
 - <application>_ZW020x_EU.hex
 - <application>_ZW020x_HK
 - list
 - Rels
 - <application>_ZW020x_HK.hex
 - <application>_ZW020x_IN
 - list
 - Rels
 - <application>_ZW020x_IN.hex
 - <application>_ZW020x_MY
 - list
 - Rels
 - <application>_ZW020x_MY.hex
 - <application>_ZW020x_US
 - list
 - Rels
 - <application>_ZW020x_US.hex
- contains list files
 - contains object files and map file
 - flash Intel hex file for ANZ ZW0201 based module
 - contains list files
 - contains object files and map file
 - flash Intel hex file for EU ZW0201 based module
 - contains list files
 - contains object files and map file
 - flash Intel hex file for HK ZW0201 based module
 - contains list files
 - contains object files and map file
 - flash Intel hex file for IN ZW0201 based module
 - contains list files
 - contains object files and map file
 - flash Intel hex file for MY ZW0201 based module
 - contains list files
 - contains object files and map file
 - flash Intel hex file for US ZW0201 based module

For ZW0301 targets:

- <application>
 - build
 - <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_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_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
 - <application>_ZW030x_IN
 - list - contains list files
 - Rels - contains object files and map file
 - <application>_ZW030x_IN.hex - flash Intel hex file for IN ZW0301 based module
 - <application>_ZW030x_MY
 - list - contains list files
 - Rels - contains object files and map file
 - <application>_ZW030x_MY.hex - flash Intel hex file for MY 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

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

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 |

7.2 Binary Sensor Sample Code

The Developer's Kit contains sample code for a non-secure and secure 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. Pressing the button a little while instead a node Info frame will be transmitted. A static controller such as the one described in [6] 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 none-secure Binary Sensor application lists the following supported command classes in the Node Information Frame:

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

When included non-secure the Secure Binary Sensor application lists the following supported command classes in the Node Information Frame:

Non-Secure Included

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

Secure Included

When included secure the Secure Binary Sensor application lists the following supported command classes in the Node Information Frame:

- Version command class
- Manufacturer Specific command class
- Security command class

The following listed in the Security Commands Supported Report frame:

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

The Basic command class is secure because application does not list it in Node Information Frame.

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 Network Wide Inclusion

By default the Bin_Sensor will enter network wide inclusion (NWI) when it is powered up and have not already been included Bin_Sensor. The Bin_Sensor will stay in NWI mode for 4 minutes or until it has been included into the network. Any key press will terminate the NWI mode and the only way to make the Bin_Sensor enter NWI mode again is by doing hardware reset either by remove and reapply the power or press the reset button on the side of the board.

The figure below show the NWI flow diagram implemented on application level for a slave. The slave_learn.c file contains the implementation situated in the util_func directory.

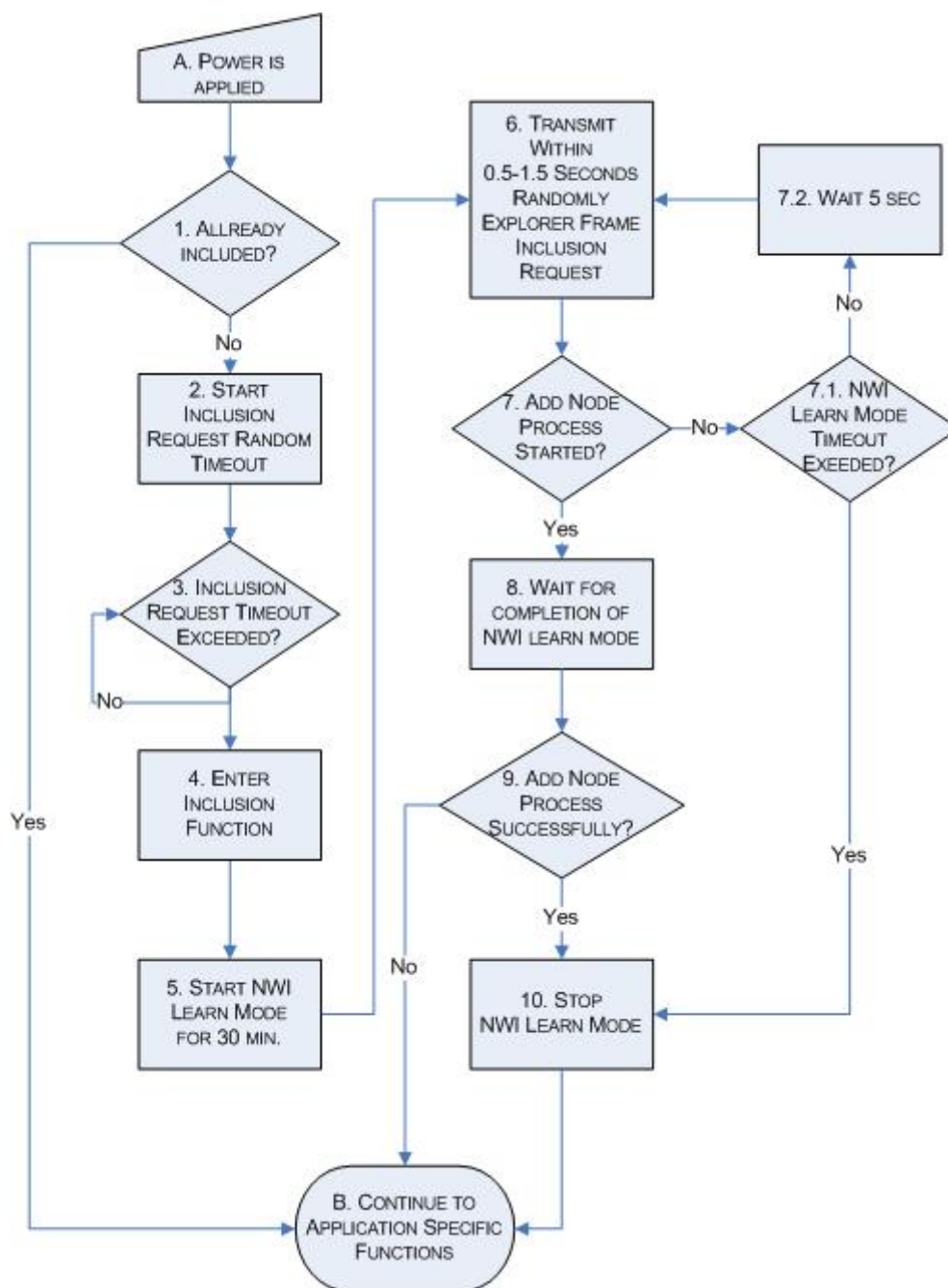


Figure 14. NWI flow diagram for a slave

7.2.2 Interface

The following table defines the functionality of the button on the Z-Wave module.

| | Button Triple Clicked | Button Clicked |
|----------------|---------------------------------------|------------------------------------|
| In Network | Node Info Frame / Enter learn mode | Basic Set (Broadcast) |
| Not in Network | Node Info Frame / Enter learn mode | None |
| Associated | Node Info Frame / Enter learn mode | Basic Set (to associated nodes) |

Learn mode is now activated by pressing the button three times within 1.5 seconds to avoid unintentional inclusion/exclusion of the node.

7.2.3 Bin_Sensor Files

The Product\Bin_Sensor directory contains sample source code for a non-secure/secure binary sensor and a non-secure/secure battery powered binary sensor slave application on a Z-Wave module. The application uses also a number of utility functions described in section 3.3.12.

MK.BAT

Batch file used to build ZW0201/ZW0301 based sample applications in ANZ, EU, HK, IN, MY and US versions respectively. Refer to chapter 7.1 regarding details about the build procedure.

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:

```
ANZ      : Build ANZ frequency target.
EU       : Build EU frequency target.
HK       : Build HK frequency target.
IN       : Build IN frequency target.
MY       : Build MY frequency target.
US       : Build US frequency target.
```

Makefile.binsensor_common

This makefile is a common file defining all dependencies etc.

eeeprom.h

This header file defines the addresses where application data are stored in the external EEPROM.

Bin_Sensor.h / Bin_Sensor.c

These files contain the source code for the binary sensor application state machine. The common API functions such as **ApplicationInitHW**, **ApplicationInitSW**, **ApplicationNodeInformation**, **ApplicationPoll**, **ApplicationSlaveUpdate** and **ApplicationCommandHandler** are defined here.

Battery_Sensor.mpw / Battery_Sensor_....Uv2

uVision3 multiproject workspace file (*.mpw) that unifies all the *.Uv2 project files. There is a project file for every particular target with respect to single chip series and frequency.

7.2.3.1 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 */
```


7.3 Binary Sensor Battery Sample Code

The Developer's Kit contains sample code for a non-secure and secure 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 WUT is fired. Upon wakeup, be it button press or WUT a Wakeup Notification Frame is sent either as broadcast or as singlecast to the device that configured the wakeup settings. If the device 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 described in [6] 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 none-secure battery-operated Binary Sensor lists the following supported command classes in the Node Information Frame:

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

When included non-secure the Secure Binary Sensor application lists the following supported command classes in the Node Information Frame:

Non-Secure Included

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

Secure Included

When included secure the secure battery-operated Binary Sensor application lists the following supported command classes in the Node Information Frame:

- Version command class
- Manufacturer Specific command class
- Security command class

The following listed in the Security Commands Supported Report frame:

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

The Basic command class is secure because application does not list it in Node Information Frame.

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 it will go in power down mode and will wakeup again either when the button is pressed or when the WUT is fired. While the Bin_sensor_Battery is awake 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 wake up caused by sensor input or button activity, then it will power down again when it is done executing any event, which have been produced, by either the sensor input or the button activity. If the binary sensor is awakened by WUT and the wakeup 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 wakeup notification frame will only be sent when the Bin_sensor_Battery has been assigned a node ID.

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 distinguished between by defining BATTERY when compiling which will also enable use of the utility function file battery.c/h.

7.3.1 Network Wide Inclusion

By default the Bin_Sensor_Battery will enter network wide inclusion (NWI) when it is powered up and have not already been included. The Bin_Sensor_Battery will stay in NWI mode for 4 minutes or until it has been included into the network. Any key press will terminate the NWI mode and the only way to make the Bin_Sensor_Battery enter NWI mode again is by doing hardware reset either by remove and reapply the power or press the reset button on the side of the board. Refer to section 7.2.1 regarding implementation details.

7.3.2 Interface

The following table defines the functionality of the button on the Z-Wave module.

| | Button Triple Pressed | Button Clicked |
|----------------------------|------------------------------------|---|
| In Network | Node Info Frame / Enter learn mode | Basic Set (Broadcast) |
| Not in Network | Node Info Frame / Enter learn mode | None |
| Associated | Node Info Frame / Enter learn mode | Basic Set (to associated nodes) |
| Wakeup Node set | Node Info Frame / Enter learn mode | Wake Up Notifications (to Wake up node) |
| Wakeup Node not set | Node Info Frame / Enter learn mode | Wake Up Notifications (Broadcast) |

7.3.3 Bin_Sensor_Battery Files

Refer to chapter 7.2.3.

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 using the Z-Wave portable controller API.

The Application complies with the Generic Controller command class [1]. When included the Development Controller application lists the following supported command classes in the Node Information Frame:

- Controller Replication command class
- Version command class

The Development Controller controls the following command classes:

- Controller Replication command class
- Basic command class
- Association command class

Controlled command classes not listed in the Node Information Frame in this sample application because it is optional to list.

For details regarding functionality supported by the development controller and user interface, refer to [15].

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 called when the Z-Wave basis software receives a frame. This could be a Basic Get Command to obtain the dim level of a multilevel switch.

7.4.1 Network Wide Inclusion

By default the controller will enter network wide inclusion (NWI) when it is powered up and have not already been included or have included other nodes itself. The controller will stay in NWI mode for 4 minutes or until it has been included into the network. Any key press will terminate the NWI mode and the only way to make the controller enter NWI mode again is by doing hardware reset either by remove and reapply the power or press the reset button on the side of the board.

The figure below show the NWI flow diagram implemented on application level for a controller. The ctrl_learn.c file contains the implementation situated in the util_func directory.

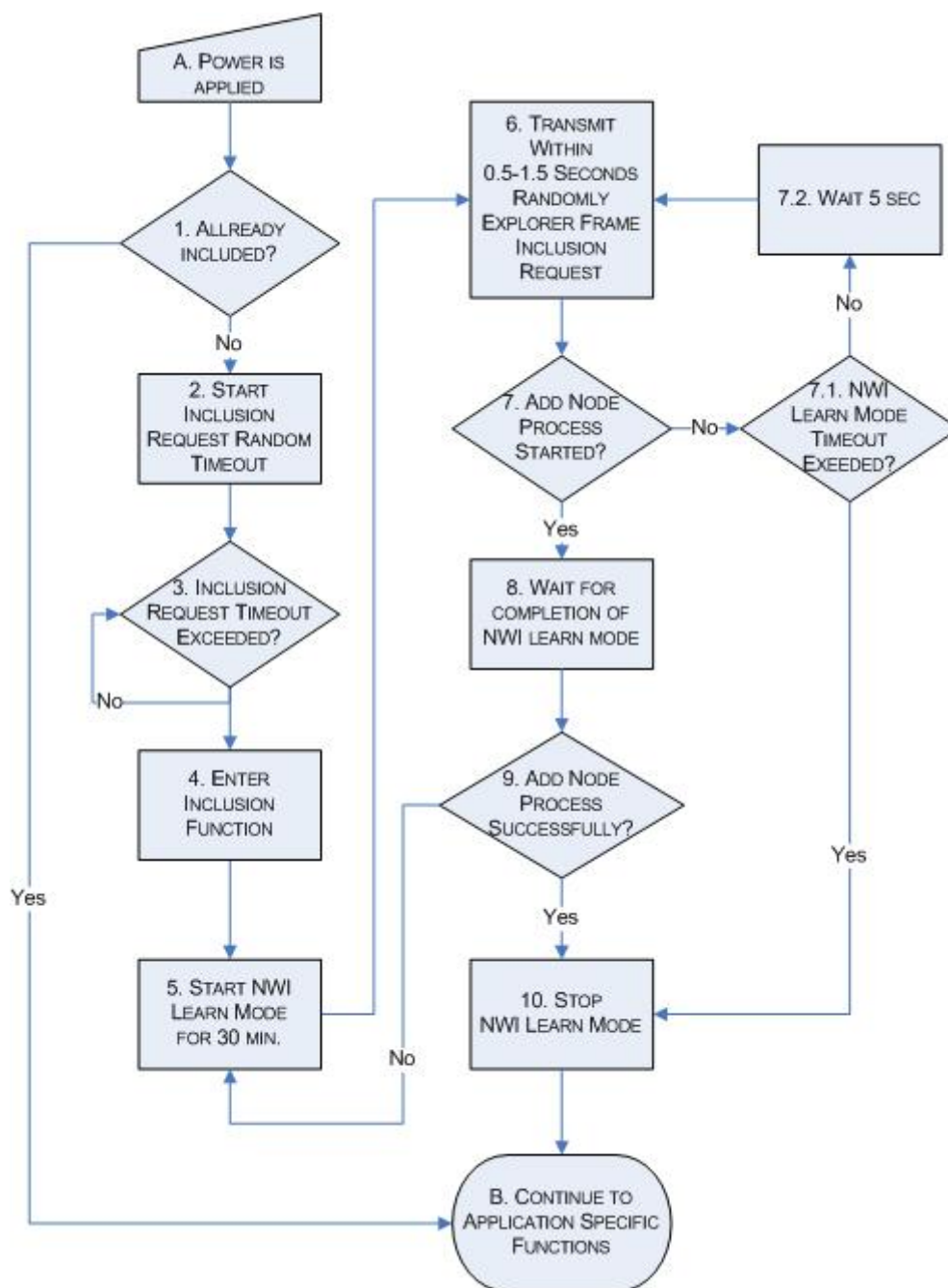


Figure 15. NWI flow diagram for a controller

7.4.2 Dev_Ctrl Files.

The Product\Dev_ctrl directory contains the source code for the controller application. The application uses also a number of utility functions described in section 3.3.12.

MK.BAT

Batch file used to build ZW0201/ZW0301 based sample applications in ANZ, EU, HK, IN, MY and US versions respectively. Refer to chapter 7.1 regarding details about the build procedure.

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:

| | |
|--------------|---|
| ANZ | : Build frequency target. |
| EU | : Build EU frequency target. |
| HK | : Build HK frequency target. |
| IN | : Build IN frequency target. |
| MY | : Build MY frequency target. |
| US | : Build US 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. |

Makefile.dev_ctrl_common

This makefile is a common file defining all dependencies etc.

eeeprom.c / eeeprom.h

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

dev_ctrl_if.h

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

dev_ctrl.c

This file contains the source code for the development controller application state machine. The common API functions such as **ApplicationInitHW**, **ApplicationInitSW**, **ApplicationNodeInformation**, **ApplicationPoll**, **ApplicationSlaveUpdate** and **ApplicationCommandHandler** are defined here.

p_button.c / p_button.h

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

dev_ctrl.mpw / dev_ctrl_....Uv2

uVision3 multiproject workspace file (*.mpw) that unifies all the *.Uv2 project files. There is a project file for every particular target with respect to single chip series and frequency.

7.5 Secure Development Controller (ATmega) Sample Code

The ZDK contains sample code that demonstrates how the basic tasks of adding, removing and controlling devices in a Z-Wave network can be accomplished using a host processor to control a Serial API based portable controller application. The application is a security updated Development Controller application. The Z-Wave Development Platform ZDP02A [31] or ZDP03A [32] is used for this purpose. The host processor is an AVR ATmega128 and software is build by environment below:

- IAR Embedded Workbench for Atmel AVR (v. 4.30A)
- IAR C/C++ Compiler for AVR 4.30A/W32 (4.30.1.5)

The AVR ISP In-System Programmer programs the AVR Atmega128.

When included non-secure the Development Controller application lists the following supported command classes in the Node Information Frame:

Non-Secure Included

- Controller Replication command class
- Version command class
- Security command class

Secure Included

When included secure the Development Controller lists the following supported command classes in the Node Information Frame:

- Version command class
- Security command class

The following listed in the Security Commands Supported Report frame:

- Controller Replication command class
- Version command class

The Basic command class is secure because application does not list it in Node Information Frame. The Development Controller controls the following command classes:

- Controller Replication command class
- Basic command class
- Association command class

Controlled command classes not listed in the Node Information Frame in this sample application because it is optional to list.

For further information about the features of the Secure Development Controller using an AVR as host, see [30].

7.5.1 Dev_Ctrl_AVR_Sec Files.

The Product\dev_ctrl_AVR_Sec directory contains the source code for the controller application. Only selected files in the directory structure is described below.

Portable.dep / .ewd / .ewp / .eww

Project files used to build AVR based sample application.

include\ZW_Security_AES_module.h

Header file used to implement security on application level.

include\AES_module.h

This header file contains definitions for implementing secure communication using AES as encrypting/decrypting mechanism.

src\ZW_Security_AES.c

These files contain shared data and functions for AES128 and functions for AES128 encryption/decryption. Files are not distributed on the Developer's Kit CD due to export restrictions. Contact support via zensys_support@sigmadesigns.com for further information.

Alternatively, implement the functions based on an Atmel's Application Note "AVR231: AES Bootloader":

http://www.atmel.com/dyn/resources/prod_documents/doc2589.pdf

7.6 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 doorbell system could be build. The Door Bell uses the frequently listening mode where it powers up the radio for a short period every 1000ms@ZW0201 / 250ms@ZW0301 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 doorbell.

7.6.1 Network Wide Inclusion

By default the Door Bell will enter network wide inclusion (NWI) when it is powered up and have not already been included. The Door Bell will stay in NWI mode for 4 minutes or until it has been included into the network. Any key press will terminate the NWI mode and the only way to make the Door Bell enter NWI mode again is by doing hardware reset either by remove and reapply the power or press the reset button on the side of the board. Refer to section 7.2.1 regarding implementation details.

7.6.2 User interface

The following list defines the functionality of the button on the Z-Wave module.

| | |
|-------------------------------|---|
| Press shortly | Wake up for 2 sec. |
| Press 3 times within 1.5 sec. | 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 awakened by button press or reset |
| Off | On | Off | The node was awakened 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 |

7.6.3 Door Bell Files

The Product\DoorBell directory contains the source code and makefiles for the application. The application uses also a number of utility functions described in section 3.3.12.

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

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

DoorBell.mpw / DoorBell_....Uv2

uVision3 multiproject workspace file (*.mpw) that unifies all the *.Uv2 project files. There is a project file for every particular target with respect to single chip series and frequency.

7.7 Door Lock Sample Code

The ZDK contains sample code for a non-secure and secure Door Lock sample application. This device shows an example of how a door lock system could be build.

A controller such as the Development Controller and secure Development Controller (Atmega) can control the Door Lock. The Door Lock uses the frequently listening mode (FLiRS) where it powers up the radio for a short period 1000ms@ZW0201 / 1000ms@ZW0301 and in case a wakeup beam for this particular node is detected then it stay awake to receive a command. It is now possible to turn the LED on/off indicating lock/unlock status. After receiving one command, it returns to frequently listening mode again to conserve battery consumption.

The Door Lock is based on the enhanced slave library and it has its generic device class set to Entry Control and the specific device class set to Door Lock.

The none-secure Door Lock lists the following supported command classes in the Node Information Frame:

- Lock command class
- Powerlevel command class
- Version command class
- Manufacturer Specific command class

When included non-secure the secure Door Lock application lists the following supported command classes in the Node Information Frame:

Non-Secure Included

- Lock command class
- Powerlevel command class
- Version command class
- Manufacturer Specific command class
- Security command class

Secure Included

When included secure the secure Door Lock lists the following supported command classes in the Node Information Frame:

- Version command class
- Manufacturer Specific command class
- Security command class

The following listed in the Security Commands Supported Report frame:

- Lock command class
- Powerlevel command class
- Version command class
- Manufacturer Specific command class

The Basic command class is secure because application does not list it in Node Information Frame.

During initialization, the Door Lock will initialize the mounted button and one LED. 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 Door Lock main function. The **ApplicationPoll** function checks button activation and act according to the state the Door Lock is in. The other main function is the **ApplicationCommandHandler** function that is called every time a command has been received, destined for the Door Lock. This function checks the command and acts according to the command.

7.7.1 Network Wide Inclusion

By default the Door Lock will enter network wide inclusion (NWI) when it is powered up and have not already been included. The Door Lock will stay in NWI mode for 4 minutes or until it has been included into the network. Any key press will terminate the NWI mode and the only way to make the Door Lock enter NWI mode again is by doing hardware reset either by remove and reapply the power or press the reset button on the side of the board. Refer to section 7.2.1 regarding implementation details.

7.7.2 Interface

The following table defines the functionality of the button on the Z-Wave module.

| | Button Triple Pressed | Button Clicked |
|----------------|------------------------------------|----------------------|
| In Network | Node Info Frame / Enter learn mode | Toggle on/off status |
| Not in Network | Node Info Frame / Enter learn mode | Toggle on/off status |

Learn mode is now activated by pressing the button three times within 1.5 seconds to avoid unintentional inclusion/exclusion of the node.

7.7.3 Secure Door Lock Files

The Product\DoorLock directory contains the source code and makefiles for the application. The application uses also a number of utility functions described in section 3.3.12.

Mk.bat

This batch file controls the build process of the sample applications by calling the appropriate makefiles and parameter settings.

Makefile / Makefile.SecureTargets

These files define the possible targets to build in this directory. It creates the directory structure and defines targets. The following compiler control line defines are used in the makefiles:

ANZ : Build Australia/New Zealand frequency targets
EU : Build European frequency targets
HK : Build Hong Kong frequency targets

IN : Build India frequency targets

MY : Build Malaysia frequency targets

US : Build US frequency targets

Makefile.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.

eprom.h

This header file contains the address definitions in the external EEPROM used to store application data.

DoorLock.c + DoorLock.h

This file contains the source code for the non-secure and secure Door Lock sample application

DoorLock....mpw / DoorLock_Secure_....Uv2

uVision3 multiproject workspace file (*.mpw) that unifies all the *.Uv2 project files. There is a project file for every particular target with respect to non-secure/secure support, single chip series and frequency.

7.7.3.1 Macros for accessing the Lock/Unlock

PIN_ON(pin)

Set output pin to 1.

Parameter:

pin - Z-Wave pin name

Example:

```
PIN_ON(TRIACpin); /* turn TRIACpin on */
```

PIN_OFF(pin)

Set output pin to 0.

Parameter:

pin - Z-Wave pin name

Example:

```
PIN_OFF(TRIACpin); /* turn TRIACpin off */
```

PIN_GET(pin)

Read pin value.

Parameter:

pin - Z-Wave pin name

Example:

```
PIN_GET(SSN);      /* Read pin SSN value*/
```

PIN_IN(pin, pullup)

Set I/O pin as input.

Parameter:

pin - Z-Wave pin name

pullup - if not zero activate the internal pullup resistor

Example:

```
PIN_IN(SSN, 0);      /* Set I/O pin SSN as input and activate the internal pullup resistor */
```

7.8 LED Dimmer Sample Code

The Developer's Kit contains sample code for a non-secure and secure LED Dimmer. This device is in effect a light switch with a built in dimmer where the light bulb is substituted with 3 LED's when using ZW0201/ZW0301. A controller such as the secure or non-secure 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.

When included the none-secure LED Dimmer application lists the following supported command classes in the Node Information Frame:

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

When included non-secure the secure LED Dimmer application lists the following supported command classes in the Node Information Frame:

Non-Secure Included

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

Secure Included

When included secure the secure LED Dimmer lists the following supported command classes in the Node Information Frame:

- Version command class
- Manufacturer Specific command class
- Security command class

The following listed in the Security Commands Supported Report frame:

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

The Basic command class is secure because application does not list it in Node Information Frame.

The none-secure LED Dimmer is a routing slave device based on the slave/routing slave API. The secure LED Dimmer is a slave device based on the slave/enhanced slave API. During initialization, the LED Dimmer will initialize the mounted button and the 3 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.

When using ZW0201/ZW0301 as the device containing the LED Dimmer application, it is primarily supposed to be mounted onto a slave interface module. It can also be mounted on a ZDP03A board, however, in this case only two of the three LEDs will be used by the application. This is determined by the architecture of the ZDP03A board.

7.8.1 Network Wide Inclusion

By default the LED Dimmer will enter network wide inclusion (NWI) when it is powered up and have not already been included. The LED Dimmer will stay in NWI mode for 4 minutes or until it has been included into the network. Any key press will terminate the NWI mode and the only way to make the LED Dimmer enter NWI mode again is by doing hardware reset either by remove and reapply the power or press the reset button on the side of the board. Refer to section 7.2.1 regarding implementation details.

7.8.2 Interface

The following table defines the functionality of the button on the Z-Wave module.

| | Button Triple Pressed | Button Clicked | Button is held |
|-----------------------|------------------------------------|----------------------|----------------|
| In Network | Node Info Frame / Enter learn mode | Toggle on/off status | Dim up/down |
| Not in Network | Node Info Frame / Enter learn mode | Toggle on/off status | Dim up/down |

Learn mode is now activated by pressing the button three times within 1.5 seconds to avoid unintentional inclusion/exclusion of the node.

7.8.3 LED_Dimmer Files

The Product\LED_Dimmer directory contains sample source code for a none-secure/secure routing/enhanced slave application on a Z-Wave module. The application uses also a number of utility functions described in section 3.3.12.

MK.BAT

Batch file used to build ZW0201/ZW0301 based sample applications in ANZ, EU, HK, IN, MY and US versions respectively. Refer to chapter 7.1 regarding details about the build procedure.

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:

```

ANZ           : Build ANZ frequency target.
EU            : Build EU frequency target.
HK            : Build HK frequency target.
IN            : Build IN frequency target.
MY            : Build MY frequency target.
US            : Build US frequency target.
NOOFFONDIM    : When enabled not all the LED's will turn off when dimming.
APPL_PROD_TEST: Enable the production test.
```

Makefile.leddimmer_common

This makefile is a common file defining all dependencies etc.

eeeprom.h

This header file defines the addresses where application data are stored in the external EEPROM.

LEDdim.h / LEDdim.c

This file contains the source code for the LED dimmer application state machine. The common API functions such as **ApplicationInitHW**, **ApplicationInitSW**, **ApplicationNodeInformation**, **ApplicationPoll**, **ApplicationSlaveUpdate** and **ApplicationCommandHandler** are defined here.

LED_Dimmer.mpw / LED_Dimmer_....Uv2

uVision3 multiproject workspace file (*.mpw) that unifies all the *.Uv2 project files. There is a project file for every particular target with respect to single chip series and frequency.

7.8.3.1 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 */
```

7.9 MyProduct Sample Code

The My Product contains the minimum framework to begin developing a slave application. To realize the application in question it's often easier to modify the existing sample code applications than build one from scratch based on MyProduct.

7.9.1 MyProduct Files

The Product\MyProduct directory contains sample source code for a routing slave application on a Z-Wave module. The application uses also a number of utility functions described in section 3.3.12.

MK.BAT

Batch file used to build ZW0201/ZW0301 based sample applications in ANZ, EU, HK, IN, MY and US versions respectively. Refer to chapter 7.1 regarding details about the build procedure.

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:

| | |
|-----|-------------------------------|
| ANZ | : Build ANZ frequency target. |
| EU | : Build EU frequency target. |
| HK | : Build HK frequency target. |
| IN | : Build IN frequency target. |
| MY | : Build MY frequency target. |
| US | : Build US frequency target. |

Makefile.MyProduct_common

This makefile is a common file defining all dependencies etc.

MyProduct.h / MyProduct.c

This file contains the source code for the MYProduct. The common API functions such as **ApplicationInitHW**, **ApplicationInitSW**, **ApplicationNodeInformation**, **ApplicationPoll**, **ApplicationSlaveUpdate** and **ApplicationCommandHandler** are defined here.

MyProduct.mpw / MyProduct_....Uv2

uVision3 multiproject workspace file (*.mpw) that unifies all the *.Uv2 project files. There is a project file for every particular target with respect to single chip series and frequency.

7.10 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_proctest_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:

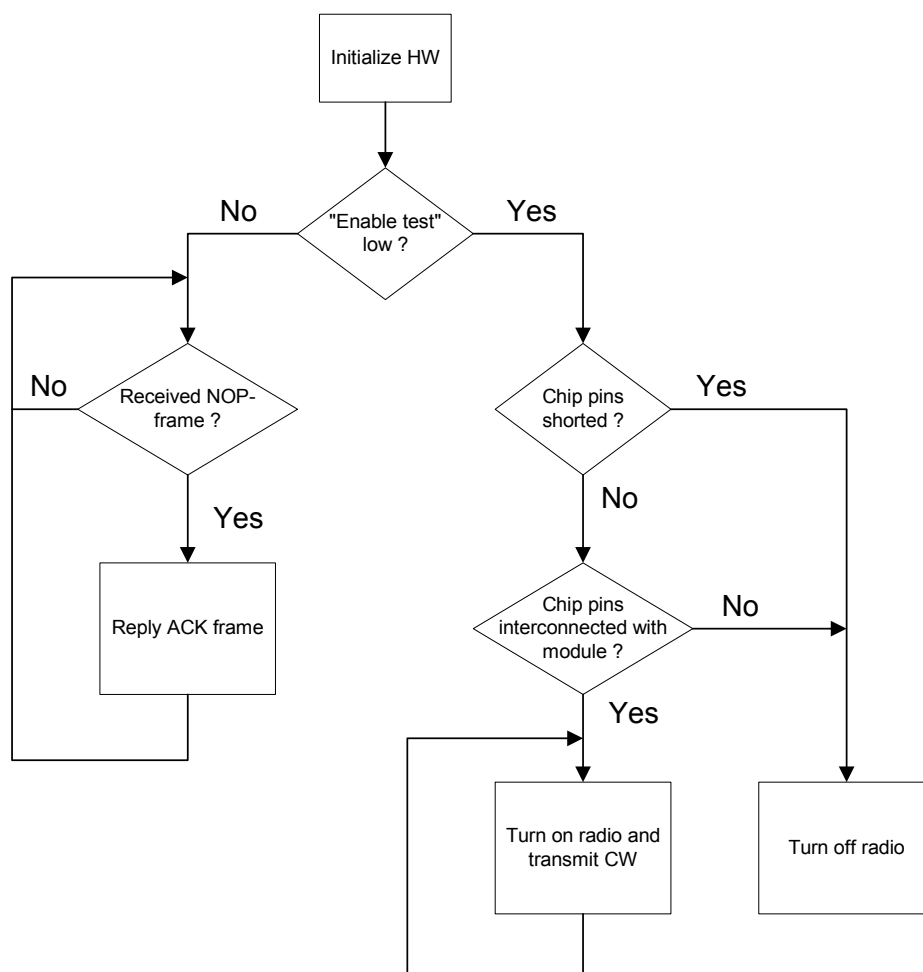


Figure 16. Prod_Test_DUT test program flow

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 from 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.

7.10.1 Production Test DUT Files

The Product\Prod_Test_DUT directory contains the source code for the Production Test DUT sample application.

MK.BAT

Batch file used to build ZW0201/ZW0301 based sample applications in ANZ, EU, HK, IN, MY and US versions respectively. Refer to chapter 7.1 regarding details about the build procedure.

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:

| | |
|-----|-------------------------------|
| ANZ | : Build ANZ frequency target. |
| EU | : Build EU frequency target. |
| HK | : Build HK frequency target. |
| IN | : Build IN frequency target. |
| MY | : Build MY frequency target. |
| US | : Build US 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.

Prod_Test_DUT.mpw / Prod_Test_DUT_....Uv2

uVision3 multiproject workspace file (*.mpw) that unifies all the *.Uv2 project files. There is a project file for every particular target with respect to single chip series and frequency.

7.11 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.

A simple generator consists of a ZW010x Interface Module and a ZMxx20 Z-Wave Module.

On the Interface module there are 6 LED diodes, which have these assignments in the Prod_Test_Gen sample application:

| LED # | Colour | Description |
|-------|--------|------------------------------------|
| D6 | Green | Power on |
| D1 | Red | Error |
| D2 | Red | Success |
| D3 | Red | Send (flashes during transmission) |
| D4 | Red | - |
| D5 | Red | Indication of Push button |

The push button on the ZMxx20 Z-Wave Module is the "Test" button.

After connection to power, the red "Error" LED 'D1' on the Interface module will be on.

When the push button is pressed, 10 NOP's will be transmitted. A device under test (a Prod_Test_DUT) is expected to verify the reception of each NOP with an ACK. During transmission, the red LED 'D3' will flash.

If all NOP's are replied correctly, the red "Error" LED 'D1' will turn off and the red "Success" LED 'D2' will turn on and stay on until the next test is conducted. If the DUT does not reply correctly, the red "Error" LED 'D1' will turn on and stay on until the next test is conducted.

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_proctest_gen library.

The application is controlled via RS232 (115200,8,N,1) or button with fixed timings:
Device will respond 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'

7.11.1 Production Test Generator Files

The Product\Prod_Test_Gen directory contains the source code for the Production Test Generator sample application.

MK.BAT

Batch file used to build ZW0201/ZW0301 based sample applications in ANZ, EU, HK, IN, MY and US versions respectively. Refer to chapter 7.1 regarding details about the build procedure.

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:

| | |
|-----|-------------------------------|
| ANZ | : Build ANZ frequency target. |
| EU | : Build EU frequency target. |
| HK | : Build HK frequency target. |
| IN | : Build IN frequency target. |
| MY | : Build MY frequency target. |
| US | : Build US 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.

Prod_Test_Gen.mpw / Prod_Test_Gen_....Uv2

uVision3 multiproject workspace file (*.mpw) that unifies all the *.Uv2 project files. There is a project file for every particular target with respect to single chip series and frequency.

7.12 Serial API Embedded Sample Code

The purpose of the Serial API embedded sample code is to show how a 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.

7.12.1 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.

7.12.2 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.

7.12.2.1 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) | | | | | | | |

Data frame:

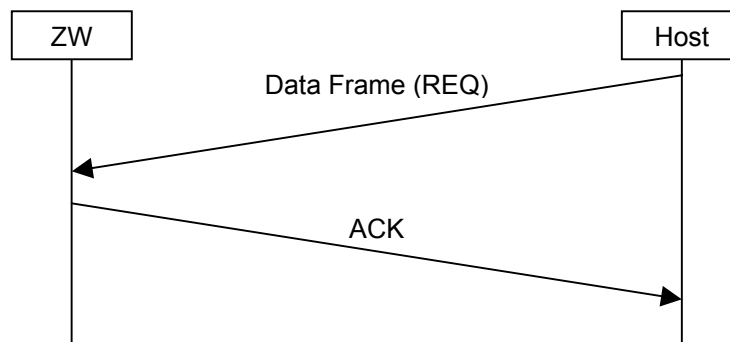
The Data frame contains the Serial API command including parameters for the command in question. The format is as follows:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----------------------|---|---|---|---|---|---|---|
| SOF | | | | | | | |
| Length | | | | | | | |
| Type | | | | | | | |
| Serial API Command ID | | | | | | | |
| Command Specific Data | | | | | | | |
| ... | | | | | | | |
| Checksum | | | | | | | |

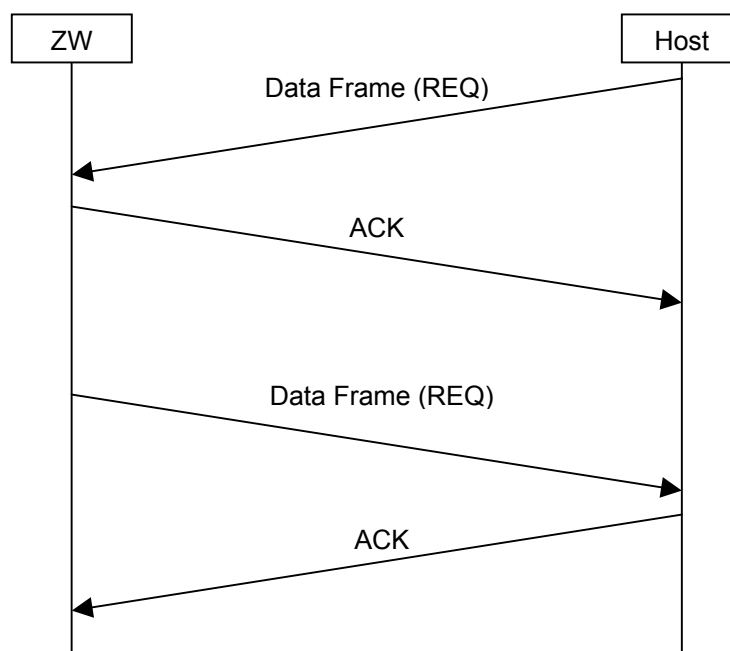
| Field | Description |
|------------------------------|---|
| SOF | Start Of Frame. Used for synchronization and is equal to 0x01 |
| Length | Number of bytes in the frame, exclusive SOF and Checksum. The host application is responsible for entering the correct length field. The current Serial API embedded sample code does no validation of 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 |

7.12.2.2 Frame Flow

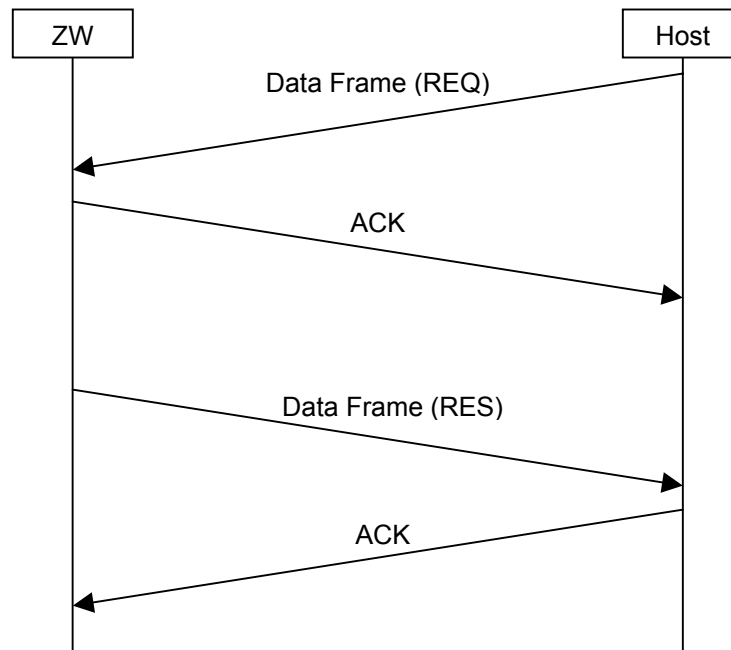
The frame flow between a host and a Z-Wave module (ZW) running the Serial API embedded sample code depends on the API call. There are four different ways to conduct communication between the host and ZW.



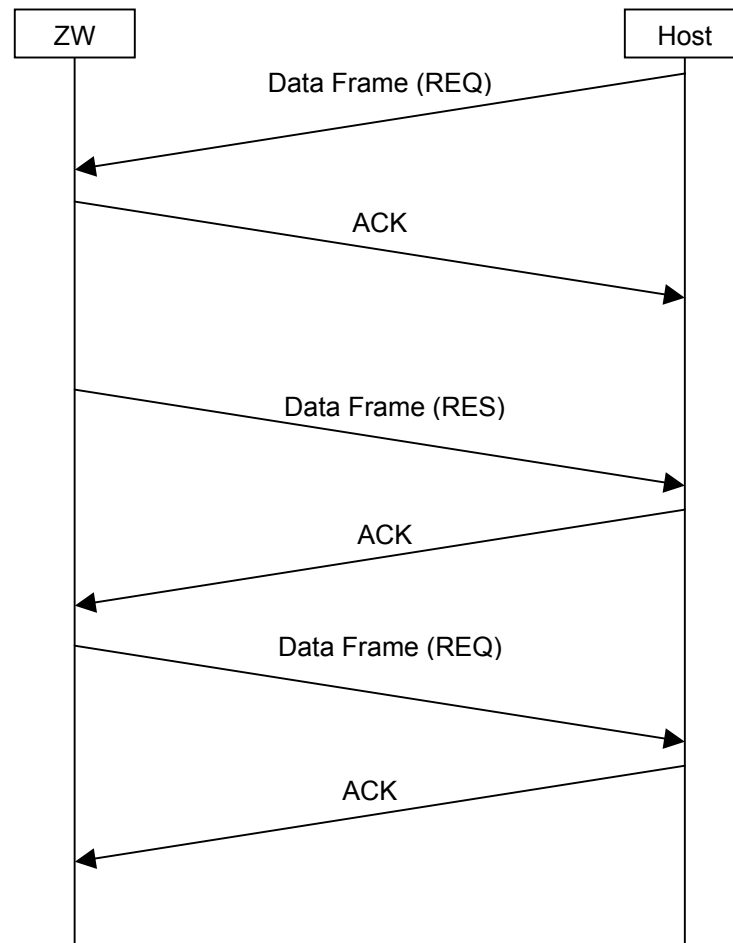
Data frame from host, which is acknowledged by ZW when successfully received. An example could be the API call **ZW_GetSUCNodeID**.



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**.



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**.



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**.

7.12.2.3 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.

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.

7.12.2.4 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.

7.12.2.5 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:

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.

7.12.2.6 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

7.12.2.7 Serial API Watchdog

Some PC based applications cannot guarantee kicking the watchdog before timeout causing the watchdog to reset the Z-Wave single chip unintentionally. The following serial API functions are therefore available to avoid this:

- Start watchdog: Enable watchdog and ApplicationPoll kick watchdog
- Stop watchdog: Disable watchdog and stop kick watchdog in ApplicationPoll

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

Start watchdog handling by using the Serial API function FUNC_ID_ZW_WATCHDOG_START:

Serial API:

HOST->ZW: REQ | 0xD2

Stop watchdog handling by using the Serial API function FUNC_ID_ZW_WATCHDOG_STOP:

Serial API:

HOST->ZW: REQ | 0xD3

7.12.2.8 Serial API Files

The Product\SerialAPI directory contains sample source code for controller/slave applications on a Z-Wave module. The application uses also a number of utility functions described in section 3.3.12.

MK.BAT

Batch file used to build ZW0201/ZW0301 based sample applications in ANZ, EU, HK, IN, MY and US versions respectively. Refer to chapter 7.1 regarding details about the build procedure.

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:

| | |
|------------|-------------------------------|
| ANZ | : Build ANZ frequency target. |
| EU | : Build EU frequency target. |
| HK | : Build HK frequency target. |
| IN | : Build IN frequency target. |
| MY | : Build MY frequency target. |
| US | : Build US frequency target. |
| LOW_FOR_ON | : Not used. |
| SIMPLELED | : Not used. |

Makefile.serialapi_common

This makefile is a common file defining all dependencies etc.

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 / serialappl.h

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.

SerialAPI.mpw / SerialAPI_....Uv2

uVision3 multiproject workspace file (*.mpw) that unifies all the *.Uv2 project files. There is a project file for every particular target with respect to single chip series and frequency.

7.12.3 PC based Controller Sample Application

The PC\Source\SampleApplications\ZWavePCController directory contains sample application source code in C# that implements a PC based Controller using the development tool Visual Studio 2008.

For further information about the features of the PC based Controller, see [6].

7.12.4 PC based Installer Tool Sample Application

The PC\Source\SampleApplications\ZWaveInstaller directory contains sample application source code in C# that implements a PC based Installer Tool using the development tool Visual Studio 2008.

For further information about the features of the PC based Installer Tool, see [7].

7.12.5 PC based Z-Wave Bridge Sample Application

The PC\Source\SampleApplications\ZWaveUPnPBridge directory contains sample application source code in C# that implements a PC based Z-Wave to UPnP Bridge using the development tool Visual Studio 2008.

For further information about the features of the PC based Z-Wave to UPnP Bridge, see [8].

8 TOOL SAMPLE CODE

The Z-Wave Developer's Kit includes tool sample code to enable customization of production environment.

8.1 Z-Wave Programmer Firmware

The ZDK contains sample code that demonstrates how to program the 100/200/300/400 Series ASIC. The ZDP02A Z-Wave Development Platform [31] or ZDP03A Z-Wave Development Platform [32] supports this purpose. The Z-Wave Programmer firmware resides on the AVR ATmega128 chip on ZDP03A and controlled by the PC based Z-Wave Programmer application [14]. For a detailed description of the communication protocol between the AVR and PC based Z-Wave Programmer application, refer to [34].

Source code developed in the following environment:

- WinAVR v20071221:
 - o GNU Binutils 2.18 (including assembler, linker, etc.)
 - o Compiler Collection (GCC) 4.2.2
 - o avr-libc 1.6.0
- Z-Wave Library v2.91
- Keil uVision PK51 v9

Project environment:

- Eclipse Platform v3.5 with plugins:
 - o AVR Eclipse Plugin
 - o (optional) Polarion Subversive SVN Connectors
 - o (optional) Eclipse Subversive - SVN Team Provider Project

The AVR ISP In-System Programmer programs the AVR ATmega128.

8.1.1 ATmega_ZWaveProgFW Files

The Tools\Programmer\ATmega_ZWaveProgFW directory contains the source code for the 400 Series low level programming application.

MK.BAT

Batch file used to build AVR based sample applications in versions for the firmware update (via Z-Wave Programmer) and complete ATmega128 firmware (via AVR ISP In-System Programmer). Refer to chapter 7.1 regarding details about the build procedure.

MAKE_FIRMWARE.BAT

Batch file used to make complete ATmega128 firmware from bootloader firmware and firmware update. Called by MK.BAT.

MAKE_MTP.BAT

Batch file used to build the ZW040x Execute Out of SRAM application, that give the ability to the ATmega128 firmware to access the MTP memory of the ZW040x chip. Called by MK.BAT.

.cproject; .project; .settings

Project files of the Eclipse IDE used to edit AVR based sample application source code.

src\ATmega_spi.c; .h

Source code of the implementation of the software SPI, which is connected to the Z-Wave Module.

src\commands.h

This header file contains definitions of the commands of the Z-Wave Programmer Communication Protocol [34].

src\conhandle.c; .h

Source files, contains the functions for handling the Programmer frames via the UART.

src\leeprom_if.c; .h

Source code of the Z-Wave Module External EEPROM interface. Reading / writing of the Z-Wave Module External EEPROM via the software SPI was implemented.

src\mtp.c; .h

Source code of the ZW040x Execute out of SRAM application, which implements the ZW040x MTP memory interface.

src\ports.h

Header file with definitions of port names of the ATmega128 in ZDP02 (ZDP03) board.

src\UART_buf_io.c; .h

Source code of buffered transmit/receive of data through the UART.

src\ZWaveFlash.c; .h

Main source code of the Z-Wave Programmer Firmware. Contains the implementation of all programmer commands handlers and Z-Wave chips programming algorithms.

9 REQUIRED DEVELOPMENT COMPONENTS

9.1 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 v9 Professional Developer's Kit for the 8051 microcontroller:

Z-Wave libraries and sample applications are built and tested on above version 9 but newer versions 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 Sigma Designs.

| | | | |
|---|--|---|--|
| Keil Software, Inc. 1501 10th Street, Suite 110 Plano, TX 75074 USA | | Keil Elektronik GmbH Bretonischer Ring 15 D-85630 Grasbrunn Germany | |
| Toll Free: | 800-348-8051 | Toll Free: | - |
| Phone: | 972-312-1107 | Phone: | (49) (089) 45 60 40 0 |
| Fax: | 972-312-1159 | Fax: | (49) (089) 46 81 62 |
| Sales: | sales.us@keil.com | Sales: | sales.intl@keil.com |
| Support: | support.us@keil.com | Support: | support.intl@keil.com |

9.2 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 9.7.

9.3 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].

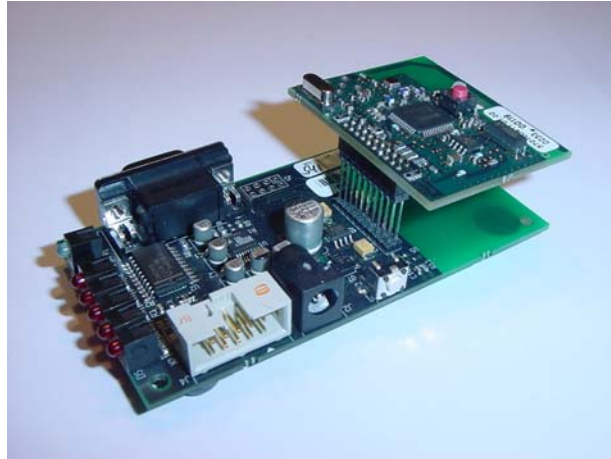


Figure 17. ZW0102 Controller/Slave Unit

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 18. ZW0102 Development Controller Unit

9.4 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].

9.5 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].

9.6 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.

9.7 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 (xxxxxxx) 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.

[illegible]

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:

1. Use the Z-Wave Programmer [14] to download eeploader_ZW0102.hex file to the module.
2. 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.
3. 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.
4. 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.
5. The PC application displays download progress and finally download status.

10 APPLICATION NOTE: SUC/SIS IMPLEMENTATION

10.1 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.

10.2 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.

10.2.1 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.

10.2.1.1 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.

10.2.2 Updates From The Primary Controller

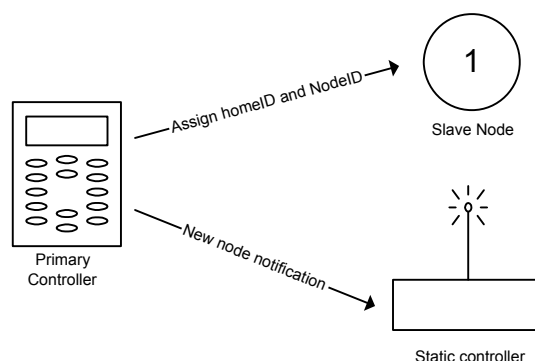


Figure 19. Inclusion of a node having a SUC in the network

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.

10.2.3 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.

10.2.4 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.

10.2.5 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.

10.3 The Primary Controller

The primary controller is responsible for choosing what static controller in the network that should act as a SUC and it will also send notifications to the SUC about all changes in the network topology. The application in a primary controller is responsible for choosing the static controller that should be the SUC. There is no fixed strategy for how to choose the static controller, so it is entirely up to the application to choose the controller that should become SUC. Once a static controller has been selected the

application must use the **ZW_SetSUCNodeID** to request that the static controller becomes SUC. The capabilities parameter in the **ZW_SetSUCNodeID** call will determine if the primary controller enables the ID Server functionality in the SUC.

Once a SUC has been selected the protocol in the primary controller will automatically send notifications to the SUC about all changes in the network topology.

NOTE: A static controller can decline the role as SUC and in that case the callback function from **ZW_SetSUCNodeID** will return with a FAILED status. The static controller can also refuse to become SIS if that was what the primary controller requested, but accept to become a SUC.

10.4 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.

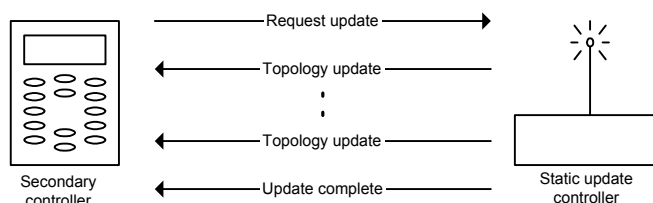


Figure 20. Requesting network updates from a SUC in the network

10.4.1 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.

10.4.2 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 than 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.

10.5 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 10.4 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.

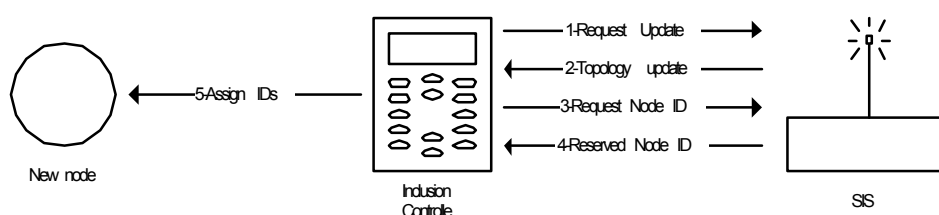


Figure 21. Inclusion of a node having a SIS in the network

10.6 Routing Slaves

The routing slave can request a update of its stored return routes from a SUC by using the **ZW_RequestNetWorkUpdate** API call. There is no API call in the routing slave to check if the SUC is known by the slave so the application must just try **ZW_RequestNetWorkUpdate** and then determine from the return value if the SUC is known or not. If the SUC was known and the update was a success then the routing slave would get a callback with the status **SUC_UPDATE_DONE**, the slave will not get any notifications about what was changed in the network.

A static update controller (SUC) can help a battery-operated routing slave to be re-discovered in case it is moved to a new location. The lost slave initiates the re-discovery process because it will be the first to recognize that it is unable to reach the configured destinations and therefore can the application call **ZW_RediscoveryNeeded** to request help from other nodes in the network.

The lost battery operated routing slave start to send "I'm lost" frames to each node beginning with node ID = 1. It continue until it find a routing slave which can help it, i.e. the helping routing slave can obtain contact with a SUC. Scanning through the node ID's is done on application level. Other strategies to send the "I'm lost" frame can be implemented on the application level.

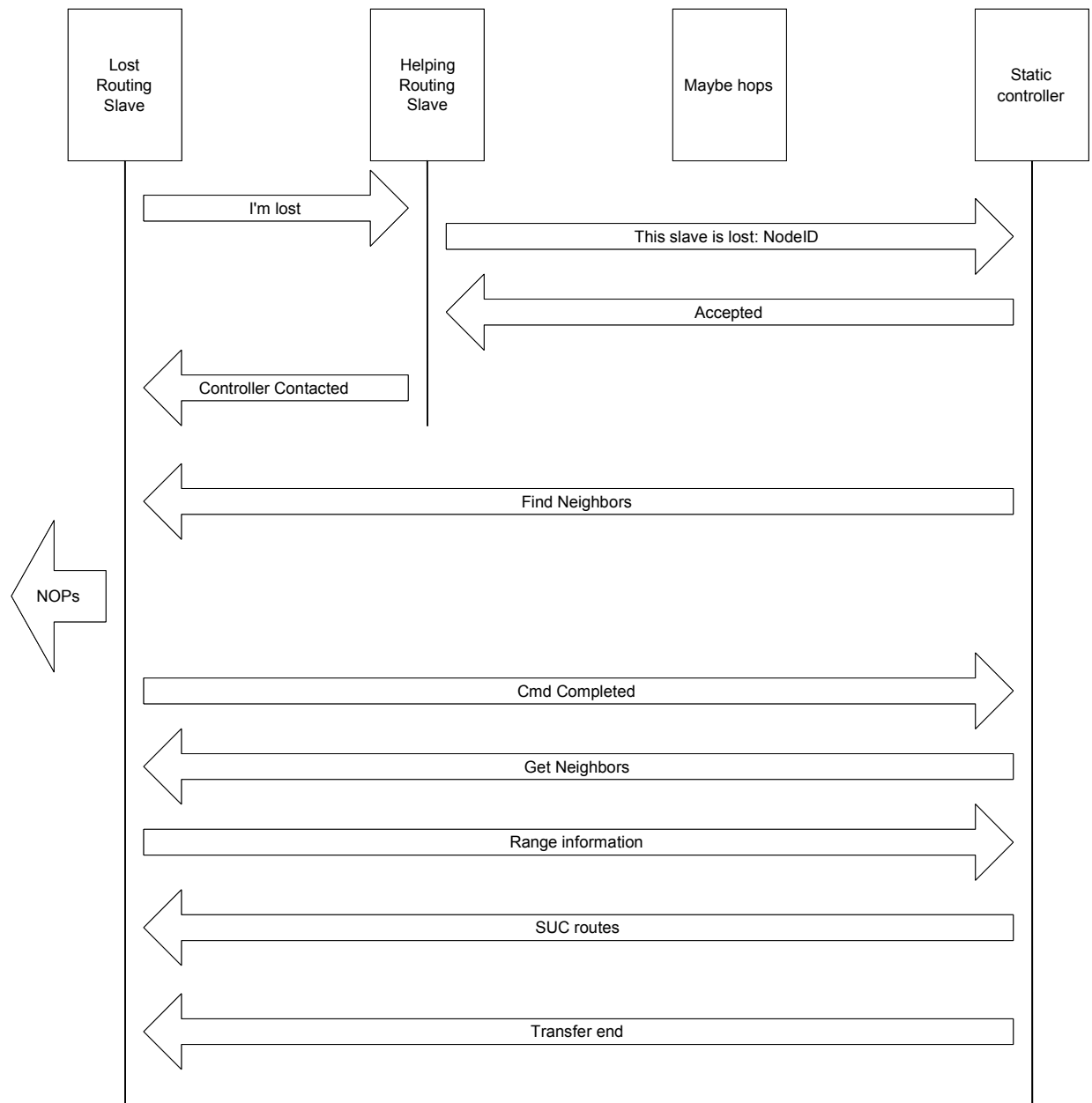


Figure 22. Lost routing slave frame flow

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.

11 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.

11.1 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.

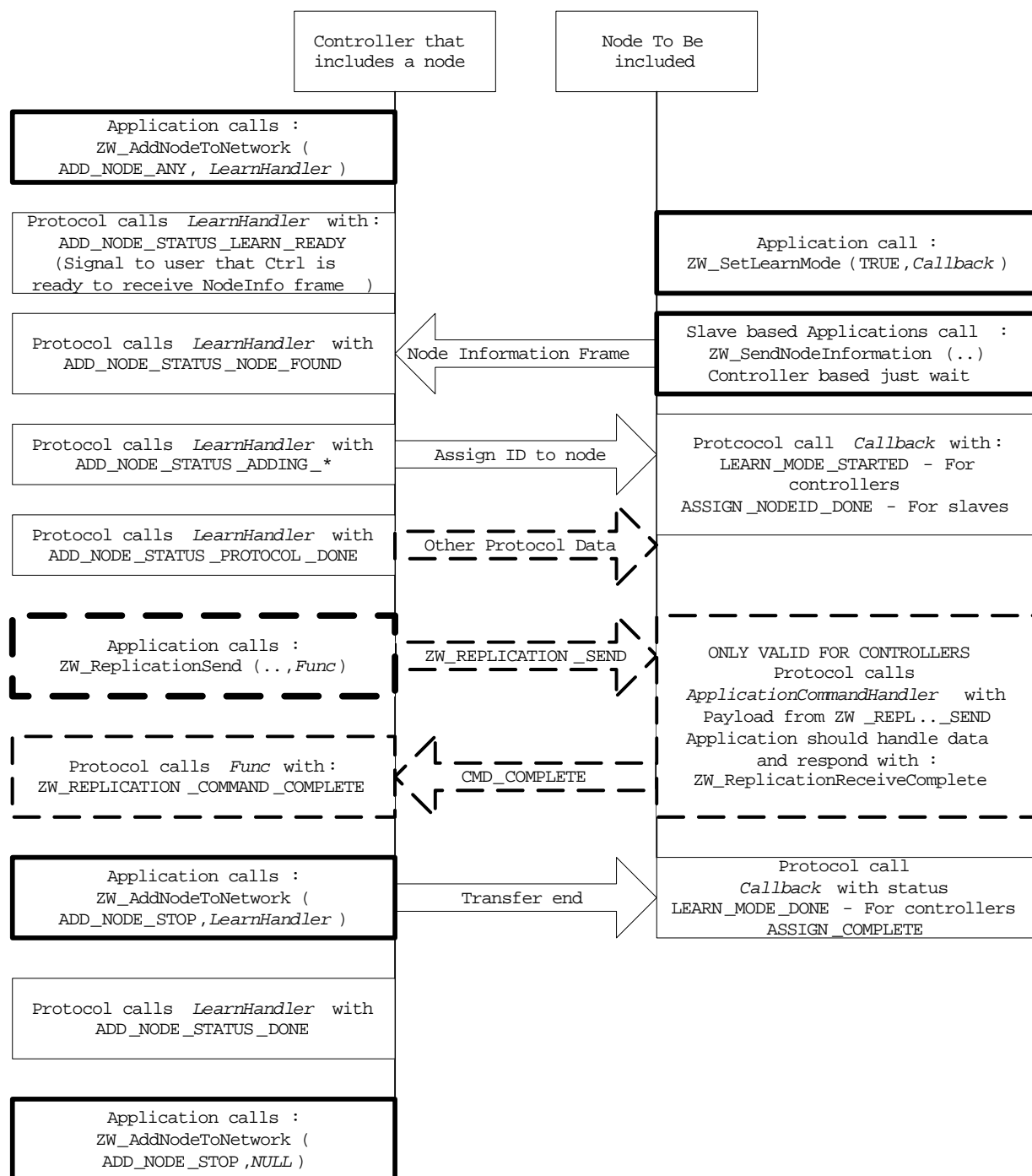


Figure 23. Node inclusion frame flow

Legend:

1. Bold frames indicate that the Application initiates an action.
2. Dashed frames indicate optional steps and frame flows.
3. *Italic* indicates a callback function specified by the application.

To allow the primary/inclusion controller in a Z-Wave network to include all kind of nodes, it is necessary to have a frame that describes the capabilities of a node. Some of the capabilities will be protocol related and some will be application specific. All nodes will automatically send out their Node Information Frame

when the action button on the node is pressed. Once a node is included into the network it can always at a later stage get the node information from a node by requesting it with the API call **ZW_RequestNodeInfo**).

All slave nodes will per default start with Home ID is 0x00000000 and Node ID 0x00. All controllers will per default start with a unique Home ID and Node ID 0x01. Both have to be changed before the node can be included into a network. Furthermore the node must enter a learn mode state in order to accept assignment of new ID's. That state is communicated from the node by sending out a Node Information Frame as described. The primary/inclusion controller can now assign a Home and Node ID to the node to be included in the Z-Wave network. In case the node is already included to a network then the primary/inclusion controller refuses to include it.

During "Other protocol data" the network topology is discovered and updated. The primary/inclusion controller request the new node to check which of the current nodes in the network it can communicate directly 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**.

The following code sample shows how add node functionality is implemented on a controller capable of adding nodes to the network:

```

/* Call to be performed when user/application wants to include a node to the network
*/
ZW_AddNodeToNetwork(ADD_NODE_ANY, LearnHandler);

/*===== LearnHandler =====
**      Function description
**      Callback function to ZW_ADD_NODE_TO_NETWORK
**-----*/
void LearnHandler(LEARN_INFO *learnNodeInfo)
{
    if (learnNodeInfo->bStatus == ADD_NODE_STATUS_LEARN_READY)
    {
        /* Application should now signal to the user that we are ready to add a node.
        User may still choose to abort */
    }
    else if (learnNodeInfo->bStatus == ADD_NODE_STATUS_NODE_FOUND)
    {
        /* Protocol is busy adding node. User interaction should be disabled */
    }
    else if (learnNodeInfo->bStatus == ADD_NODE_STATUS_ADDING_SLAVE)
    {
        /* Protocol is still busy, this is just an information that it is a slave based
        unit that is being added */
    }
    else if (learnNodeInfo->bStatus == ADD_NODE_STATUS_ADDING_CONTROLLER)
    {
        /* Protocol is still busy, this is just an information that it is a controller
        based unit that is being added */
    }
    else if (learnNodeInfo->bStatus == ADD_NODE_STATUS_PROTOCOL_DONE)
    {
        /* Protocol is done. If it was a controller that was added, the application can
        now transfer information with ZW_ReplicationSend if any applications specific
        data that needs to be transferred to the included controller at inclusion time
        */

        /* When application is done it informs the protocol */
        ZW_AddNodeToNetwork(ADD_NODE_STOP, LearnHandler);
    }
    else if (learnNodeInfo->bStatus == ADD_NODE_STATUS_FAILED)
    {
        /* Add node failed - Application should indicate this to user */
        ZW_AddNodeToNetwork(ADD_NODE_STOP_FAILED, NULL);
    }
    else if (learnNodeInfo->bStatus == ADD_NODE_STATUS_DONE)
    {
        /* It is recommended to stop the process again here */
        ZW_AddNodeToNetwork(ADD_NODE_STOP, NULL);

        /* Add node is done. Application can move on Now is a good time to check if the
        added node should be set as SUC or SIS */
    }
}

```

The following code samples show how an application typically implement the code needed in order to be able to include itself in an existing network.

Sample code for controller based devices:

```
/* Call to be performed when a controller wants to be include in the network */
ZW_SetLearnMode (TRUE, InclusionHandler);
/*Controller based devices just wait for the learn process to start*/

/*===== InclusionHandler =====
**                      Callback function to ZW_SetLearnMode
**-----*/
void InclusionHandler(
LEARN_INFO *learnNodeInfo)
{
    if ((*learnNodeInfo).bStatus == LEARN_MODE_STARTED)
    {
        /* The user should no longer be able to exit learn mode.
        ApplicationCommandHandler should be ready to handle ZW_REPLICATION_SEND_DATA
        frames if it supports transferring of Application specific data* /
    }
    else if ((*learnNodeInfo).bStatus == LEARN_MODE_FAILED)
    {
        /* Something went wrong - Signal to user */
    }
    else if ((*learnNodeInfo).bStatus == LEARN_MODE_DONE)
    {
        /* All data have been transmitted. Capabilities may have changed. Might be a
        good idea to read ZW_GET_CONTROLLER_CAPABILITIES() and to check that
        associations still are valid in order to check if the controller have been
        included or excluded from network*/
    }
}
```

Sample code for slave based devices:

```

/* Call to be performed when a slave wants to be include in the network */
ZW_SetLearnMode(TRUE, InclusionHandler);
ZW_SendNodeInformation(NODE_BROADCAST, TRANSMIT_OPTION_LOW_POWER,...);

/*===== InclusionHandler =====
**                               Callback function to ZW_SetLearnMode
**-----*/
void InclusionHandler
    BYTE bStatus /* IN Current status of Learnmode*/
    BYTE nodeID) /* IN resulting nodeID - If 0x00 the node was removed from network*/
{
    if(bStatus == ASSIGN_RANGE_INFO_UPDATE)
    {
        /* Application should make sure that it does not send out NodeInfo now that we
        are updating range */
    }
    if(bStatus == ASSIGN_COMPLETE)
    {
        /* Assignment was complete. Check if it was inclusion or exclusion and maybe
        tell user we are done */
        if (nodeID != 0)
        {
            /* Node was included in a network*/
        }
        else
        {
            /* Node was excluded from a network. Reset any associations */
        }
    }
    else if (bStatus == ASSIGN_NODEID_DONE)
    {
        /* ID is assigned. Protocol will call with bStatus=ASSIGN_COMPLETE when done */
    }
}

```

11.2 Excluding nodes from the network

The API calls required by the controller that exclude and the device that is to be excluded is described. The callbacks as well as the steps the protocol takes without any application level involvement is also described. Finally it illustrates the frame flow between the two devices during the exclusion process.

The Z-Wave API calls **ZW_RemoveNodeFromNetwork** and **ZW_SetLearnMode** are used to exclude nodes from a Z-Wave network. The primary/inclusion controller use the API call **ZW_RemoveNodeFromNetwork** when removing a node from a network and **ZW_SetLearnMode** is used by the controller or slave node that is to be removed.

For the primary/inclusion controller that is including a node the **ZW_RemoveNodeFromNetwork** is called with either:

- REMOVE_NODE_ANY - Remove any type of node from the network
- REMOVE_NODE_SLAVE - Only remove a node based on slave libraries
- REMOVE_NODE_CONTROLLER - Only remove a node based on controller libraries

To avoid the need to differentiate on the user interface whether it is a controller or slave the REMOVE_NODE_ANY can be used. The application can decide which actions to take based on the callback values.

REMOVE_NODE_SLAVE and REMOVE_NODE_CONTROLLER are available to support backward compatibility in case they are used on devices with separate slave and controller exclusion procedures.

The figure below illustrates the exclusion process between a primary/inclusion controller and a node that the user wishes to exclude from the network.

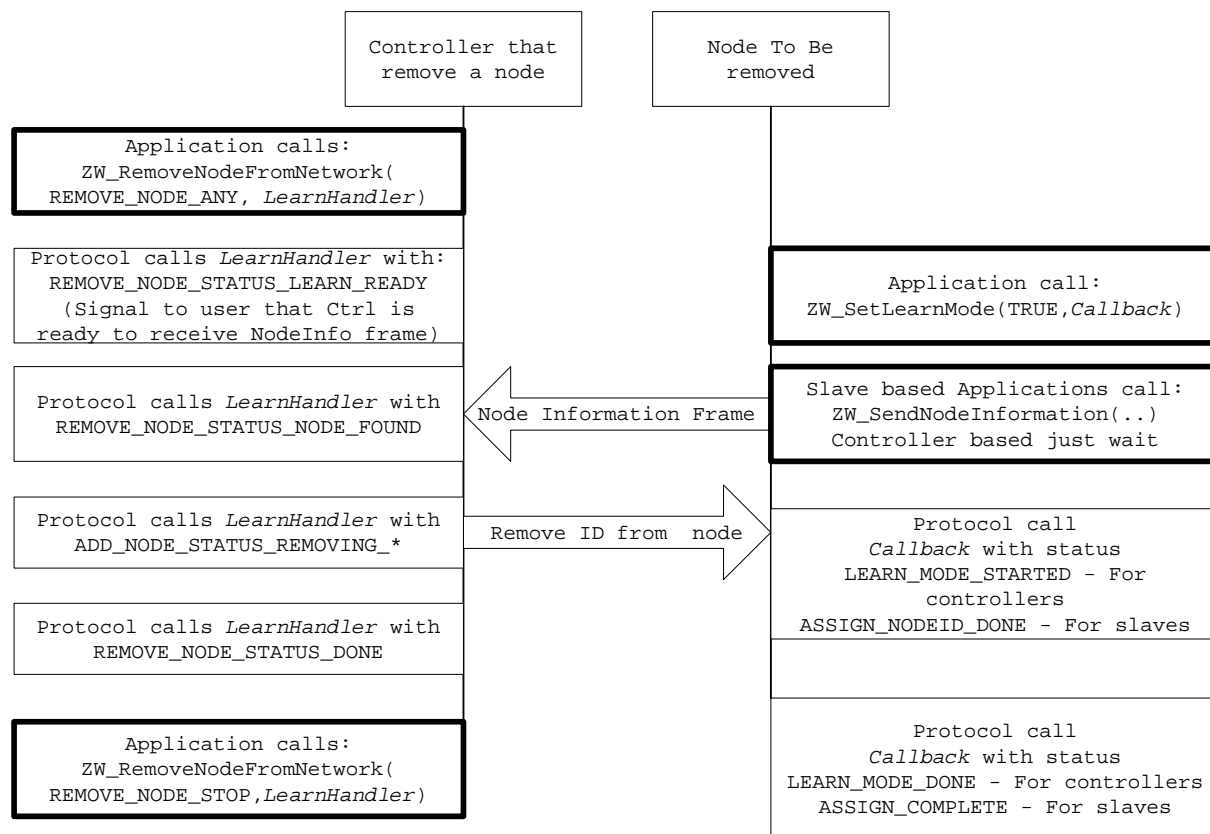


Figure 24. Node exclusion frame flow

The following code sample shows how remove node functionality is implemented on a controller capable of removing nodes from the network:

```
/* Call to be performed when user/application wants to remove a node from the network
*/
ZW_RemoveNodeFromNetwork(REMOVE_NODE_ANY, LearnHandler);

/*===== LearnHandler =====
**      Function description
**      Callback function to ZW_RemoveNodeFromNetwork
**-----*/
void LearnHandler(LEARN_INFO *learnNodeInfo)
{
    if (learnNodeInfo->bStatus == REMOVE_NODE_STATUS_LEARN_READY)
    {
        /* Application should now signal to the user that we are ready to remove a node.
        User may still choose to abort */
    }
    else if (learnNodeInfo->bStatus == REMOVE_NODE_STATUS_NODE_FOUND)
    {
        /* Protocol is busy removing node. User interaction should be disabled */
    }
    else if (learnNodeInfo->bStatus == REMOVE_NODE_STATUS_REMOVING_SLAVE)
    {
        /* Protocol is still busy, this is just an information that it is a slave based
        unit that is being removed*/
    }
    else if (learnNodeInfo->bStatus == REMOVE_NODE_STATUS_REMOVING_CONTROLLER)
    {
        /* Protocol is still busy, this is just an information that it is a controller
        based unit that is being removed */
    }
    else if (learnNodeInfo->bStatus == REMOVE_NODE_STATUS_DONE)
    {
        /* Node is no longer part of the network*/

        /* When done - stop the process with */
        ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP, NULL);
    }
    else if (learnNodeInfo->bStatus == ADD_NODE_STATUS_FAILED)
    {
        /* Remove node failed - Application should indicate this to user */
        ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP, NULL);
    }
}
}
```

For the device that is excluded, the process is no different from an inclusion See paragraph 11 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.

12 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 successful change the controller that called **ZW_ControllerChange** will be secondary and no longer able to include devices.

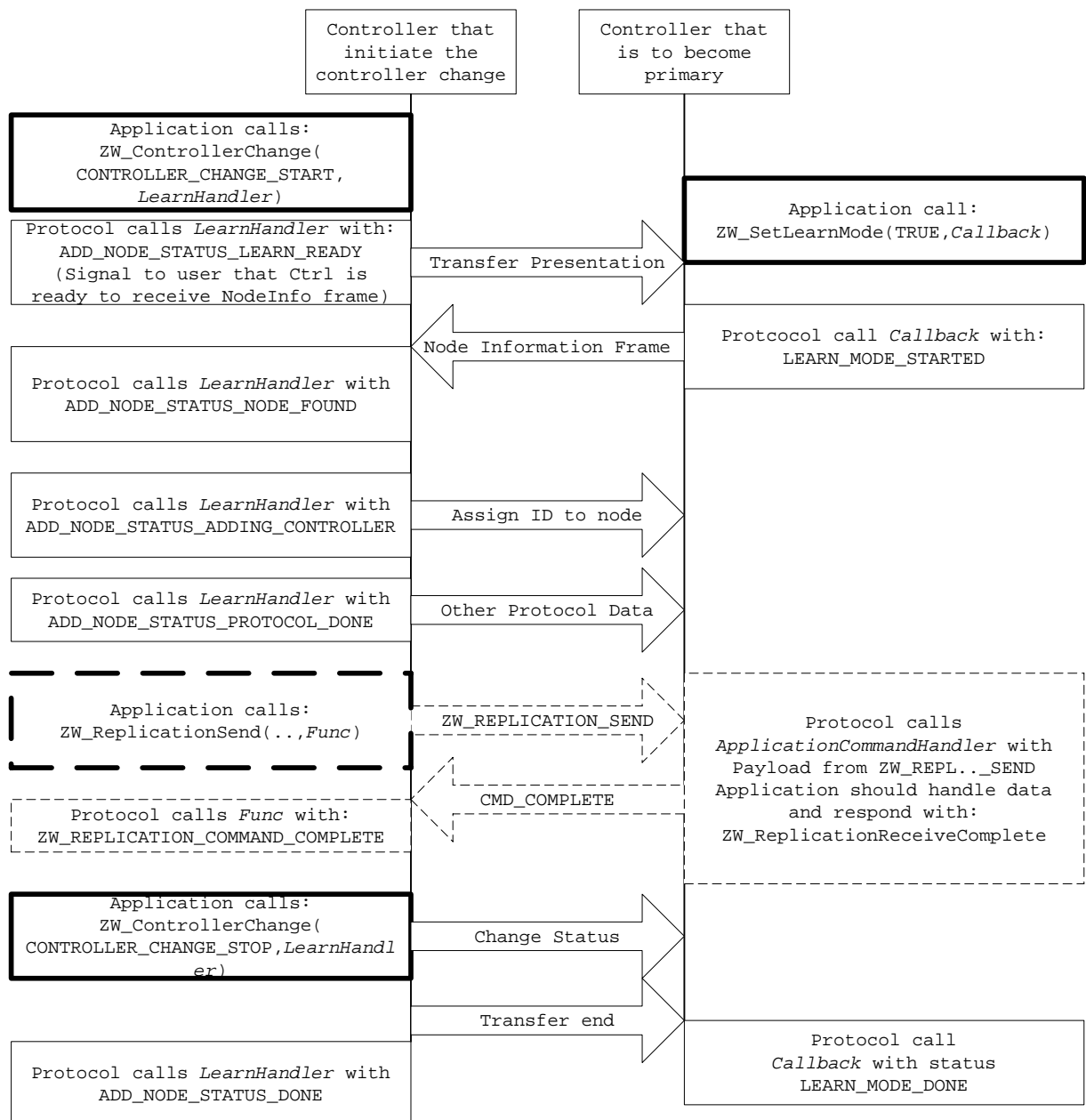


Figure 25. Controller shift frame flow

13 REFERENCES

- [1] SD, SDS10242, Software Design Specification, Z-Wave Device Class Specification.
- [2] SD, DSH10086, Datasheet, ZW0x0x Z-Wave Interface Module.
- [3] SD, DSH10087, Datasheet, ZW0x0x Z-Wave Development Module.
- [4] SD, DSH10033, Datasheet, ZM1220 Z-Wave Module.
- [5] SD, DSH10034, Datasheet, ZM1206 Z-Wave Module.
- [6] SD, INS10240, Instruction, PC Based Controller User Guide.
- [7] SD, INS10241, Instruction, PC Installer Tool Application User Guide.
- [8] SD, INS10245, Instruction, Z-Wave Bridge User Guide.
- [9] SD, INS10029, Instruction, ZW0102 Single Chip Implementation Guideline.
- [10] SD, APL10312, Application Note, Programming the 200 and 300 Series Z-Wave Single Chip Flash.
- [11] SD, INS10336, Instruction, Z-Wave Reliability Test Guideline.
- [12] SD, INS10249, Instruction, Z-Wave Zniiffer User Guide.
- [13] SD, INS10250, Instruction, Z-Wave DLL User's Manual.
- [14] SD, INS10679, Instruction, Z-Wave Programmer User Guide.
- [15] SD, INS10236, Instruction, Development Controller User Guide.
- [16] SD, INS10579, Instruction, Programming the ZW0102 Flash and Lock Bits.
- [17] SD, DSH10088, Datasheet ZMxx06 Converter Module.
- [18] SD, DSH10230, Datasheet, ZM2106C Z-Wave Module.
- [19] SD, INS10326, Instruction, ZW0201 Single Chip Implementation Guidelines.
- [20] SD, SRN11886, Software Release Note, ZW0201/ZW0301 Developer's Kit v4.53.00.
- [21] SD, APL10512, Application Note, Battery Operated Applications Using the ZW0201/ZW0301.
- [22] SD, DSH10856, Datasheet, ZM3106C Z-Wave Module.
- [23] SD, DSH10275, Datasheet, ZM2120C Z-Wave Module.
- [24] SD, DSH10857, Datasheet, ZM3120C Z-Wave Module.
- [25] SD, APL10292, Application Note, ZW0102 Triac Controller Guideline.
- [26] SD, APL10370, Application Note, ZW0201/ZW0301 Triac Controller Guideline.
- [27] SD, APL10514, Application Note, The ZW0201/ZW0301 ADC.
- [28] SD, INS10680, Instruction, Z-Wave XML Editor.
- [29] SD, INS11018, Instruction, Secure PC Based Controller User Guide (OBSOLETE, see INS10240).
- [30] SD, INS10681, Instruction, Secure Development Controller (AVR) User Guide.
- [31] SD, DSH10704, Datasheet, ZDP02A Z-Wave Development Platform.
- [32] SD, DSH11243, Datasheet, ZDP03A Z-Wave Development Platform.
- [33] SD, SDS11060, Software Design Specification, Z-Wave Command Class Specification.
- [34] SD, INS11072, Instruction, Z-Wave Programmer Communication Protocol.
- [35] SD, INS11552, Instruction, 400 Series Crystal Calibration User Guide

INDEX

A

| | |
|---|-----------------------|
| ADC_Buf | 141 |
| ADC_GetRes | 145 |
| ADC_Init | 138 |
| ADC_Int | 144 |
| ADC_IntFlagClr | 145 |
| ADC_Off | 137 |
| ADC_SelectPin | 140 |
| ADC_SetAZPL | 142 |
| ADC_SetResolution | 142 |
| ADC_SetThres | 144 |
| ADC_SetThresMode | 143 |
| ADC_Start | 137 |
| ADC_Stop | 137 |
| AES128 encryption/decryption | 27, 239 |
| App_RFSetup.a51 | 215 |
| Application area in non volatile memory | 131 |
| ApplicationCommandHandler (Not Bridge Controller library) | 65 |
| ApplicationCommandHandler_Bridge (Bridge Controller library only) | 73 |
| ApplicationControllerUpdate | 54, 82, 181, 276, 277 |
| ApplicationControllerUpdate (All controller libraries) | 71 |
| ApplicationInithw | 62 |
| ApplicationInitSW | 63 |
| ApplicationNodeInformation | 67 |
| ApplicationPoll | 64 |
| ApplicationSlaveNodeInformation (Bridge Controller library only) | 75 |
| ApplicationSlaveUpdate | 211 |
| ApplicationSlaveUpdate (All slave libraries) | 70 |
| ApplicationTestPoll | 63 |
| AVR ATmega128 | 14, 26 |

C

| | |
|---------------------------|-----|
| Calibration | 30 |
| Command prompt | 221 |
| Crystal calibration | 30 |

D

| | |
|---------------|-----|
| DOS box | 221 |
|---------------|-----|

E

| | |
|---------------------------|---------|
| Eeploader.exe | 274 |
| EEPROM_APPL_OFFSET | 61, 131 |
| Enhanced Slave | 162 |
| Extern_eep.hex file | 274 |
| External EEPROM | 43, 274 |

F

| | |
|-----------------------------------|-----|
| FCC compliance test | 216 |
| FLASH_APPL_FREQ_OFFS | 215 |
| FLASH_APPL_LOW_POWER_OFFS | 215 |
| FLASH_APPL_MAGIC_VALUE_OFFS | 215 |

| | |
|--|-----|
| FLASH_APPL_NORM_POWER_OFFS..... | 215 |
| FLASH_APPL_PLL_STEPUP_OFFS | 215 |
| FUNC_ID_SERIAL_API_GET_CAPABILITIES..... | 262 |
| FUNC_ID_SERIAL_API_SET_TIMEOUTS..... | 262 |
| FUNC_ID_SERIAL_API_SOFT_RESET..... | 263 |

G

| | |
|--------------------------|-----|
| GP Timer | 128 |
| GP Timer interrupt | 130 |

I

| | |
|------------------------------------|---------|
| I/O pins | 217 |
| Inclusion controller..... | 53, 278 |
| Intellectual property rights | 27 |
| Interrupt | 42 |
| Interrupt service routines | 42 |

K

| | |
|---------------|-----|
| Keil | 270 |
| KEILPATH..... | 221 |

L

| | |
|---------------------|-----|
| Listening flag..... | 67 |
| Lock bits..... | 273 |

M

| | |
|--------------------------|-----|
| Make files..... | 220 |
| Memory optimization..... | 59 |
| MemoryGetBuffer | 134 |
| MemoryGetByte | 132 |
| MemoryGetID | 131 |
| MemoryPutBuffer..... | 135 |
| MemoryPutByte | 133 |
| MK.BAT | 220 |

N

| | |
|-----------------------------|---------|
| Node Information Frame..... | 67, 169 |
| NON_ZERO_SIZE..... | 38 |
| NON_ZERO_START_ADDR..... | 38 |

P

| | |
|---|-----------------|
| PIN_GET | 218 |
| PIN_IN | 217 |
| PIN_OFF..... | 218 |
| PIN_ON | 218 |
| PIN_OUT | 217 |
| PIN_TOGGLE..... | 219 |
| Power on reset..... | 90 |
| Primary controller..... | 45, 53, 54, 275 |
| Production test..... | 63 |
| PVT and RF regulatory measurements..... | 31 |
| PWM mode | 129 |
| PWM Timer..... | 128 |

R

| | |
|--------------------------------|--------------|
| Random number generator | 77 |
| Return route | 99, 103, 112 |
| RF frequency | 215 |
| RF settings | 215 |
| RF transmit power levels | 215 |
| Routing slave | 278 |
| Routing Slave | 162 |

S

| | |
|--|-----------------|
| SerialAPI_ApplicationNodeInformation | 69 |
| SerialAPI_ApplicationSlaveNodeInformation | 75 |
| SIS | 58, 82, 278 |
| Static update controller | 52, 54, 57, 275 |
| Stiftung Secure Information and Communication Technologies | 27 |
| Stop mode | 91 |
| SUC | 57, 82 |
| SUC ID Server | 53, 58 |
| SUC/SIS node | 162 |

T

| | |
|-------------------------------|--------------|
| Timer 0 | 42 |
| Timer 1 | 42 |
| Timer 2 | 42 |
| Timer 3 | 42 |
| Timer mode | 129 |
| TimerCancel | 127 |
| TimerRestart | 127 |
| TimerStart | 126 |
| TOOLS DIR | 221 |
| TRANSMIT_OPTION_EXPLORE | 99, 103, 108 |
| TRIAC_Init | 123 |
| TRIAC_Off | 125 |
| TRIAC_SetDimLevel | 125 |
| TXnormal Power | 216 |

U

| | |
|-------------------------------|-----|
| UART_ClearRx | 151 |
| UART_ClearTx | 151 |
| UART_Disable | 151 |
| UART_Enable | 150 |
| UART_Init | 147 |
| UART_Read | 152 |
| UART_RecByte | 148 |
| UART_RecStatus | 147 |
| UART_SendByte | 149 |
| UART_SendNL | 150 |
| UART_SendNum | 149 |
| UART_SendStatus | 148 |
| UART_SendStr | 150 |
| UART_Write | 152 |
| uninitialized RAM bytes | 233 |

W

| | |
|--------------------|----|
| Wakeup beam..... | 98 |
| Watchdog..... | 96 |
| Wut fast mode..... | 91 |
| Wut mode | 91 |

Z

| | |
|--|----------|
| ZDP02A Development module | 14, 26 |
| ZDP03A Development module | 14, 26 |
| ZW_ADC_BUFFER_DISABLE (Macro) | 141 |
| ZW_ADC_BUFFER_ENABLE (Macro) | 141 |
| ZW_ADC_CLR_FLAG (Macro) | 145 |
| ZW_ADC_GET_READING (Macro) | 145 |
| ZW_ADC_INT_DISABLE (Macro) | 144 |
| ZW_ADC_INT_ENABLE (Macro) | 144 |
| ZW_ADC_OFF (Macro) | 137 |
| ZW_ADC_RESOLUTION_12 (Macro)..... | 142 |
| ZW_ADC_RESOLUTION_8 (Macro)..... | 142 |
| ZW_ADC_SELECT_AD1 (Macro) | 140 |
| ZW_ADC_SELECT_AD2 (Macro) | 140 |
| ZW_ADC_SELECT_AD3 (Macro) | 140 |
| ZW_ADC_SELECT_AD4 (Macro) | 140 |
| ZW_ADC_SET_AZPL (Macro) | 142 |
| ZW_ADC_SET_THRESHOLD (Macro)..... | 144 |
| ZW_ADC_START (Macro) | 137 |
| ZW_ADC_STOP (Macro) | 137 |
| ZW_ADC_THRESHOLD_LO (Macro)..... | 143 |
| ZW_ADC_THRESHOLD_UP (Macro)..... | 143 |
| ZW_ADD_NODE_TO_NETWORK (Macro) | 157 |
| ZW_AddNodeToNetwork..... | 157, 280 |
| ZW_ARE_NODES_NEIGHBOURS(Macro) | 160 |
| ZW_AreNodesNeighbours..... | 160 |
| ZW_ASSIGN_RETURN_ROUTE (Macro) | 161 |
| ZW_ASSIGN_SUC_RETURN_ROUTE (Macro)..... | 162 |
| ZW_AssignReturnRoute | 161 |
| ZW_AssignSUCReturnRoute | 162 |
| ZW_CONTROLLER_CHANGE (Macro)..... | 163 |
| ZW_ControllerChange | 163, 288 |
| ZW_CREATE_NEW_PRIMARY_CTRL (Macro)..... | 190 |
| ZW_CreateNewPrimaryCtrl | 190 |
| ZW_DEBUG_CMD_INIT (Macro)..... | 214 |
| ZW_DEBUG_CMD_POLL (Macro) | 214 |
| ZW_DEBUG_INIT..... | 153 |
| ZW_DEBUG_SEND_BYTE..... | 153 |
| ZW_DEBUG_SEND_NUM | 153 |
| ZW_DebugInit..... | 214 |
| ZW_DebugPoll..... | 214 |
| ZW_DELETE_RETURN_ROUTE (Macro)..... | 165 |
| ZW_DELETE_SUC_RETURN_ROUTE (Macro) | 166 |
| ZW_DeleteReturnRoute | 165 |
| ZW_DeleteSUCReturnRoute..... | 166 |
| ZW_EEPROM_INIT (Macro) | 136 |
| ZW_EepromInit..... | 136 |
| ZW_ENABLE_SUC (Macro)..... | 192 |
| ZW_EnableSUC | 188, 192 |
| ZW_ExploreRequestInclusion | 76 |

| | |
|--|---------------|
| ZW_GET_CONTROLLER_CAPABILITIES (Macro) | 167 |
| ZW_GET_NEIGHBOR_COUNT (Macro) | 168 |
| ZW_GET_NODE_STATE (Macro) | 169 |
| ZW_GET_PROTOCOL_STATUS (Macro) | 77 |
| ZW_GET_RANDOM_WORD (Macro) | 77 |
| ZW_GET_ROUTING_INFO (Macro) | 170 |
| ZW_GET_SUC_NODE_ID (Macro) | 171, 206 |
| ZW_GET_VIRTUAL_NODES (Macro) | 193 |
| ZW_GetControllerCapabilities | 167 |
| ZW_GetNeighborCount | 168 |
| ZW_GetNodeProtocolInfo | 169 |
| ZW_GetProtocolStatus | 77 |
| ZW_GetRandomWord | 77 |
| ZW_GetRoutingInfo | 170 |
| ZW_GetRoutingMAX | 171 |
| ZW_GetSUCNodeID | 171, 206 |
| ZW_GetVirtualNodes | 193 |
| ZW_IS_FAILED_NODE_ID (Macro) | 172 |
| ZW_IS_NODE_WITHIN_DIRECT_RANGE (Macro) | 206 |
| ZW_IS_VIRTUAL_NODE (Macro) | 194 |
| ZW_isFailedNode | 172 |
| ZW_IsNodeWithinDirectRange | 206 |
| ZW_IsPrimaryCtrl | 172 |
| ZW_IsVirtualNode | 194 |
| ZW_MEM_FLUSH (Macro) | 136 |
| ZW_MEM_GET_BUFFER (Macro) | 134 |
| ZW_MEM_GET_BYTE (Macro) | 132 |
| ZW_MEM_PUT_BUFFER (Macro) | 135 |
| ZW_MEM_PUT_BYTE (Macro) | 133 |
| ZW_MEMORY_GET_ID (Macro) | 131 |
| ZW_MemoryFlush | 136 |
| ZW_NODE_MASK_BITS_IN (Macro) | 155 |
| ZW_NODE_MASK_CLEAR (Macro) | 155 |
| ZW_NODE_MASK_CLEAR_BIT (Macro) | 154 |
| ZW_NODE_MASK_NODE_IN (Macro) | 156 |
| ZW_NODE_MASK_SET_BIT (Macro) | 154 |
| ZW_NodeMaskBitsIn | 155 |
| ZW_NodeMaskClear | 155 |
| ZW_NodeMaskClearBit | 154 |
| ZW_NodeMaskNodeIn | 156 |
| ZW_NodeMaskSetBit | 154 |
| ZW_Poll | 79 |
| ZW_POLL (Macro) | 79 |
| ZW_PRIMARYCTRL (Macro) | 172 |
| ZW_PWM_CLEAR_INTERRUPT (Macro) | 130 |
| ZW_PWM_INT_ENABLE (Macro) | 130 |
| ZW_PWM_PRESCALE (Macro) | 129 |
| ZW_PWM_SETUP (Macro) | 128 |
| ZW_PWMClearInterrupt | 130 |
| ZW_PWMEnable | 130 |
| ZW_PWMPrescale | 129 |
| ZW_PWMSetup | 128 |
| ZW_Random | 79 |
| ZW_RANDOM (Macro) | 79 |
| ZW_REDISCOVERY_NEEDED (Macro) | 207 |
| ZW_RediscoveryNeeded | 162, 207, 278 |
| ZW_REMOVE_FAILED_NODE_ID (Macro) | 173 |

| | |
|---|----------------------------|
| ZW_REMOVE_NODE_FROM_NETWORK (Macro) | 177 |
| ZW_RemoveFailedNodeID | 173 |
| ZW_RemoveNodeFromNetwork | 177, 285 |
| ZW_REPLACE_FAILED_NODE (Macro) | 175 |
| ZW_ReplaceFailedNode | 175 |
| ZW_REPLICATION_COMMAND_COMPLETE (Macro) | 179 |
| ZW_REPLICATION_SEND_DATA (Macro) | 180 |
| ZW_ReplicationReceiveComplete | 179 |
| ZW_ReplicationSend | 180 |
| ZW_REQUEST_NETWORK_UPDATE (Macro) | 82 |
| ZW_REQUEST_NEW_ROUTE_DESTINATIONS (Macro) | 209 |
| ZW_REQUEST_NODE_INFO (Macro) | 181, 211 |
| ZW_REQUEST_NODE_NEIGHBOR_UPDATE (Macro) | 182 |
| ZW_RequestNetWorkUpdate | 71, 82, 162, 192, 277, 278 |
| ZW_RequestNewRouteDestinations | 209 |
| ZW_RequestNodeInfo | 181, 211, 282 |
| ZW_RequestNodeNeighborUpdate | 182 |
| ZW_RF_POWERLEVEL_GET (Macro) | 81 |
| ZW_RF_POWERLEVEL_REDISCOVERY_SET (Macro) | 84 |
| ZW_RF_POWERLEVEL_SET (Macro) | 80 |
| ZW_RF020x.h | 215 |
| ZW_RF030x.h | 215 |
| ZW_RFPowerLevelGet | 81 |
| ZW_RFPowerlevelRediscoverySet | 84 |
| ZW_RFPowerLevelSet | 80 |
| ZW_SEND_CONST (Macro) | 122 |
| ZW_SEND_DATA (Macro) | 98 |
| ZW_SEND_DATA_ABORT (Macro) | 122 |
| ZW_SEND_DATA_BRIDGE (Macro) | 108 |
| ZW_SEND_DATA_GENERIC (Macro) | 103 |
| ZW_SEND_DATA_META_BRIDGE (Macro) | 114 |
| ZW_SEND_DATA_META_GENERIC (Macro) | 112 |
| ZW_SEND_DATA_MULTI (Macro) | 118 |
| ZW_SEND_DATA_MULTI_BRIDGE (Macro) | 120 |
| ZW_SEND_NODE_INFO (Macro) | 85 |
| ZW_SEND_SLAVE_NODE_INFO (Macro) | 195 |
| ZW_SEND_SUC_ID (Macro) | 183 |
| ZW_SEND_TEST_FRAME (Macro) | 86 |
| ZW_SendConst | 122 |
| ZW_SendData | 98 |
| ZW_SendData_Bridge | 108 |
| ZW_SendData_Generic | 103 |
| ZW_SendDataAbort | 122 |
| ZW_SendDataMeta_Bridge | 114 |
| ZW_SendDataMeta_Generic | 112 |
| ZW_SendDataMulti | 118 |
| ZW_SendDataMulti_Bridge | 120 |
| ZW_SendNodeInformation | 85 |
| ZW_SendSlaveNodeInformation | 195 |
| ZW_SendSUCID | 183 |
| ZW_SendTestFrame | 86 |
| ZW_SET_DEFAULT (Macro) | 184, 202 |
| ZW_SET_EXT_INT_LEVEL (Macro) | 87 |
| ZW_SET_LEARN_MODE (Macro) | 185, 203 |
| ZW_SET_PROMISCUOUS_MODE (Macro) | 88 |
| ZW_SET_ROUTING_INFO (Macro) | 187 |
| ZW_SET_RX_MODE (Macro) | 89 |

| | |
|---|--------------------|
| ZW_SET_SLAVE_LEARN_MODE (Macro) | 196 |
| ZW_SET_SLEEP_MODE (Macro) | 89 |
| ZW_SET_SUC_NODE_ID (Macro) | 188 |
| ZW_SET_WUT_TIMEOUT (Macro) | 146 |
| ZW_SetDefault | 184, 202 |
| ZW_SetExtIntLevel | 87 |
| ZW_SetLearnMode | 185, 203, 280, 285 |
| ZW_SetPromiscuousMode (Not Bridge Controller library) | 88 |
| ZW_SetRFReceiveMode | 89 |
| ZW_SetRoutingInfo | 187 |
| ZW_SetRoutingMAX | 188 |
| ZW_SetSlaveLearnMode | 196 |
| ZW_SetSleepMode | 89 |
| ZW_SetSUCNodeID | 188, 192 |
| ZW_SetWutTimeout | 146 |
| ZW_STORE_HOME_ID (Macro) | 200 |
| ZW_STORE_NODE_INFO (Macro) | 201 |
| ZW_StoreHomeID | 200 |
| ZW_StoreNodeInfo | 201 |
| ZW_SUPPORT9600_ONLY(Macro) | 205 |
| ZW_Support9600Only | 205 |
| ZW_TIMER_CANCEL (Macro) | 127 |
| ZW_TIMER_RESTART (Macro) | 127 |
| ZW_TIMER_START (Macro) | 126 |
| ZW_TRIAC_DIM_SET_LEVEL (Macro) | 125 |
| ZW_TRIAC_INIT (Macro) | 123 |
| ZW_TRIAC_INIT_2_WIRE (Macro) | 123 |
| ZW_TRIAC_LIGHT_SET_LEVEL (Macro) | 125 |
| ZW_TRIAC_OFF (Macro) | 125 |
| ZW_TX_COUNTER (Macro) | 199 |
| ZW_Type_Library | 93 |
| ZW_TYPE_LIBRARY (Macro) | 93 |
| ZW_UART_CLEAR_RX (Macro) | 151 |
| ZW_UART_CLEAR_TX (Macro) | 151 |
| ZW_UART_DISABLE (Macro) | 151 |
| ZW_UART_ENABLE (Macro) | 150 |
| ZW_UART_INIT (Macro) | 147 |
| ZW_UART_READ_RX (Macro) | 152 |
| ZW_UART_REC_BYTE (Macro) | 148 |
| ZW_UART_REC_STATUS (Macro) | 147 |
| ZW_UART_SEND_BYTE (Macro) | 149 |
| ZW_UART_SEND_NL (Macro) | 150 |
| ZW_UART_SEND_NUM (Macro) | 149 |
| ZW_UART_SEND_STATUS (Macro) | 148 |
| ZW_UART_SEND_STRING (Macro) | 150 |
| ZW_UART_WRITE_TX (Macro) | 152 |
| ZW_Version | 94 |
| ZW_VERSION (Macro) | 94 |
| ZW_VERSION_BETA (Macro) | 95 |
| ZW_VERSION_MAJOR (Macro) | 95 |
| ZW_VERSION_MAJOR / ZW_VERSION_MINOR / ZW_VERSION_BETA | 95 |
| ZW_VERSION_MINOR (Macro) | 95 |
| ZW_WATCHDOG_DISABLE (Macro) | 96 |
| ZW_WATCHDOG_ENABLE (Macro) | 96 |
| ZW_WATCHDOG_KICK (Macro) | 97 |
| ZW_WatchdogDisable | 96 |
| ZW_WatchdogEnable | 96 |

| | |
|--|-----|
| ZW_WatchdogKick | 97 |
| ZW0102 Controller/Slave Unit | 271 |
| ZW0102 Development Controller Unit | 271 |
| ZW0201 Controller/Slave Unit | 272 |
| ZW0201 Development Controller Unit | 272 |
| ZW0301 Controller/Slave Unit | 272 |
| ZW0301 Development Controller Unit | 272 |
| Z-Wave Programmer | 270 |
| zwTransmitCount | 199 |