Instruction

Z-Wave Plus V2 Application Framework SDK7

Document No.:	INS14259
Version:	11
Description:	-
Written By:	JROSEVALL;JFR;PSH;KEWAHID;JSMILJANIC;COLSEN;NOBRIOT; ALMUNKHA
Date:	
Reviewed By:	NTJ;ABRUGGER;BBR;CRASMUSSEN;LTHOMSEN;COLSEN;SCBROWNI;CAOWENS
Restrictions:	Public

Approved by:

REVISION RECORD

Doc. Rev	Date	Author	Pages Affected	Brief Description of Changes
1	20181120	COLSEN JFR	ALL	Based on INS13953– Z-Wave Plus Application Framework v6.8x.0x Initial revision.
1	20181213	KEWAHID	5.4	Added a True Status description in the Architecture section.
1	20181213	ESOSTERG	8	Updated Utilities section.
1	20181214	JESMILJA	7	Updated Command Classes description
2	20181217	JFR	5.1 8.4 9	Added application memory constraints Updated peripheral drivers Added firmware update images and bootloader
3	20190116	MLEDESMA	ALL	Grammar and structure (consistent format) modification
4	20190313	JOROSEVA	5.3	Updated description of Power Management
4	20190315	JOROSEVA	6.3	Added description of NVM3 File System
4	20190315	JESMILJA	5.1 6.2	Updated description of how to create application Updated instruction of usage of config file Removed 5.2.6 and 5.2.7 as obsolete
4	20190320	JOROSEVA	6.6	Updated description of ApplicationInit() and ApplicationTask()
4	20190320	JFR	9.2 8.4.3 6.3 All	Updated flashing of boot loader and app. Added ADC driver. Added how to adjust Tx power. Minor typos.
5	20190325	JFR	6.5	Added watchdog.
6	20190704	CHOLSEN	6.4	Minor text changes and fix of example code.
7	20190712	SCBROWNI	All	Minor typos
8	20190812	PESHORTY	5.2	Added description of ZAF_SetMaxInclusionRequestIntervals
8	20190829	PESHORTY	6.5	Added note about watchdog always being enabled in production code
8	20190829	CHOLSEN	8.6	Added section “Callback before entering sleep”
8	20190903	JFR	8.4.1.1	GPIO port usage in serial API application.
8	20190903	JOROSEVA	5.2.1	Learn mode status events.
8	20190903	SCBROWNI	5.2	Editorial review of Section 5.2
9	20191022	CHOLSEN JFR	8.4.2 4	Minor correction in source code. Added “Introduction to the Z-Wave Technology”.
10	20200518	JFR	9	Added location of OTA key.
11	20200611	SCBROWNI	All	Tech Pubs review of new sections since version 7

Table of Contents

Table of Figures

1 Definitions, acronyms and abbreviations

Abbreviation	Explanation
AGI	Association Group Information
CC	Command Class
NIF	Node Information Frame
OTA	Over The Air
S0	Security 0 Command Class
S2	Security 2 Command Class
SDK	Software Development Kit
ZAF	The Z-Wave Plus Application Framework
AOS	Always on slave
RSS	Reporting Sleeping Slave
LSS or FLiRS	Listening Sleeping Slave

2 Introduction

This document describes the Z-Wave Plus V2 Application Framework (ZAF) version 10.1x.x distributed on Z-Wave 700 SDK 7.1x.x.

3 Purpose

The purpose of the ZAF is to facilitate the implementation of robust Z-Wave Plus V2 compliant products.

4 Introduction to the Z-Wave Technology

Z-Wave is a wireless mesh protocol oriented to the residential control and automation market but also suitable for light commercial applications. The technology provides a simple yet reliable method to wirelessly control lights and A/V equipment in your house. Z-Wave works in the unlicensed industrial, scientific, and medical (ISM) bands around 900 MHz. Regional frequencies vary slightly. Each Z-Wave network may comprise up to 232 nodes. Nodes may retransmit a message to guarantee delivery. The typical communication range between two nodes is 100 feet.

The Z-Wave ecosystem offers a routing protocol stack and a complete Z-Wave Plus Application Framework of device types and command classes for interoperable deployments. Interoperability is ensured between all device types thanks to the Z-Wave certification program. The Z-Wave logo is only granted to products passing certification.

4.1 Protocol Stack Overview

Z-Wave offers a routing protocol that reliably transfers messages up to five hops away; i.e., up to 500 feet. The protocol stack comprises a PHY/MAC layer to control access to RF media, a transport layer to handle frame integrity and retransmissions, and a network layer with all its routing magic and application interfaces.

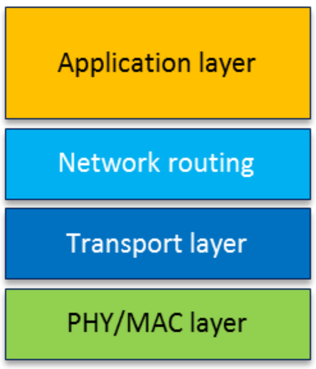


Figure 1. Z-Wave Protocol Stack

The maximum size of payload data is 46 bytes when routing is used. The Z-Wave protocol uses standard collision-avoidance methods, postponing a transmission a random number of milliseconds when media is busy. The Z-Wave transport layer controls the transfer of data between two nodes including acknowledgement and optional retransmission.

Multicast and broadcast may only be used in direct range. Broadcast and multicast may be used to reach more than one destination address. In case of multicast, the same payload will be delivered to selected nodes only.

The Z-Wave application layer is responsible for handling application commands. Commands are divided into two classes: Z Wave protocol and application-specific. Most protocol-related operations are just address assignment logic, but commands that are more complex are defined for advanced network management operations.

Each Z-Wave network has a unique 32-bit identifier called Home ID. Every new node joining the network inherits the same Home ID from the primary controller. Individual nodes in the network are addressed using an 8-bit Node ID that is unique within the network.

4.2 Classic Z-Wave

The following text makes references to classic nodes. In short, the term "Classic Z-Wave node" covers previous generations of Z-Wave nodes that do not implement recently introduced features, such as Network-Wide Inclusion (NWI), Dynamic Route Resolution, and FLiRS communication.

4.3 Node Types

There are two main types of devices: controllers and slaves. Controllers can handle network management and communication to classic nodes. Slaves provide no network management capability.

4.4 Controllers

A controller node maintains a routing table for all operational links in the network. This table allows the controller to calculate routes between any two nodes in the network. The primary controller may refresh the routing table and distribute updated routing tables to other controllers.

Controllers come in three variants: portable, static, and bridge. However, SDK 7.11.x contains only the bridge controller because the static controller is discontinued.

The portable controller is optimized for battery operation. It is typically used for remote control devices.

The bridge controller is intended for mains-powered control panels, gateways, or network managers. The bridge controller may also act as a repeater for other nodes.

4.5 Slaves

A slave device has simpler functionality than a controller. It may repeat a message for other nodes.

Reporting Sleeping Slave (RSS) Role Type node is intended for battery-powered devices that only wake up and communicates when a local event has occurred. An RSS device may be used for sensor-style devices, such as alarms and sensors.

Listening Sleeping Slave (LSS) Role Type node is intended for battery-powered devices that can be reached even though they are sleeping thanks to a wakeup beam (FLiRS device). Referred to as duty cycling in the literature, the Frequently Listening Routing Slave (FLiRS) wakes up in fixed intervals to listen very briefly for a preamble pattern. This enables the design of products with battery lifetimes measured in years. Yet, it is possible to reach such devices on short notice.

RSS and LSS nodes cannot operate as repeaters as they are sleeping most of the time to conserve battery.

4.6 Network Operation

Management of Z-Wave nodes constitutes two main operations: inclusion/exclusion and association. Inclusion adds a new node to the network. Exclusion removes a node. Only primary controllers can include and exclude nodes.

Association is the creation of a logical connection between applications. In other words, it defines what controls what. Association is handled by the application layer.

4.7 Routing Principles

Z-Wave uses source routing to reach a destination. Source routing allows implementation of a lightweight protocol, avoiding distributed routing tables in all repeaters. This puts a limit to the length of routes. Real-world deployments indicate that residential networks rarely have more than 2-hop routes. Z-Wave's support for 5-hop routes is a sufficient and efficient compromise.

The route is carried in the routing header and every repeater forwards the frame according to the routing header. Only always-listening nodes can participate in routing, but routing may also be used to reach FLiRS nodes.

Network Wide Inclusion (NWI) allows a user to include a new node even though the new node is not within range of the primary controller. Dynamic route resolution allows a node to repair broken routes during normal operation. Classic nodes do not support NWI and dynamic route resolution.

Network Wide Exclusion (NWE) uses the same explorer strategy as Network Wide Inclusion (NWI) to accomplish an out-of-range exclusion of nodes from the network. It is also possible to remove a specific node from the network by specifying the Node ID.

4.8 Application Development

Depending on node type functionality (such as controller vs. slave), developers may choose from a selection of libraries. On top of the chosen library, an application designer may choose from a wide range of Command Classes; including light control, sensors, garage port control, and many others. Command Classes are a collection of functionally related commands. A device may implement several functions and therefore support more Command Classes.

Z-Wave applications are designed as a state machine periodically polled from the Z-Wave library. This allows for the design of products with fewer CPU resources than typically required for OSs with threads, tasks, priorities, etc. This again translates into inexpensive products suited for mass production.

Depending on the actual product, an application may interface to the Z-Wave protocol stack in three ways:

- Most constrained devices, like a light dimmer with one button, may have their applications running in the on-chip MCU. In this configuration, the Z-Wave API is used directly via function calls provided by the binary image implementing the Z-Wave library.

- Larger devices, like a remote control with display, may have their own host processor. The application designer may prefer to implement all application logic in the host processor; only running the Z-Wave protocol stack in the on-chip MCU. The Z-Wave Serial API provides an abstracted version of the Z-Wave API that is accessed via an on-chip serial port. The application design principle for the Z-Wave part should still be a state machine that reacts to incoming events, callback functions, and timeouts.
- Most advanced devices like IP gateways and PC-based light control servers may use an even more abstracted API provided via the Network Management Command Class. In this model, all communication is carried in IP packets. The Z/IP Gateway library provides this mapping.

4.9 Managing Interoperability

Interoperability is a key part of the Z-Wave ecosystem. Every product must pass certification to be granted the Z-Wave logo. The Z-Wave Alliance manages the Z-Wave certification program, but certification testing is performed by independent test houses. Certification ensures that a product correctly implements all device and command classes that it claims to support.

5 Architecture

Figure 1 shows the architecture of the ZAF and its relation to the hardware and the Z-Wave protocol.

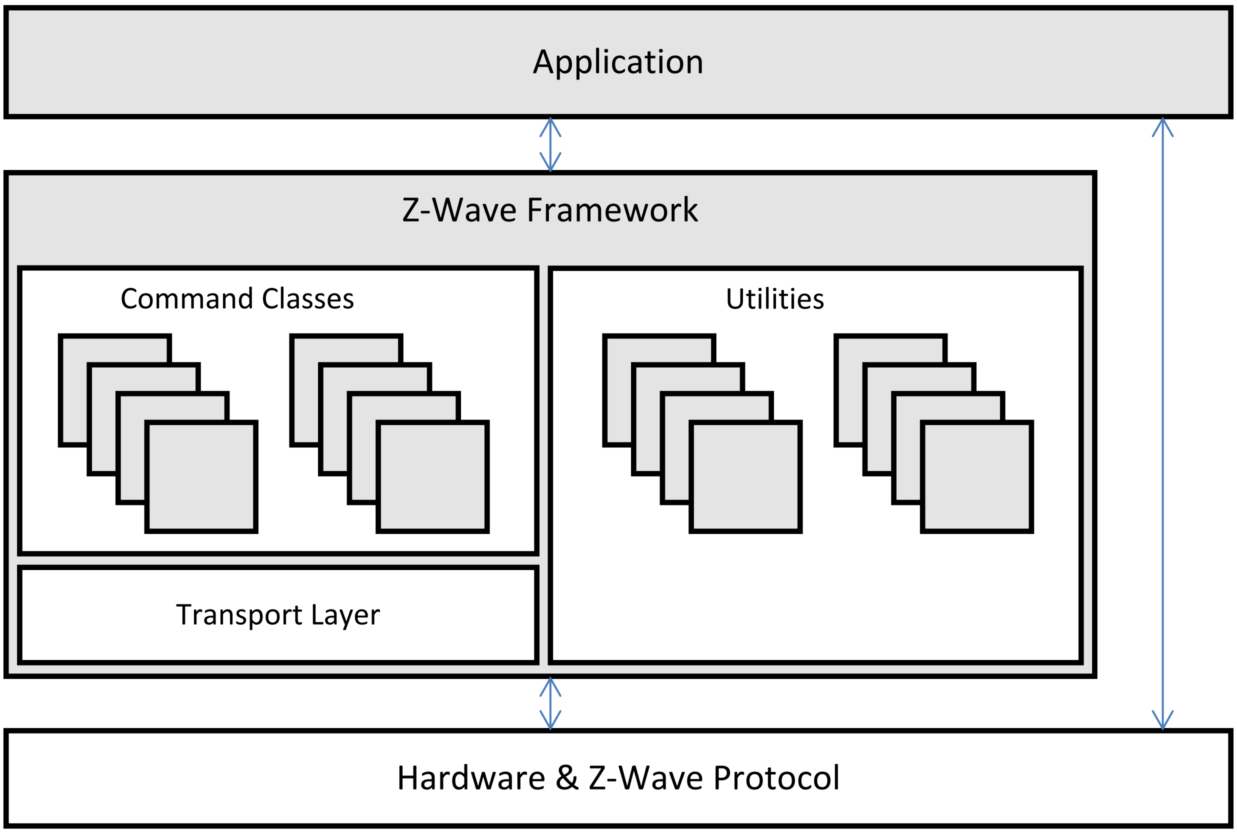


Figure 1. The Z-Wave Plus Application Framework Architecture

The ZAF consists of three blocks:

- **Transport Layer:**
  This layer handles all communication with the protocol, which includes single cast, multicast, Multi-Channel encapsulation, delivery of bundled commands, etc.
- **Command Classes:**
  These modules parse and compose Command Class frames.
- **Utilities:**
  Utilities are composed of different modules including those that are used for handling I/O communication specific for the WSTK and BRD 8029 boards bundled with the SDK. Other modules are battery monitoring and firmware updating, etc.

The framework implements an event-driven application design.

The framework provides built-in features for developing simpler applications. Transmit buffers are mutex-protected to ensure that the application has only one transmit request job (unsolicited event) at a time. The transmit buffer is released only when the transmit request job is completed or has timed out. The Framework can handle one request job and one response job at the same time.

The ZAF is split into the following two folders:

- `/CommandClasses/` contains CC modules. All CC modules share a protected transmit buffer provided by the ZW_tx_mutex module. The ZW_tx_mutex module implements two transmit buffers, one for request calls and one for response calls.
- `/ApplicationUtilities/` contains utility modules and interfaces to the transport layer. Some of the modules are used for simple MMI-setup such as button and LED handling. Other modules like association_plus, battery_monitor, battery_plus, and ota_util are more complex utility modules which interface to CC and the client application.

5.1 Application Memory Constraints

The following memory resources are available for certified application development including Z-Wave Framework and Utilities:

- 64 kB of Flash memory for executable code
- 8 kB of RAM for temporary data

The above limitations MUST NOT be exceeded. Violating the limitations may impair compatibility with future SDK versions.

The code space usage for the distributed certified applications are given as example for the beta release:

Application SDK 7.00.00 Beta	Binary Total	Application	Framework (ZAF)	Utilities (Components)
DoorLockKeyPad	180256 bytes	4139 bytes	24197 bytes	2326 bytes
Powerstrip	186064 bytes	3832 bytes	28259 bytes	2326 bytes
SensorPIR	181884 bytes	2546 bytes	26559 bytes	2326 bytes
SwitchOnOff	177684 bytes	2211 bytes	22667 bytes	2326 bytes
WallController	179788 bytes	3355 bytes	22907 bytes	2326 bytes

The certified DoorLockKeyPad application need 4139 bytes + 24197 bytes + 2326 bytes = 30662 bytes out of the 64 kB allocated for the Application, Framework and Utilities. In addition, a separate OTA buffer is allocated for firmware update.

The bootloader resides in separate 10 kB storage area and is not part of the certified application.

Note: The numbers listed under sub-modules (the last three columns) are the raw object file code sizes. The linker will remove unused functions/code and result in a slightly smaller actual code space usage.

The application code space usage can be found in the .map file in Simplicity Studio:

https://www.silabs.com/community/mcu/32-bit/knowledge-base.entry.html/2016/03/07/interpreting_thegcc-BpRD

5.2 Smart Start

The Smart Start feature is part of the protocol and automatically handles the inclusion process without having a user physically interact with a device. When powered on for the first time, the device tells the world that it is ready for inclusion and most likely a controller nearby will hear this and include the device. If inclusion process times out, it retries again after a given time.

5.2.1 Starting smart start inclusion

The Smart Start inclusion process starts when the application invokes "ZAF_setNetworkLearnMode()" with the parameter "E_NETWORK_LEARN_MODE_INCLUSION_SMARTSTART".

The Z-Wave protocol informs the application about the status of inclusion using the command status handler with the following events:

- EZWAVECOMMANDSTATUS_NETWORK_LEARN_MODE_START
- EZWAVECOMMANDSTATUS_LEARN_MODE_STATUS

The EZWAVECOMMANDSTATUS_NETWORK_LEARN_MODE_START event contains a status telling whether the inclusion was started successfully.

The EZWAVECOMMANDSTATUS_LEARN_MODE_STATUS event can contain several different statuses:

- ELEARNSTATUS_SMART_START_IN_PROGRESS
- ELEARNSTATUS_LEARN_IN_PROGRESS
- ELEARNSTATUS_ LEARN_MODE_COMPLETED_FAILED
- ELEARNSTATUS_ LEARN_MODE_COMPLETED_TIMEOUT
- ELEARNSTATUS_ ASSIGN_COMPLETE

The status ELEARNSTATUS_SMART_START_IN_PROGRESS indicates that the process of smart start inclusion to a controller has commenced.

The status ELEARNSTATUS_LEARN_IN_PROGRESS indicates that the process of classic inclusion to a controller has started.

If the status is ELEARNSTATUS_ LEARN_MODE_COMPLETED_FAILED, the application must reset its NVM and reboot.

If the status is ELEARNSTATUS_ LEARN_MODE_COMPLETED_TIMEOUT, the application must re-enter the Smart Start inclusion process.

5.2.2 Configuring smart start inclusion

When an end device is in smart start it will send out several inclusion request with increasing delay between. By default, the maximum inclusion request interval is set to 512 seconds.

The maximum inclusion interval can be changed from the application. A higher value might be used in battery powered products to save battery power if it is expected that the product will be in smart start inclusion for a long time. A lower value might be used to ensure a faster smart start inclusion when power consumption is not an issue.

The command for changing the maximum inclusion interval is:

Function:

- ***void ZAF_SetMaxInclusionRequestIntervals(uint32_t intervals)***
  Function for setting the maximum inclusion request delay.

Parameter:

- ***uint32_t intervals***
  Maximum inclusion request delay in 128 seconds steps. Valid range is 5-99

The inclusion requests will be sent out on interval shown in the table below:

Request number	*Delay since last inclusion request*
0 (Power up)	0 sec
1	Random(0..16 sec)
2	Random(16..32 sec)
3	Random(32..64 sec)
4	Random(64..128 sec)
5	Random(128..256 sec)
6	Random(256..512 sec)
..	..
X	Random(Interval*64..interval*128 sec)

5.3 Power Management

Power management is handled by the Z-Wave protocol. All listening nodes are prevented by the protocol from entering any energy mode other than EM0 and EM1. Frequently listening (FLiRS) nodes can enter energy modes EM0-EM2, while non-listening nodes can enter the EM0-EM2 modes and the EM4-hibernate mode.

A new module is introduced in the ZAF/ApplicationUtilities called PowerManagement. This module is used to communicate to the protocol if a FLiRS or non-listening application wants to stay awake for a given time. The module makes it possible to register so-called Power Locks that when enabled prevent the processor from going into a low power mode. A power lock can be set to time out after a selectable number of milli seconds or can be enabled indefinitely until cancelled by a new function call. (Indefinite power locks are enabled by setting the timeout value to zero.)

Power Locks are registered in one of two types of configurations described in the table below:

Power Lock configuration	*Description*
PM_TYPE_RADIO	Prevents FLiRS and non-listening nodes from entering other energy modes than EM0 and EM1. This means that the radio transceiver will stay operational.
PM_TYPE_PERIPHERAL	Prevents non-listening nodes from going to deeper sleep than EM2. The radio transceiver is not operational in EM2 but the Low Energy Peripherals are.

For further details, refer to section 8.6.

5.4 True Status Module

The True Status module implements the requirements for **Lifeline Reports** as specified in document “Z-Wave Plus v2 Device Type Specification” [15].

The main components of the True Status module are described in the following sub-sections.

5.4.1 True Status Engine (TSE)

The True Status Engine component is located in the ZAF_ApplicationUtilities_TrueStatusEngine folder and consists of the following source files:

- ZAF_TSE.c
- ZAF_TSE.h

TSE implements the functionality for registering and handling the state change events that a node wants to report to its lifeline association group members.

A state change can be triggered by a command from a remote note, or by a local change (e.g., a button press).

Public functions:

- ***bool ZAF_TSE_Init();***

  Function for initializing the True Status Engine. Called by the ZAF during initialization.

- ***bool ZAF_TSE_Trigger(void* pCallback, void* pData, bool overwrite_previous_trigger);***

  Function for registering state change events and triggering the report to be sent to lifeline group members after a predefined delay. The delay is currently defined to 250 ms. The delay is a means to prevent network collisions and to avoid generation of redundant reports from rapid state changes.

  The function also takes care of sending the report to the relevant lifeline group members only; i.e., it will **not** send the report to a lifeline destination that issued the command causing the state change. The current implementation can hold up to 3 different state change requests in a queue awaiting the predefined delay to expire. Additional requests during this period will be discarded.

Arguments:

- **pCallback:** Pointer to a callback function for sending the state change report to the lifeline group members. (Described in more details in the next section).
- **pData:** Pointer to a data struct that will be passed in argument to the pCallback function. The pData pointed struct MUST first contain a RECEIVE_OPTIONS_TYPE_EX variable indicating properties about the received frame that triggered the change. Local changes must also include a RECEIVE_OPTIONS_TYPE_EX in the pData struct.
- **overwrite_previous_trigger:** Boolean parameter indicating if a previous trigger with the same pCallback and the same source endpoint in the pData struct should be discarded or not. Set it to true to overwrite previous triggers and false to stack up all the trigger messages.

Returns true if success. False if the request could not be handled (queue is full).

5.4.2 True Status Callback Functions

The True Status Callback Functions consist of several functions that implement the functionality for sending the actual state change reports to the lifeline association group members. These functions are implemented in each of the relevant command class modules.

The True Status Callback Function is one of the arguments passed to the ***ZAF_TSE_Trigger()*** function (described in the previous section), and it will be executed by the True Status Engine for sending the state change reports to each of the relevant members of the lifeline group, one at a time.

An example of a True Status Callback Function implementation can be found in the Command Class BinarySwitch:

void CC_BinarySwitch_report_stx(TRANSMIT_OPTIONS_TYPE_SINGLE_EX txOptions, s_CC_binarySwitch_data_t* pData);

Arguments:

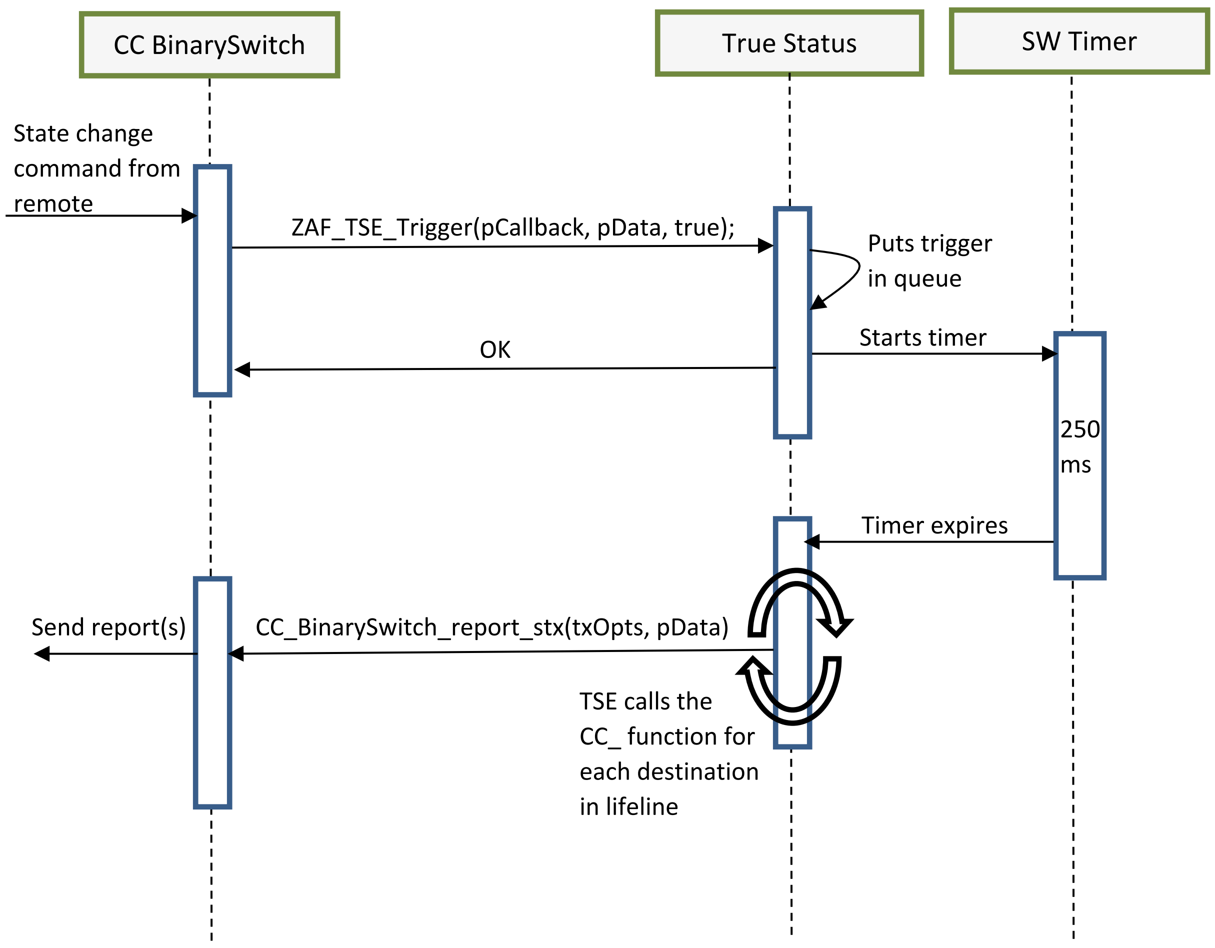
- **txOptions:** Tx Options passed from the True Status Engine with the required destination parameters for sending the state change report to a given lifeline member.
- **pData:** Pointer to the data struct previously registered at the call to ***ZAF_TSE_Trigger()***. Contains the command data for the report to be sent.

It is important to note that a True Status Callback Function must only send the state change reports by single-cast addressing, not by multicast. Therefore, be sure to use only the ***Transport_SendRequestEP()*** transmit function for this purpose as it is also done in the ***CC_BinarySwitch_report_stx()*** implementation.

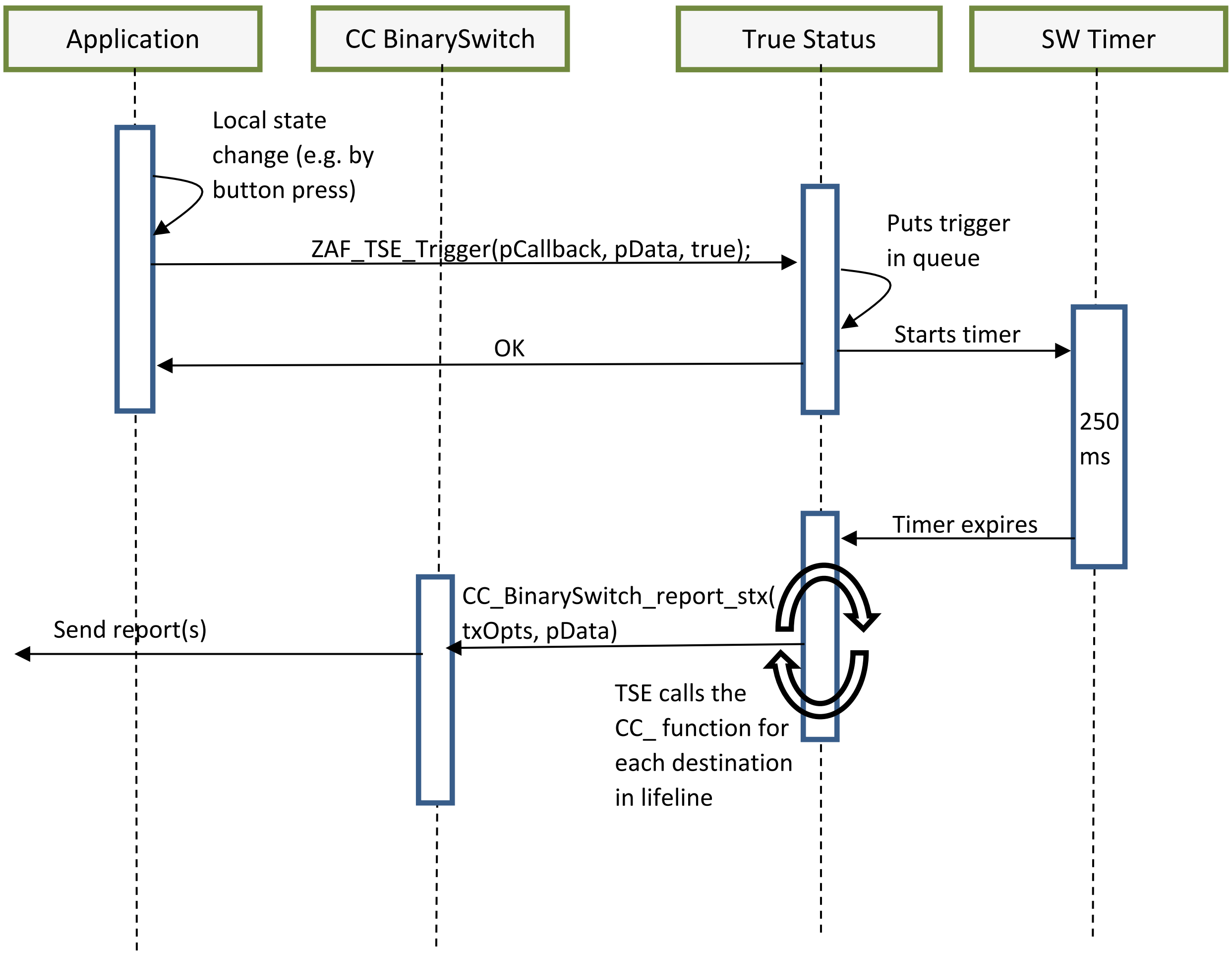
5.4.3 True Status Sequence Flows

The following diagrams show the function call flows for a BinarySwitch example for the two use cases: 1) state change triggered by a command from a remote note, and 2) state change triggered by a local change (e.g., a button press).

5.4.3.1 Use Case 1 – State Change Triggered by a Command from a Remote Note


CC BinarySwitch
True Status
SW Timer
State change command from remote
ZAF_TSE_Trigger(pCallback, pData, true);
Puts trigger in queue
OK
Starts timer
250 ms
Timer expires
Send report(s)
CC_BinarySwitch_report_stx(txOpts, pData)
TSE calls the CC_ function for each destination in lifeline

5.4.3.2 Use Case 2 – State Change Triggered by a Local Change


Application
CC BinarySwitch
True Status
SW Timer
Local state change (e.g. by button press)
ZAF_TSE_Trigger(pCallback, pData, true);
Puts trigger in queue
OK
Starts timer
250 ms
Timer expires
CC_BinarySwitch_report_stx(txOpts, pData)
Send report(s)
TSE calls the CC_ function for each destination in lifeline

6 How to develop A Z-Wave plus application

The Z-Wave Plus application's basic functionality is defined by the device type and role type, which are explained further in [15] and [2]. It is important to determine the right combination of device type and role type for your Z-Wave Plus application.

Once the device and role types are determined, the application development can start.

6.1 Create Application Folder and Set Up Build Environment

6.1.1 Select the application to start with

Pick one of existing software samples and modify it to match your needs. Refer to chapter 5 in [21] for more details. Selection should be made mainly based on Role Type: AOS, RSS or LSS:

- AOS: SwitchOnOff, WallController, PowerStrip
- RSS: SensorPIR
- LSS/FLiRS: DoorLock

In addition, there are some additional options to consider:

- Endpoint implementation is supported by Power Strip.
- Association groups are implemented in Wall Controller.

6.1.2 Create new Simplicity Studio project

Follow these steps to set up a working directory for your application:

1. Get a general understanding of the build environment, refer to [3] for more information.
2. Select one of the applications and create an example project in Simplicity studio.
3. Rename imported application to a suitable name for your application by renaming the root folder, i.e. to *<MyApp>*.
4. Navigate in to the *<MyApp>/src* folder and rename *.c* file to *<MyApp>.c*.
5. As for any other example, search for APP_FREQ and replace it with the frequency you are using. For more details, refer to chapter 6 in [20].
6. Build the application to verify that the changes did not break anything. It is also possible to download the application to the chip to check that the application runs without error.

6.2 Setting Up config_app.h

6.2.1 Generic Type, Specific Type, and Device Options

A Z-Wave Plus device must specify a Generic Device Class and a Specific Device Class by using the following two definitions:

- GENERIC_TYPE
- SPECIFIC_TYPE

Refer to [8] and [15] to select the appropriate Generic and Specific Device Classes for your product. The identifiers values are defined in *ZW_classcmd.h*.

DEVICE_OPTIONS_MASK is a bitmask used to specify a set of device options. See the *ZW_basis_api.h* file for details.

6.2.2 Role Type, Node Type, Icon Type, and User Icon Type (Z-Wave Plus Info CC)

The Role Type, node type, icon type, and user icon type are all values used by the Z-Wave Plus Command Class. They are set using the following definitions:

- APP_ROLE_TYPE
- APP_NODE_TYPE
- APP_ICON_TYPE
- APP_USER_ICON_TYPE

For further information about the Icon Types used in the Z-Wave Plus Info CC, refer to [4]. Contact the Z-Wave Alliance to acquire new Icon Types.

6.2.3 Manufacturer Specific CC / Firmware Update

The Manufacturer specific CC requires some product specific values to be defined. Refer to the Manufacturer Specific Command Class in [12] for more information. The ZAF uses the following macros for these values:

- APP_MANUFACTURER_ID
- APP_PRODUCT_TYPE_ID
- APP_PRODUCT_ID

The Firmware Update Command Class uses this definition:

- APP_FIRMWARE_ID: The firmware ID consists of the product type ID and the product ID.

The manufacturer and product IDs are used to identify a Z-Wave product from a manufacturer point of view. Other applications can also use this information to show a company logo, type of product, user guide, etc. on the user interface. This information is exchanged via the Manufacturer Specific CC and currently allocated manufacturer IDs are listed in the *ZW_classcmd.h* file. See [12] for more information about these two Command Classes.

6.2.4 Association Group Information (AGI)

Refer to [12] and [6] for a general understanding of AGI and Associations.

In an application, the AGI must be configured in config_app.h.

For Z-Wave Plus products, the Lifeline group is mandatory. It can be defined using the following declaration:

```
#define AGITABLE_LIFELINE_GROUP \
 {<command class X>, <command A>}, \
 {<command class Y>, <command B>}, \
 {<command class Z>, <command C>}
```

Refer to [15] to find out what events and Command Classes to use in the Lifeline group. Definitions for Command Classes and Commands are found in *ZW_classcmd.h*.

Other groups (apart from Lifeline) must be defined using the following format if the application is not implementing Multi Channel endpoints:

```
#define AGITABLE_ROOTDEVICE_GROUPS \
 {<profile 1 MSB>, <profile 1 LSB>, {<command class X, command A}, "<group 2 name>"}\
 {<profile 2 MSB>, <profile 2 LSB>, {<command class X, command A}, "<group 3 name>"}\
```

If the application is implementing Multi Channel endpoints, the AGI definition must be filled out using the endpoint groups. An example of Multi Channel endpoint implementation is Power Strip application.

```
#define AGITABLE_ROOTDEVICE_GROUPS \
 AGITABLE_ENDPOINT_1_GROUPS, \
 AGITABLE_ENDPOINT_2_GROUPS, \
 AGITABLE_ENDPOINT_3_GROUPS, \
 AGITABLE_ENDPOINT_4_GROUPS
```

AGITABLE_ENDPOINT_X_GROUPS represents the non-lifeline groups for endpoint X. Each of the endpoint groups are defined as the Root Device groups defined by *AGITABLE_ROOTDEVICE_GROUPS*, but with a matching definition name, other profiles, Command Classes, etc.

```
#define AGITABLE_ENDPOINT_1_GROUPS \
 {<profile X MSB>, <profile X LSB>, {<command class>, <command>}, "group name"}, \
 {ASSOCIATION_GROUP_INFO_REPORT_PROFILE_CONTROL,
ASSOCIATION_GROUP_INFO_REPORT_PROFILE_CONTROL_KEY01, {COMMAND_CLASS_SWITCH_MULTILEVEL_V4,
SWITCH_MULTILEVEL_SET_V4}, "Button 1"}
```

When the product implements Multi Channel endpoints, it must be mapped to the root association groups. Root groups mapping provides backwards compatibility because devices that do not support Multi Channel must be able to ask the Root Device which groups are supported. Root group mapping defines groups in the Root Device mapping to endpoints' groups. An example is given in Table 1.

Table 1. Root Device and Endpoint Group Mapping

Root Device Group	Endpoint ID	Endpoint Ggroup ID
Root Device Group 2	Endpoint 1	Endpoint 1 Group 2
Root Device Group 3	Endpoint 2	Endpoint 2 Group 2

The following shows how root group mapping is configured in *config_app.h*. The details of the root group mapping setup depend on the application.

```
#define ASSOCIATION_ROOT_GROUP_MAPPING_CONFIG \
  {<ROOT DEVICE GROUP 2>, <ENDPOINT ID 1>, <ENDPOINT GROUP 2>}, \
  {<ROOT DEVICE GROUP 3>, <ENDPOINT ID 2>, <ENDPOINT GROUP 2>}
```

Refer to the *config_app.h* file in the Wall Controller application for a group mapping example. The SensorPIR also shows how AGI can be set up.

Group definitions are used to initialize the AGI in the main source file of the application.

6.2.5 Security

Security level is configured in the *config_app.h* file with the definition of REQUESTED_SECURITY_KEYS flag. Definitions can be found in the *ZW_security_api.h* file; see [5] for more information about security.

Application security level bits must be specified when developing a product. Security level "No-secure" is the default value.

```
#define REQUESTED_SECURITY_KEYS \ (SECURITY_KEY_S2_UNAUTHENTICATED_BIT|SECURITY_KEY_S0_BIT)
```

Added functionality:
ApplicationSecureKeysRequested() includes security keys flag REQUESTED_SECURITY_KEYS.

6.3 Setting Up config_rf.h

The Tx power should be set by modifying the file config_rf.h to comply with local regulatory authorities. This file contains 2 defines:

```
#define APP_MAX_TX_POWER              0
#define APP_MEASURED_0DBM_TX_POWER   33
```

Where, APP_MEASURED_0DBM_TX_POWER is for tuning of TX power to 0 dBm in a conducted environment. During the turning process, 'APP_MAX_TX_POWER' should be 0 (0dBm).

APP_MAX_TX_POWER is a value from -100 to +100 deci dBm (-10dBm to +10dBm) and is for adjusting Tx power by the application framework after completion of Tx power tuning.

Valid range is:

```
100 >= (APP_MAX_TX_POWER + APP_MEASURED_0DBM_TX_POWER) >= -100
```

To set maximum power, use:

```
#define APP_MAX_TX_POWER            100
#define APP_MEASURED_0DBM_TX_POWER    0
```

Procedure for setting TX power

a) Set APP_MAX_TX_POWER to 0 and set APP_MEASURED_0DBM_TX_POWER to 0 and compile the application
b) Do a conducted measurement of the Tx power with these settings on the product. It is important that this is done on the finished product.
c) Set the APP_MEASURED_0DBM_TX_POWER to the measured, conducted value.

d) Set the APP_MAX_TX_POWER to the desired/allowed Tx power in the region where the product is to be used.
e) Compile the firmware
f) Perform the regulatory radiated measurements with the regulatory limit value (step d), e.g. 50 deci dBm (5 dBm). If the regulatory measurements allow for a e.g. ½ dB more, then change the deci dBm to 55 and re-compile. If the radiation is too high in one direction and e.g. ½ dB less is needed, then change to 45 deci dBm and recompile.

6.4 Setting Up Files in Non-Volatile Memory

All applications must open a NVM3 [18] file system that is used by the ZAF for storing files in non-volatile memory. The file system can also be used by applications to store application specific files. The file system is opened by calling the function ApplicationFileSystemInit(..) found in ZAF_nvm3_app.c.

The application file system occupies an area of 12k bytes in flash. For the file system to function properly in all circumstances it is recommended that one not store more than 4k bytes of data in it, including both application-specific files and files used by ZAF. Application developers must not increase the default size of the application file system (NVM3_APP_NVM_SIZE) since it will make it collide with the area used for storing Z-Wave protocol files.

The NVM3 file system uses a cache in RAM for storing the locations of files in flash. By default, the maximum number of files in the cache is set to 10. If the total number of files used by both ZAF and application exceeds 10, NVM3_APP_CACHE_SIZE in ZAF_nvm3_app.c must be increased accordingly, or the file system will slow down considerably.

Each file in an NVM3 file system must be assigned an individual 20-bit object key as file identifier [18]. The range of integers 0x51000—0x5FFFF have been reserved to be used as file identifiers for the ZAF and must NOT be used for application specific files. Application specific files should preferably use identifiers in the range 0x00000—0x0FFFF.

Each sample application lists the files it uses in an array called g_aFileDescriptors[]. The list contains the file identifiers and sizes of the files used by ZAF in the application. Some sample applications also have application-specific files included in the list. On upstart, the application checks if all files in the list are present in the file system. All application file systems must have a file with file identifier ZAF_FILE_ID_APP_VERSION that contains the version number of the application. If this file is missing, the file system is considered unwritten or corrupt leading to reformatting of the file system and all files set to default.

To introduce an application-specific file to the file system just define a new 20-bit object key number that is not already in use. The file will be added to the file system once it is first written using the function nvm3_writeData() [19].

```
#define FILE_ID_MYNEWFILE (0x00112)

typedef struct SMyAppData
{
  int32_t status;
  uint8_t vector[10];
} SMyAppData;
```

```
SMyAppData wrtData;
wrtData.status = -5;
memset(&(wrtData.vector), 0x55, sizeof(wrtData.vector));

//This call will create and write a file containing wrtData in the file system.
nvm3_writeData(pFileSystemApplication, FILE_ID_MYNEWFILE, &wrtData, sizeof(wrtData));

SMyAppData rdData;

//This call will read the content of the file to rdData. Please be warned that it is
//not possible to read part of the file. If sizeof(rdData) < sizeof(wrtData) above,
//the read function will return an error.
nvm3_readData(pFileSystemApplication, FILE_ID_MYNEWFILE, &rdData, sizeof(rdData));
```

It is recommended that one write default content to application-specific files in the function SetDefaultConfiguration() that gets called after formatting the file system.

When performing firmware update, the content of the application file system remains intact. If the file identifiers remain the same, old files used by the previous firmware can still be used. When developing a newer version of the firmware where new files are added or files are changed in any way, it is necessary to increase the APP_VERSION, APP_REVISION or APP_PATCH number in the applications config_app.h. The bootloader does not accept files having lower or the same version number. Then, on first upstart of the new firmware, the application will be able to recognize that the file system on the chip belongs to an older version of the firmware by reading the file with identifier ZAF_FILE_ID_APP_VERSION.

6.5 Watchdog Enable/Disable

The watchdog timer is enabled in the application by default and will reset the device if the application task runs for more than 1 second without releasing the processor.

To disable the watchdog during development, comment out the WDOGn_Enable() function call inside the ApplicationTask function as shown below:

NOTE: The watchdog should always be enabled in production code but can be disabled in debug build

```
static void
ApplicationTask(SApplicationHandles* pAppHandles)
{
  // Init
  DPRINT("Enabling watchdog\n");
  WDOGn_Enable(DEFAULT_WDOG, true);   <<-- Comment out this line to disable the watchdog
```

6.6 Source File

The application must implement one function for initialization: ApplicationInit(..). The function is called by the Z Wave main function during system startup and gets the wakeup reason as input argument, see the *ZW_basis_api.h* file. Application specific hardware initializations like pin configuration should be done from this function. After configuration is finished the function must register an application task by calling ZW_ApplicationRegisterTask(..) so that the application can start running.

The application task function, ApplicationTask(..), is registered by FreeRTOS in the list of tasks that are ready to run. In the software examples, ApplicationTask(..) first performs application specific initialization of software on startup. Among other things it configures the event distribution functionality with event handlers for different types of events. Subsequently it goes into an eternal loop where it lets the event distributor handle any events. See the description in section 8.5. The FreeRTOS scheduler will ensure that the application task is given processor time according to its priority.

6.6.1 Command Class Lists Configuration

The application uses 3 Command Class lists to send out the Node Information Frame (NIF) depending on the inclusion status. These three lists define the NIF for:

- Not included
- Non-securely included
- Securely included

The following Command Class list is used when the node is not included or non-securely included. When the device is included, Security and Security_2 Command Classes are removed by the ZAF. This Command Class list is present in the NIF:

```
static code BYTE cmdClassListNonSecureNotIncluded[]
```

The following Command Class list is used when node is securely included. It represents the Command Classes that are supported non-securely even though the device has been securely included.

```
static code BYTE cmdClassListNonSecureIncludedSecure[]
```

The following Command Class list is used when node is securely included. It contains only the Command Classes that are supported securely. This Command Class list is advertised through the Security Command Class, Security Commands Supported Report Command.

```
static BYTE cmdClassListSecure[]
```

6.6.2 Endpoint Configuration

If your device must implement endpoints, the AGI and association group mapping must be set up in the *config_app.h* file. After configuring each endpoint in AGI (see Section 6.2.4), the AssociationInitEndpointSupport() must be called instead of the AssociationInit() for association group mapping (see Chapter 8.2.1 for the different initializations).

Finally, the transport layer needs to be initialized for endpoints by calling the Transport_AddEndpointSupport() function. This function sets up endpoints' functionalities and Command Class lists. Read [13] for more information about individual and aggregated endpoints.

```
void Transport_AddEndpointSupport(
 EP_FUNCTIONALITY_DATA* pFunctionality,
 EP_NIF* pList, BYTE sizeList);
```

7 Command classes

An essential feature of the ZAF is the communication through Command Classes. For this purpose, each of the Command Classes has a C module where incoming commands are handled, and outgoing commands are transmitted.

7.1 General Interfacing to CCs

The application will typically use CCs in the two following ways:

- Send an unsolicited command, or
- Respond to a received command

7.1.1 Unsolicited Transmission

See the following example of a CC API definition. Although some of the CCs does not comply with that naming scheme yet, it is the plan that more and more CCs will comply for every future version of the ZAF.

```
job_status_t CC_Basic_Set_tx(
  agi_profile_t * pProfile,
  uint8_t sourceEndpoint,
  uint8_t value,
  void (*pCbFunc)(transmission_result_t * pTransmissionResult));
```

A controlling device, e.g., a wall switch, may want to send a Basic Set Command. This is the purpose of the function given in the above example. Three of the four arguments are generic arguments found in all Command Class APIs: pProfile, sourceEndpoint, and pCbFunc. These arguments are detailed below. The argument bValue is specific for the Basic Set Command. To find more information about Basic Set or other application CCs and their commands, see [11].

- **pProfile**
  The pProfile points to an AGI profile. This must be one of the profiles that was set up in the AGI section of your application's *config_app.h* file.
- **sourceEndpoint**
  If the application is implementing endpoints, the source endpoint must be set to ENDPOINT_X where X is the number of the endpoint, e.g., ENDPOINT_1. If the application does not have endpoints, the source endpoint must be set to ENDPOINT_ROOT. The enumeration for ENDPOINT_ROOT and ENDPOINT_X can be found in the *ZW_TransportEndpoint.h* file.
- **pCbFunc**
  This argument is a pointer to a callback function with one argument of the type TRANSMISSION_RESULT. The definition of this type can be found in the *CC_Common.h* file. The given callback function will be called once for each node found in the Associations. When the transmission for the last node is done, the *isFinished* value will be set to TRANSMISSION_RESULT_FINISHED.

7.1.2 Respond to Received Command

Each CC implementation has a *handleCommandClass…()* function that extracts the received frame for a given Command Class. This function needs to be added into the application's *Transport_ApplicationCommandHandlerEx()* function switch-case block. Normally, the frame is carrying a "Set" or "Get" Command that results in a function call for reading or writing data. It is up to the application to implement these functions provided as external functions in the CC header file.

On the application side, *Transport_ApplicationCommandHandlerEx()* will figure out which CC should handle incoming frame:

```
SwitchOnOff.c

/* Call command class handlers */
switch (pCmd->ZW_Common.cmdClass)
{
   case COMMAND_CLASS_SWITCH_BINARY_V2:
     frame_status = handleCommandClassBinarySwitch(rxOpt, pCmd, cmdLength);
     break;
```

When the handler of the corresponding Command Class has been triggered, it will process the received frame further:

```
CC_Binary_Switch.c

switch (pCmd->ZW_Common.cmd)
  {
    case SWITCH_BINARY_GET:
      if(FALSE == Check_not_legal_response_job(rxOpt))
      {
        :
        /* Get the values from the application */
        pTxBuf->ZW_SwitchBinaryReportV2Frame.currentValue =
appBinarySwitchGetCurrentValue(rxOpt->destNode.endpoint);

       :
```

The CC handler will prepare the Report frame and then go back to the application to get application–specific data:

```
CMD_CLASS_BIN_SW_VAL
appBinarySwitchGetCurrentValue(uint8_t endpoint)
{
  UNUSED(endpoint);
  return onOffState;
}
```

7.1.3 CC Version

The version of the Command Class is set within each CC using the macro:
`REGISTER_CC(COMMAND_CLASS, CC_VERSION, CC_handler);`

This macro defines the version of CC in a single place and there is no need to set the CC version anywhere else.

In the case of a Binary Switch CC, the version is defined as:

```
REGISTER_CC(COMMAND_CLASS_SWITCH_BINARY, SWITCH_BINARY_VERSION_V2,
handleCommandClassBinarySwitch);
```

7.1.4 True Status Support

Starting from Z-Wave+ V2, if CC is on the list of mandatory command classes defined in [16], it must support True Status Engine(TSE). See more about TSE in Chapter 5.4.

7.2 Implementing a CC

Not all CCs have been implemented in the ZAF yet. Hence, in some cases, the required Command Class must be developed for the application.

7.3 Association Group Information CC

For a general introduction to AGI CC, refer to [12].

AGI CC is implemented in two C modules:

- CC_AssociationGroupInfo
- Agi

Together, these modules serve two purposes:

- Advertise capabilities of each association group, and
- Find associated nodes to which the application wants to send an unsolicited command.

7.3.1 API

- CC_AGI_Init()
- CC_AGI_LifeLineGroupSetup()
- CC_AGI_ResourceGroupSetup()

7.4 Battery CC

Battery CC is implemented in module CC_Battery. Additional functionality may be added to the application, if needed. See `CC_Battery_BatteryGet_handler()`in SensorPIR.c as an example.

7.5 Indicator CC

Indicator CC is mandatory since Z-Wave+ V2. It can be used to identify a device in the network, by sending command that will, for example, set the LED indicator to ON over a certain period.

Indicator CC is implemented in module CC_Indicator.

7.6 Notification CC Version 8

Notification CC is typically used in sensor applications because it supports many different notification/sensor types. For general information related to Notification CC, see [11].

Historically, this CC supported both push and pull mode and the ZAF implementation has tried to cover both modes. However, push and pull mode have been separated in version 8 of the CC and the ZAF now supports and tests only push mode, because supporting both modes at the same time results into a conflict in certification.

Notification CC is implemented by two C modules:

- CC_Notification
- notification

The Notification CC API consists of the following functions. For information about what they do, arguments, and return values, refer to the source header files of notification.h.

- InitNofication()
- AddNotification()
- DefaultNotificationStatus()
- NotificationEventTrigger()
- UnsolicitedNotificationAction()

The typical call order is showed below:

1. Initialize the Notification CC by calling **InitNofication()**.
2. Optionally, set the notification status for all available notifications (limited by the definition MAX_NOTIFICATIONS in config_app.h) by calling **DefaultNotificationStatus()**.
3. Call **AddNotification()** a number of times depending on how many different notification types the application implements.
4. Trigger an event by calling **NotificationEventTrigger()** and transmit it by calling **UnsolicitedNotificationAction()**.
5. If a state was triggered contrary to an event, the state can be cancelled by calling **NotificationEventTrigger()** with cancelling arguments and then transmitting the cancellation by calling **UnsolicitedNotificationAction()**.

7.7 Supervision CC

Refer to [7] for the definition of the Supervision Command Class.

CC Supervision is built into the Application Framework and handles Supervision communication on S2 encapsulated frames. Supervision is only supported for Set and Report commands.

The default setting is that CC Supervision supports only one Supervision Report per Supervision Get command. In Supervision Get command, the 'more status updates' field is set to CC_SUPERVISION_STATUS_UPDATES_NOT_SUPPORTED. The application does not need to initialize CC Supervision if the default configuration fulfills application demands.

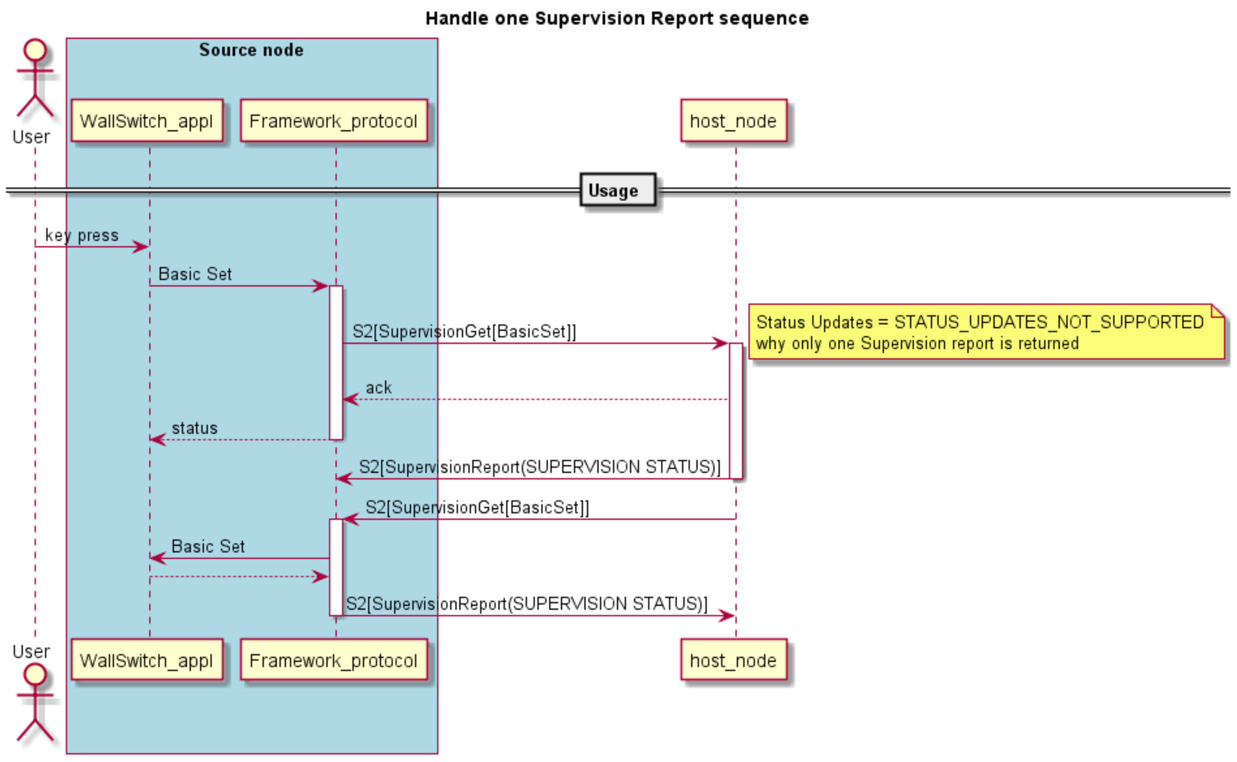
Initialization for CC Supervision:

```
void
CommandClassSupervisionInit( cc_supervision_status_updates_t status_updates,
  void (*pGetReceivedHandler)(SUPERVISION_GET_RECEIVED_HANDLER_ARGS * pArgs),
  void (*pReportReceivedHandler)(cc_supervision_status_t status, uint8_t duration));
```

7.7.1 Configuration Scenarios

7.7.1.1 Default Configuration

The application does not handle more than one Supervision report. The device receives a Supervision Get and replies with a Supervision Report containing CC_SUPERVISION_STATUS_SUCCESS. There is no need to call `CommandClassSupervisionInit()` for initialization.


Handle one Supervision Report sequence
Source node
User
WallSwitch_appl
Framework_protocol
host_node
Usage
key press
Basic Set
S2[SupervisionGet[BasicSet]]
Status Updates = STATUS_UPDATES_NOT_SUPPORTED
why only one Supervision report is returned
ack
status
S2[SupervisionReport(SUPERVISION STATUS)]
S2[SupervisionGet[BasicSet]]
Basic Set
S2[SupervisionReport(SUPERVISION STATUS)]
User
WallSwitch_appl
Framework_protocol
host_node

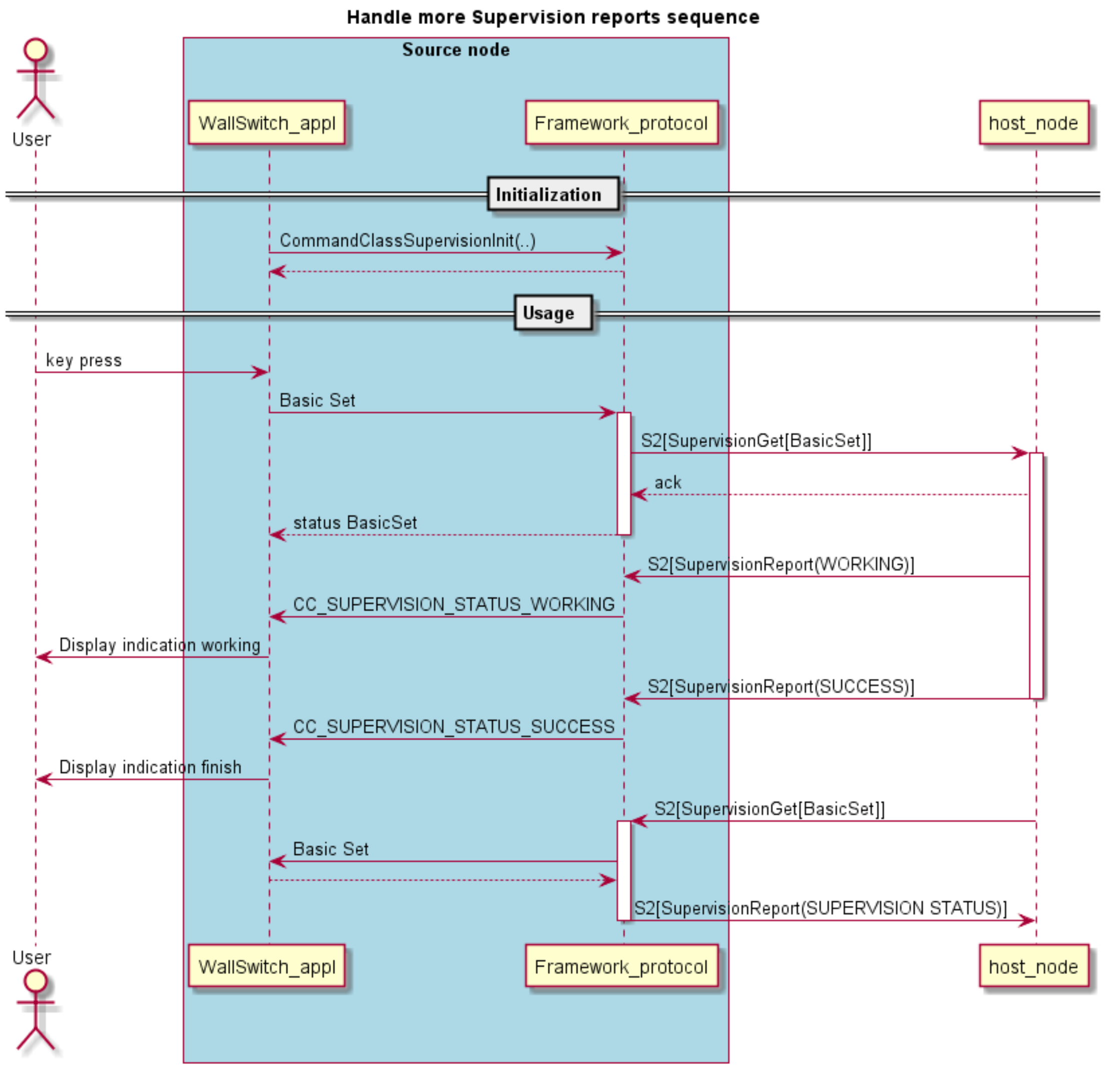
7.7.1.2 Handle More Supervision Reports

The application has the capability to display that a destination node is processing the transmitted command. An example is a Wall Controller with a display that shows a working device

(CC_SUPERVISION_STATUS_WORKING) until a given command has been performed (CC_SUPERVISION_STATUS_SUCCESS).

```
void ApplicationInitSW(..)
{
   CommandClassSupervisionInit(
     CC_SUPERVISION_STATUS_UPDATES_NOT_SUPPORTED,
     NULL,
     ZCB_CommandClassSupervisionReportReceivedHandler);
}

void ZCB_CommandClassSupervisionReportReceivedHandler(cc_supervision_status_t status, BYTE
duration)
{
  :
}
```

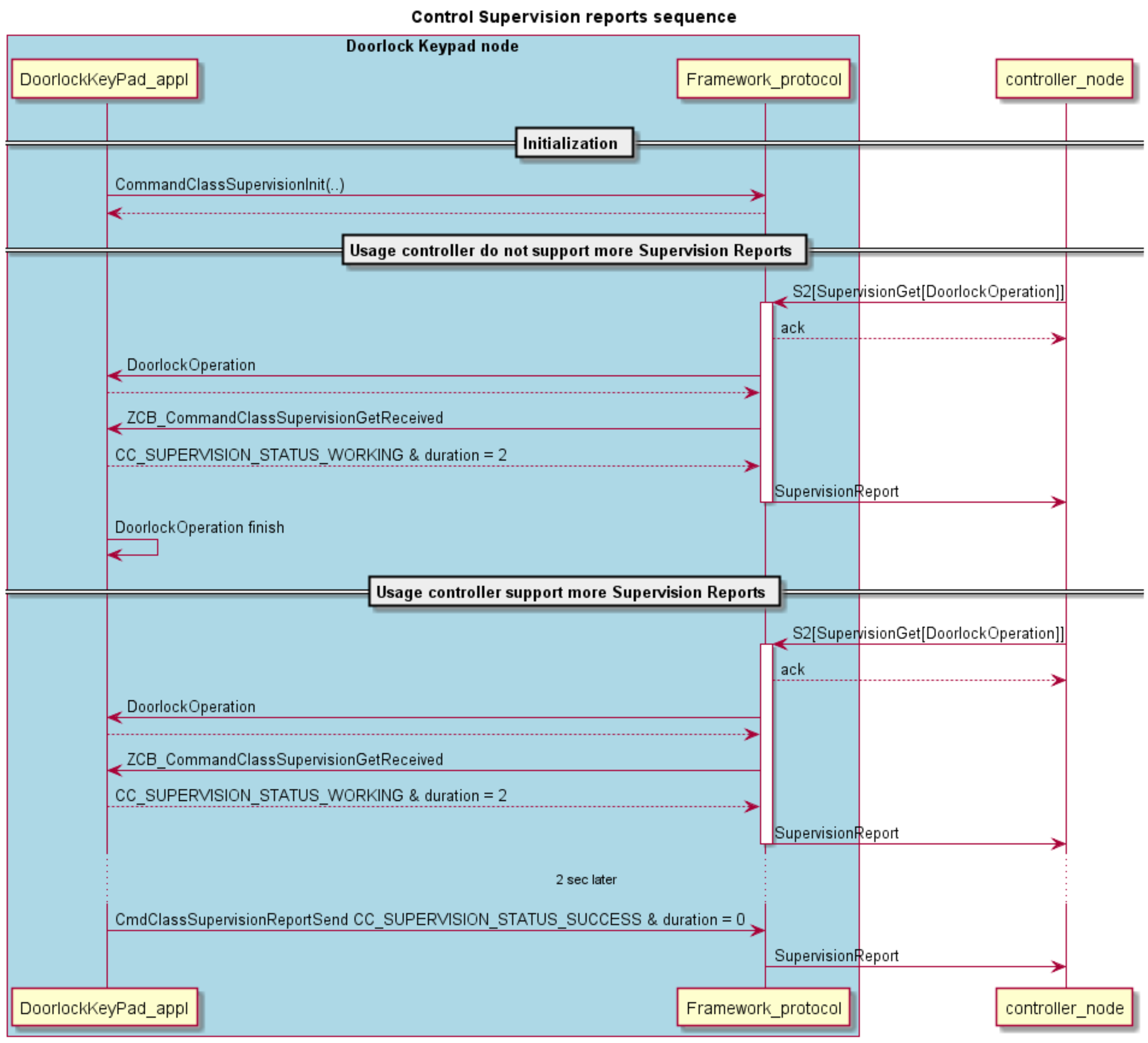

Handle more Supervision reports sequence
Source node
User
WallSwitch_appl
Framework_protocol
host_node
Initialization
CommandClassSupervisionInit(..)
Usage
key press
Basic Set
S2[SupervisionGet[BasicSet]]
ack
status BasicSet
S2[SupervisionReport(WORKING)]
CC_SUPERVISION_STATUS_WORKING
Display indication working
S2[SupervisionReport(SUCCESS)]
CC_SUPERVISION_STATUS_SUCCESS
Display indication finish
S2[SupervisionGet[BasicSet]]
Basic Set
S2[SupervisionReport(SUPERVISION STATUS)]
User
WallSwitch_appl
Framework_protocol
host_node

7.7.1.3 Control Supervision Reports

The application has the capability to send several Supervision Reports to report on an ongoing activity. An example is a Door Lock Key Pad that reports when a Door Lock Operation is started and when it is finished.

```
void AppStateManager(..)
{
   CommandClassSupervisionInit(
       CC_SUPERVISION_STATUS_UPDATES_NOT_SUPPORTED,
       ZCB_CommandClassSupervisionGetReceived,
       NULL);
}

void ZCB_CommandClassSupervisionGetReceived(SUPERVISION_GET_RECEIVED_HANDLER_ARGS * pArgs)
{
  :
}
```


Control Supervision reports sequence
Doorlock Keypad node
DoorlockKeyPad_appl
Framework_protocol
controller_node
Initialization
CommandClassSupervisionInit(..)
Usage controller do not support more Supervision Reports
S2[SupervisionGet[DoorlockOperation]]
ack
DoorlockOperation
ZCB_CommandClassSupervisionGetReceived
CC_SUPERVISION_STATUS_WORKING & duration = 2
SupervisionReport
DoorlockOperation finish
Usage controller support more Supervision Reports
S2[SupervisionGet[DoorlockOperation]]
ack
DoorlockOperation
ZCB_CommandClassSupervisionGetReceived
CC_SUPERVISION_STATUS_WORKING & duration = 2
SupervisionReport
2 sec later
CmdClassSupervisionReportSend CC_SUPERVISION_STATUS_SUCCESS & duration = 0
SupervisionReport
DoorlockKeyPad_appl
Framework_protocol
controller_node

8 Utilities

Some of the commonly used functionalities are handled by utilities, which can be adapted by the developer for an application.

8.1 AGI Module

The AGI module works with the Association Plus module to provide a general API for handling the setup and use of associations between multiple Z-Wave nodes. The AGI module interfaces to "`CC_AssociationGroupInfo.h`" for external access to AGI tables.

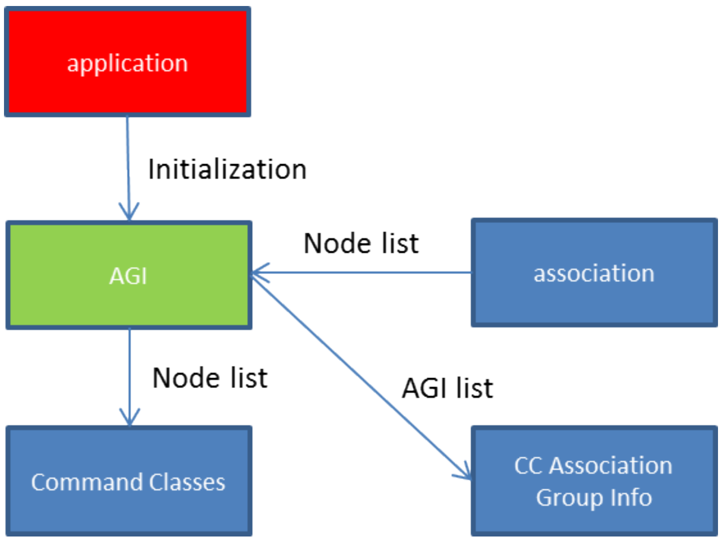


Figure 2. AGI Behavior Diagram

The AGI module contains a configuration part and an API to extract destination nodes for a specific association group.

8.1.1 Configuration of AGI

The AGI module has a constructer called *AGI_Init()*, which needs to be called before the AGI groups are configured. The constructer call resets the module parameters.

```
void AGI_Init(void);
```

The configuration of group 1 (Lifeline) is done when calling the `CC_AGI_LifeLineGroupSetup` function to setup the profile and command class groups. This function is also used to configure each endpoint of group 1 (endpoint Lifeline). For the Root Device configuration, the endpoint parameter must be set to *ENDPOINT_ROOT* (0):

```
void CC_AGI_LifeLineGroupSetup(
        cc_group_t const * const pCmdGrpList,
        uint8_t listSize,
        uint8_t endpoint);
```

The configuration of group 2..N for endpoint X is done by calling the *AGI_ResourceGroupSetup* function. The function sets up a list of type *AGI_GROUP* including profile, Command Class group, and group name. For the Root Device configuration, the endpoint parameter must be set to *ENDPOINT_ROOT* (0)

```
void AGI_ResourceGroupSetup(
        AGI_GROUP const * const pTable,
        uint8_t tableSize,
        uint8_t endpoint);
```

8.1.1.1 Example 1: How to Set Up AGI for a Wall Controller.

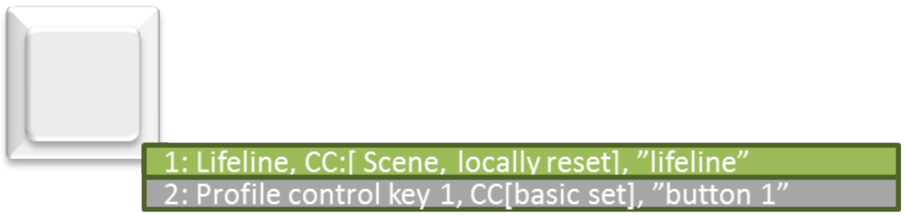

1: Lifeline, CC:[Scene, locally reset], "lifeline"
2: Profile control key 1, CC[basic set], "button 1"

Figure 3. Wall Controller that sends Command Class Scene and Device Reset Locally over Association group 1 (lifeline) and Command Class Basic over Association group 2

The applications use definitions from *config_app.h* to configure agiTableLifeLine[] and agiTableRootDeviceGroups[].

Table 2. Configuration of the Wall Controller

```
CMD_CLASS_GRP  agiTableLifeLine[] =
{
  {COMMAND_CLASS_CENTRAL_SCENE, CENTRAL_SCENE_NOTIFICATION},
  {COMMAND_CLASS_DEVICE_RESET_LOCALLY, DEVICE_RESET_LOCALLY_NOTIFICATION}
};

AGI_GROUP agiTableRootDeviceGroups[] =
{
  {
    ASSOCIATION_GROUP_INFO_REPORT_PROFILE_CONTROL,
    ASSOCIATION_GROUP_INFO_REPORT_PROFILE_CONTROL_KEY01,
    {COMMAND_CLASS_BASIC, BASIC_SET}, "Button 1"
  }
};

void AppStateManager(EVENT_APP event)
{
  switch (currentState) {
    case STATE_APP_STARTUP:
      if (EVENT_APP_INIT == event) {
        AGI_Init();

        CC_AGI_LifeLineGroupSetup(
          agiTableLifeLine,
          sizeof_array(agiTableLifeLine),
          ENDPOINT_ROOT);

        AGI_ResourceGroupSetup(
          agiTableRootDeviceGroups,
          sizeof_array (agiTableRootDeviceGroups),
          ENDPOINT_ROOT);
  :
```

8.1.1.2 Example 2: How to Extend the Wall Controller with 2 Buttons

In this example, each button represents an endpoint. The Root Device association group 2 is moved to endpoint 1. Root mapping to endpoints is not shown in the current example, see chapter 8.2.1 for more information.

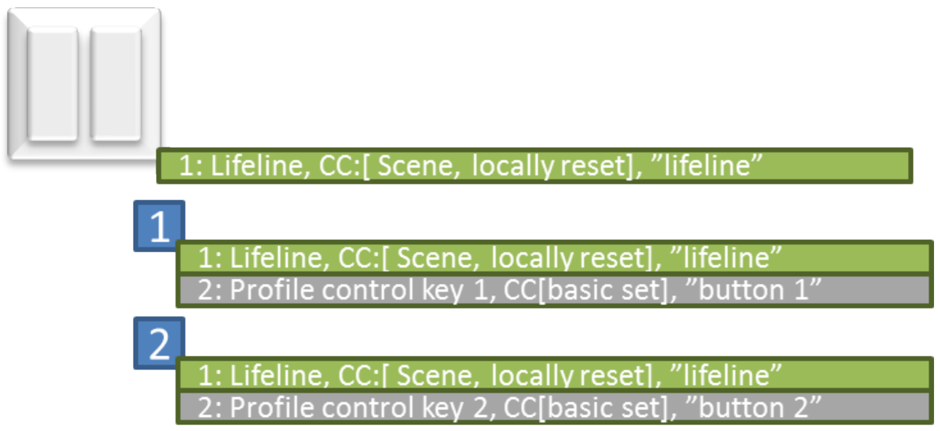

1: Lifeline, CC:[Scene, locally reset], ”lifeline”
1
1: Lifeline, CC:[Scene, locally reset], ”lifeline”
2: Profile control key 1, CC[basic set], ”button 1”
2
1: Lifeline, CC:[Scene, locally reset], ”lifeline”
2: Profile control key 2, CC[basic set], ”button 2”

Figure 4. Extended with Two Endpoints and Remove Root group 2

Table 3. Add Endpoints to a Wall Controller

```
CMD_CLASS_GRP agiTableLifeLine[] =
{
  {COMMAND_CLASS_CENTRAL_SCENE, CENTRAL_SCENE_NOTIFICATION},
  {COMMAND_CLASS_DEVICE_RESET_LOCALLY, DEVICE_RESET_LOCALLY_NOTIFICATION}
};

AGI_GROUP agiTableEndpoint1Groups[] =
{
  ASSOCIATION_GROUP_INFO_REPORT_PROFILE_CONTROL,
  ASSOCIATION_GROUP_INFO_REPORT_PROFILE_CONTROL_KEY01,
  {COMMAND_CLASS_BASIC, BASIC_SET}, "Button 1"
};

AGI_GROUP agiTableEndpoint2Groups[] =
{
  ASSOCIATION_GROUP_INFO_REPORT_PROFILE_CONTROL,
  ASSOCIATION_GROUP_INFO_REPORT_PROFILE_CONTROL_KEY02,
  {COMMAND_CLASS_BASIC, BASIC_SET}, "Button 2"
};

void AppStateManager(EVENT_APP event)
{
  switch (currentState) {
    case STATE_APP_STARTUP:
      if (EVENT_APP_INIT == event) {
        AGI_Init();
        /*Root device configuration*/
        CC_AGI_LifeLineGroupSetup(
          agiTableLifeLine,
          sizeof_array(agiTableLifeLine),
          ENDPOINT_ROOT);

        /*Endpoint configuration*/
        CC_AGI_LifeLineGroupSetup(
          agiTableLifeLine,
          sizeof_array(agiTableLifeLine),
          ENDPOINT_1);
        AGI_ResourceGroupSetup(
          agiTableEndpoint1Groups,
          sizeof_array(agiTableEndpoint1Groups),
          ENDPOINT_1);
        CC_AGI_LifeLineGroupSetup(
          agiTableLifeLine,
          sizeof_array(agiTableLifeLine),
          ENDPOINT_2);
        AGI_ResourceGroupSetup(
          agiTableEndpoint2Groups,
          sizeof_array(agiTableEndpoint2Groups),
          ENDPOINT_2);
  :
```

8.1.2 Using AGI

The AGI module handles the reading of the destination node list from the Association module. After the configuration of the AGI module, the application does not access the AGI module again. The Command Classes access the AGI module when calling the *ReqNodeList()* function for an unsolicited event job.

```
TRANSMIT_OPTIONS_TYPE_EX * ReqNodeList(
    AGI_PROFILE const * const pProfile,
    CMD_CLASS_GRP* pCurrentCmdGrp,
    uint8_t sourceEndpoint);
```

8.2 Association Module

The Association module is a small database handling all associations. The Association and Multi-Channel Associations Command Class modules are interfacing with the Association module for adding or removing associations. The database is stored in NVM. The AGI module handles the reading of the node list from the database.

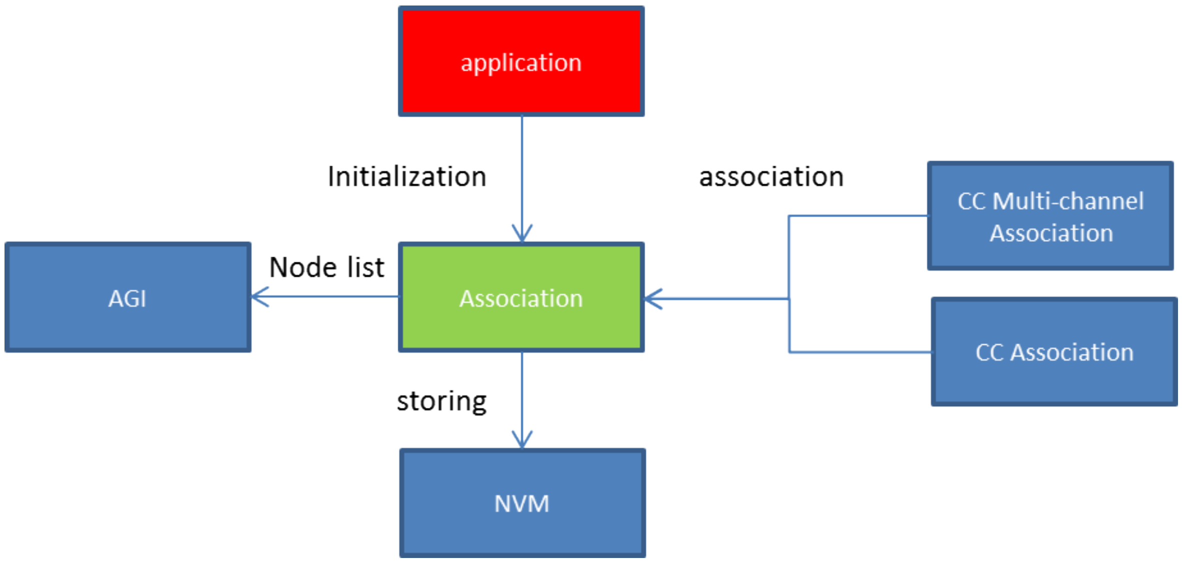


Figure 5. Association Behavior Diagram

8.2.1 Initialization

The initialization of the Association module is done in the `SetDefaultConfiguration` and `LoadConfiguration` functions and depends on support for endpoints.

The number of associations is calculated from the following definitions:

- Number of associations in a group: `MAX_ASSOCIATION_IN_GROUP`
- Number of association groups: `MAX_ASSOCIATION_GROUPS`
- Number of individual endpoints: `NUMBER_OF_INDIVIDUAL_ENDPOINTS`
- Number of aggregated endpoints: `NUMBER_OF_AGGREGATED_ENDPOINTS`
- The total number of endpoints is sum of individual and aggregated endpoints in the device: `NUMBER_OF_ENDPOINTS`

Definitions are configured in the *config_app.h* file and used in *association_plus.h*.

When no endpoint is implemented, the application must call the *AssociationInit()* function. The parameter “`forceClearMem`” is used for clearing the association NVM data, and “`pFS`“ is a pointer to the application file system in NVM.

```
void AssociationInit(bool forceClearMem, nvm3_Handle_t* pFS);
```

When the application implement endpoints, the application must call the *AssociationInitEndpointSupport()* function. The function takes the following parameters:

- “`forceClearMem`” is used for clearing the association NVM data.
- “`pMapping`” is used for backwards compatibility with non-Multi Channel devices. The root group mapping is used to configure the Root Device to advertise association groups on behalf of endpoints.
- “`nbrGrp`” is the number of groups in pMapping list.
- “`pFS`” is a pointer to the application file system in NVM.

```
void AssociationInitEndpointSupport(
            bool forceClearMem,
            ASSOCIATION_ROOT_GROUP_MAPPING* pMapping,
            uint8_t nbrGrp,
            nvm3_Handle_t* pFS);
```

8.2.1.1 Example 3: How to Use Group Mapping

In this example, the Wall Controller Root Device has 2 association groups and maps them to its endpoints; see previous example in 8.1.1.2. The goal is to map associations from the [Root Device, group 2] to the [endpoint 1, group 2] and from the [Root Device, group 3] to the [endpoint 2, group 2]. More information can be found in [6].

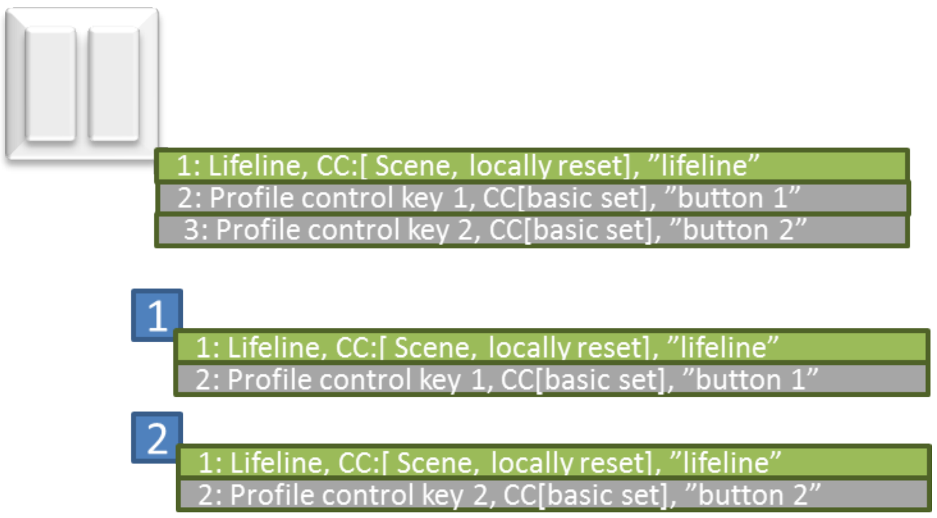

1: Lifeline, CC:[Scene, locally reset], "lifeline"
2: Profile control key 1, CC[basic set], "button 1"
3: Profile control key 2, CC[basic set], "button 2"
1
1: Lifeline, CC:[Scene, locally reset], "lifeline"
2: Profile control key 1, CC[basic set], "button 1"
2
1: Lifeline, CC:[Scene, locally reset], "lifeline"
2: Profile control key 2, CC[basic set], "button 2"

Figure 6. Wall Controller with Root Device Group Mapping

Table 4. AGI Root groups 2 and 3 (Mapped to Endpoint 1 and 2) Setup

```
AGI_GROUP agiTableRootDeviceGroups[] =
{
  {
    ASSOCIATION_GROUP_INFO_REPORT_PROFILE_CONTROL,
    ASSOCIATION_GROUP_INFO_REPORT_PROFILE_CONTROL_KEY01,
    {COMMAND_CLASS_BASIC, BASIC_SET}, "Button 1"
  },
  {
    ASSOCIATION_GROUP_INFO_REPORT_PROFILE_CONTROL,
    ASSOCIATION_GROUP_INFO_REPORT_PROFILE_CONTROL_KEY02,
    {COMMAND_CLASS_BASIC, BASIC_SET}, "Button 2"
  }
};

/*root grp, endpoint, endpoint group*/
ASSOCIATION_ROOT_GROUP_MAPPING rootGroupMapping[] = {
  {ASS_GRP_ID_2, ENDPOINT_1, ASS_GRP_ID_2},
  {ASS_GRP_ID_3, ENDPOINT_2, ASS_GRP_ID_2}
};

void AppStateManager(EVENT_APP event)
```

```
{
  switch (currentState) {
    case STATE_APP_STARTUP:
      if (EVENT_APP_INIT == event) {
        AGI_Init();

        /*Root device configuration*/
        AGI_ResourceGroupSetup(
          agiTableRootDeviceGroups,
          sizeof_array(agiTableRootDeviceGroups),
          ENDPOINT_ROOT);
  :
}

void SetDefaultConfiguration(void)
{
  :
  AssociationInitEndpointSupport(TRUE,
                                 rootGroupMapping,
                                 sizeof_array(rootGroupMapping));
  :
}

bool LoadConfiguration(void)
{
  :
  AssociationInitEndpointSupport(FALSE,
                                 rootGroupMapping,
                                 sizeof_array(rootGroupMapping));
  :
}
```

8.3 Interfacing Firmware Update Module “ota_util”

The Ota_util module handles OTA firmware update. It is interfacing to Firmware Update Meta Data CC and delivers an interface to the application to manage the process.

Ota_util API:

```
BYTE
OtaInit(
  BOOL (CODE *pOtaStart)(WORD fwId, WORD CRC),
  VOID_CALLBACKFUNC(pOtaExtWrite)( BYTE *pData, BYTE dataLen),
  VOID_CALLBACKFUNC(pOtaFinish)(BYTE val));
```

Input parameters `pOtaStart` and `pOtaFinish` is used to inform the application of the status of firmware update and give the application the capability to control the start of firmware update. It is possible to not call `OtaInit` and the process runs without the application with standard parameters for `txOption`. Input parameter `pOtaExtWrite` is used to update host firmware.

[3] gives a detailed description on how to interface to the “ota_util” module.

8.4 Peripheral Drivers

Several peripheral drivers are available (EMDRV and EMLIB) and must be linked to the application before the hardware device in question can be accessed. EMDRV exist on top of the lower level EMLIB. Source code is also available for customization.

The Z-Wave 700 SDK is event-driven that requires an event-driven design to access the hardware device. Calls are already implemented for handling GPIOs (see Section 8.4.1). Refer to links below for further details.

Software Documentation (e.g., EMDRV and EMLIB)

`https://siliconlabs.github.io/Gecko_SDK_Doc/efr32fg13/html/index.html`

Technical Resource Search (Application Notes)

`https://www.silabs.com/support/resources.ct-application-notes.ct-example-code.p-microcontrollers_32-bit-mcus?`

32-bit Peripheral Examples (EMLIB)

`https://github.com/SiliconLabs/peripheral_examples`

8.4.1 GPIO

For the Wireless Starter Kit Mainboard (BRD4001A) with a ZGM130S Radio Board (BRD4202A) and a Buttons and LEDs EXP Board (BRD8029A), the file *board.c* provides a high-level support for buttons and LEDs (Note: *board.c* does not currently make use of the EFR32 Flex Gecko board support packages).

The mapping of buttons and LEDs to actual GPIO ports and pins are defined in the two header files: *extension_board_4001a.h* and *radio_board_zgm130s.h* included with *board.h.*

The board should be initialized with Board_Init() from ApplicationInit().

```
ZW_APPLICATION_STATUS ApplicationInit(EResetReason_t eResetReason)
{
  :
  Board_Init();
  :
}
```

To subscribe to button events from a specific button, simply call *Board_EnableButton()* from the application thread.

```
void Board_EnableButton(button_id_t btn);
```

The application will then receive events of type *BUTTON_EVENT* (*<btn_id>_DOWN, <btn_id>_UP, <btn_id>_SHORT_PRESS, <btn_id>_HOLD, <btn_id>_LONG_PRESS*).

LEDs are controlled with the *Board_SetLed()* function that simply takes an *LED_ON* or *LED_OFF* value as parameter:

```
void Board_SetLed(led_id_t led, led_state_t state);
```

8.4.1.1 GPIO port usage in serial API

The GPIO port usage in the serial API applications for EFR32ZG14 and ZGM130S respectively are as follows:

GPIO	EFR32ZG14 Usage
PA0	**UART/VCOM TX**
PA1	**UART/VCOM RX**
PB11	*For future use*
PB12	*For future use*
PB13	*For future use*
PB14	**SAW Filter Select 1**
PB15	**SAW Filter Select 2**
PC10	*For future use*
PC11	*For future use*
PD13	*For future use*
PD14	**VCOM Enable**
PD15	*For future use*
PF0	**Serial Wire Debug Clock**
PF1	**Serial Wire Debug Data I/O**
PF2	**Serial Wire Debug Output**
PF3	*For future use*

GPIO	ZGM130S Usage
PA0	**UART0/VCOM TX**
PA1	**UART0/VCOM RX**
PA2	**UART0/VCOM CTS**
PA3	**UART0/VCOM RTS**
PA4	*For future use*
PA5	**VCOM Enable**
PB11	*For future use*

PB12	*For future use*
PB13	*For future use*
PB14	**SAW Filter Select 1**
PB15	**SAW Filter Select 2**
PC6	*For future use*
PC7	*For future use*
PC8	*For future use*
PC9	*For future use*
PC10	*For future use*
PC11	*For future use*
PD9	*For future use*
PD10	*For future use*
PD11	*For future use*
PD12	*For future use*
PD13	*For future use*
PD14	*For future use*
PD15	*For future use*
PF0	**Serial Wire Debug Clock**
PF1	**Serial Wire Debug Data I/O**
PF2	**Serial Wire Debug Output**
PF3	*For future use*
PF4	*For future use*
PF5	*For future use*
PF6	*For future use*
PF7	*For future use*

8.4.2 UART Driver

The UART driver uses the UART RX interrupt to wake up the application when it has collected enough relevant data. This is done as follows:

1. Define a new application event, e.g., EAPPLICATIONEVENT_SERIALDATARX in the enum EApplicationEvent.
2. Install an event handler (this is a callback) in static const EventDistributorEventHandler g_aEventHandlerTable.

```
Trigger the event from the ISR
Void USART0_RX_IRQHandler() {
… Collect data …

  If(enough_data) {
    xTaskNotifyFromISR(g_AppTaskHandle,
      1<< EAPPLICATIONEVENT_SERIALDATARX,
      eSetBits,
      NULL);
  }
}
```

3. Enable the UART RX IRQ (for more information, read the standard EMLIB documentation such as [22]).

The event handler is called in the application thread context, which is good because this ensures thread safety in the application. It is not recommended to add more threads to the application.

8.4.3 ADC Driver

The Z-Wave Application Framework contains an API for measuring the supply voltage using the Analog-to-Digital Converter (ADC) of the ZGM130S.

Applications can use this API to get the current supply voltage level. The typical use case is for obtaining the battery status in battery operated devices.

The API consists of the following functions:

Function:	Description:
void ZAF_ADC_Enable(void)	Initialize and enables the ADC
void ZAF_ADC_Disable(void)	Disables the ADC
uint32_t ZAF_ADC_Measure_VSupply(void)	Returns the supply voltage in millivolts

API function prototypes are defined in the following header file, which much be included from the Application:

#include "ZAF_adc.h"

Typical usage example:

```
ZAF_ADC_Enable();                       // Enable the ADC

VBATT_mV = ZAF_ADC_Measure_VSupply(); // Read the battery voltage
```

```
    ZAF_ADC_Disable();                  // All done - disable the ADC
```

Further usage examples can be found in the certified Z-Wave applications DoorLockKeyPad and SensorPIR.

8.5 Event Distributor

The objectives of the Event Distributor are to receive events on several FIFO message queues, wake up the application task whenever a message is available, and call the event handler associated with each queue.

8.5.1 Event Loop

A Z-Wave application task is basically an endless event loop calling *EventDistributorDistribute()*.

```
ApplicationTask(SApplicationHandles* pAppHandles)
{
  /* Task initialization */
  :
  /* Endless event loop */
  for (;;)
  {
    EventDistributorDistribute(&g_EventDistributor, 10, 0);
  }
}
```

The events to the application task arrives on several queues. If none of the queues has messages to read, the application task will sleep for a number of milliseconds (10 ms in the example above). Whenever a message arrives on a queue, the application task will wake up and dispatch the message to one of the configured functions.

The first parameter of type *SEventDistributor* contains configuration data for the event distributor.

```
uint32_t EventDistributorDistribute(const SEventDistributor* pThis,
                                    uint32_t iEventWait,
                                    uint32_t NotificationClearMask);
```

The *SEventDistributor* variable used with *EventDistributorDistribute()* must first be initialized with EventDistributorConfig():

```
EEventDistributorStatus EventDistributorConfig(
                            SEventDistributor* pThis,
                            uint8_t iEventHandlerTableSize,
                            const EventDistributorEventHandler* pEventHandlerTable,
                            void(*pNoEvent)(void) );
```

The *pEventHandlerTable* is an array of *EventDistributorEventHandler*, which is simply a function pointer to the event handler functions.

```
typedef void (*EventDistributorEventHandler)(void);
```

Note: The parameter *pNoEvent*, if non-null, will be called every time *EventDistributorDistribute()* wakes up and finds that there are no messages to process.

Each message queue is assigned a *notification bit numbe*r. The order of event handler functions in *pEventHandlerTable* must correspond to those bit numbers; e.g., when a message arrives on the queue with notification bit number 2, the event handler function at array index 2 in *pEventHandlerTable* will be called.

The bit numbers are conveniently defined with the *EApplicationEvent* enumeration:

```
typedef enum
{
  EAPPLICATIONEVENT_TIMER = 0,
  EAPPLICATIONEVENT_ZWRX,
  EAPPLICATIONEVENT_ZWCOMMANDSTATUS,
  EAPPLICATIONEVENT_APP
} EApplicationEvent;
```

The event handler table to use with *EventDistributorConfig()* can then be defined like this:

```
static const EventDistributorEventHandler m_aEventHandlerTable[] =
{
  AppTimerNotificationHandler,
  EventHandlerZwRx,
  EventHandlerZwCommandStatus,
  EventHandlerApp
};
```

An implementation of *AppTimerNotificationHandler()* is provided by the framework, the other remaining functions must be implemented by the application developer.

8.5.2 Event Queues

The queue creation (and notification bit assignment) for message queues receiving the Z-Wave packages and Z-Wave command results are specified in *ApplicationInit()* with *ZW_ApplicationRegisterTask()*. Application timer events are not placed on a queue, but are sent to the application using the notification interface and must therefore be assigned a notification bit number by calling *AppTimerInit()*:

```
ZW_APPLICATION_STATUS ApplicationInit(EResetReason_t eResetReason)
{
  :
  AppTimerInit(EAPPLICATIONEVENT_TIMER, NULL);
  :
  ZW_ApplicationRegisterTask(ApplicationTask,
                             (1536 / 4),
                             EAPPLICATIONEVENT_ZWRX,
                             EAPPLICATIONEVENT_ZWCOMMANDSTATUS,
                             &ProtocolConfig);
  :
}
```

The application message queue must be created with the FreeRTOS function *xQueueCreateStatic()*. First, a storage space for the queue must be defined (in this example, we are allocating a space for 5 application events on the queue). After the queue has been created, it must be assigned the application notification bit with *QueueNotifyingInit()*. Finally, the *ZAF_EventHelper* module must be configured to use the application queue:

```
static EVENT_APP eventQueueStorage[5];
static StaticQueue_t m_AppEventQueueObject;
static QueueHandle_t m_AppEventQueueHandle;
static SQueueNotifying m_AppEventNotifyingQueue;
static TaskHandle_t m_AppTaskHandle;

void
ApplicationTask(SApplicationHandles* pAppHandles)
{
  m_AppTaskHandle = xTaskGetCurrentTaskHandle();
  :
  m_AppEventQueueHandle = xQueueCreateStatic(sizeof_array(eventQueueStorage),
                                             sizeof(eventQueueStorage[0]),
                                             (uint8_t*)eventQueueStorage,
                                             &m_AppEventQueueObject);

  QueueNotifyingInit(&m_AppEventNotifyingQueue,
                     m_AppEventQueueHandle,
                     m_AppTaskHandle,
                     EAPPLICATIONEVENT_APP);

  ZAF_EventHelperInit(&m_AppEventNotifyingQueue);
  :
}
```

The *ZAF_EventHelper* module provides a simple way of placing events on the application event queue:

```
File zaf_event_helper.h:

void ZAF_EventHelperInit(SQueueNotifying * pQueueNotifyingHandle);
bool ZAF_EventHelperEventEnqueue(const uint8_t event);
bool ZAF_EventHelperEventEnqueueFromISR(const uint8_t event);
```

8.5.3 Event Handler

Each of the event handlers in the EventDistributorEventHandler table used when calling EventDistributorConfig() is simply called when one or more messages are available on its message queue. The event handler must receive and process all available messages from the queue. For example, the application event handler will typically be implemented as follows, where all events are simply forwarded to the application state manager function:

```
static void EventHandlerApp(void)
{
  uint8_t event;
  while (xQueueReceive(m_AppEventQueue, &event, 0) == pdTRUE)
  {
    AppStateManager((EVENT_APP)event);
  }
}
```

8.5.4 Job Event Queue

Applications often have the need to temporarily queue up events that first should be processed when the active job is finished. For this reason, the *ZAF_JobHelper* module provides its own job event queue that is not handled by the Event Distributor. The application must explicitly enqueue and dequeue events on the job queue as needed. First, the application must call *ZAF_JobHelperInit()* to initialize the job event queue to hold *JOB_QUEUE_BUFFER_SIZE* events. The *ZAF_JobHelperJobEnqueue()* and *ZAF_JobHelperJobDequeue()* can then be used to put and get events on the queue.

```
File zaf_job_helper.h:

#define JOB_QUEUE_BUFFER_SIZE 3

void ZAF_JobHelperInit(void);
bool ZAF_JobHelperJobEnqueue(uint8_t event);
bool ZAF_JobHelperJobDequeue(uint8_t * pEvent);
```

8.5.5 Simple Event Handling

In the simplest form, only the application event queue will be used, for example, to execute a single command when a user presses a button. The file board.c uses the *ZAF_EventHelper* module to send button events to the application queue which will result in the *EventHandlerApp()* function activating *AppStateManager()* with the button event.

1. In learn mode the user presses key01, which changes the state to *STATE_APP_STARTUP:*

```
void AppStateManager(EVENT_APP event)
{
  :
  switch(currentState) {
    :
    case STATE_APP_LEARN_MODE:
      :
      if ((BTN_EVENT_SHORT_PRESS(APP_BUTTON_LEARN_RESET) == (BUTTON_EVENT)event ||
          (EVENT_SYSTEM_LEARNMODE_STOP == (EVENT_SYSTEM)event)) {
        :
        ChangeState(STATE_APP_STARTUP);
        :
      }
      :
  }
}
```

When learn process is completed, *EventHandlerZwCommandStatus* will be triggered by protocol with status *EZWAVECOMMANDSTATUS_LEARN_MODE_STATUS*, which will trigger *LearnCompleted()*, from where *ZAF_EventHelperEventEnqueue((EVENT_APP) EVENT_SYSTEM_LEARNMODE_FINISHED)* is called:

```
static void LearnCompleted(void)
{
..:
 ZAF_EventHelperEventEnqueue((EVENT_APP) EVENT_SYSTEM_LEARNMODE_FINISHED);
 Transport_OnLearnCompleted(bNodeID);
}
```

2. In state *STATE_APP_STARTUP*, the wanted functionality is executed and the state changes back to the idle state *STATE_APP_IDLE*.

```
:
case STATE_APP_STARTUP:
  if (EVENT_APP_INIT == event) {
    :
    ChangeState(STATE_APP_IDLE);
  }
  :
```

8.5.6 Multiple Event Jobs Handling

It is possible to enqueue more jobs during the execution with the *ZAF_JobHelper* module.

The example below illustrates how the job queue is used to expand a button press into multiple actions (send a Basic Set followed by send a Notification) to be performed in the state *STATE_APP_TRANSMIT_DATA*.

```
case STATE_APP_IDLE:
  :
  if ((BTN_EVENT_DOWN(PIR_EVENT_BTN) == (BUTTON_EVENT)event) ||
      (BTN_EVENT_HOLD(PIR_EVENT_BTN) == (BUTTON_EVENT)event))
  {
    :
    ChangeState(STATE_APP_TRANSMIT_DATA);

    if (false == ZAF_EventHelperEventEnqueue(EVENT_APP_NEXT_EVENT_JOB))
    {
      DPRINT("\r\n** EVENT_APP_NEXT_EVENT_JOB fail\r\n");
    }
    /*Add event's on job-queue*/
    ZAF_JobHelperJobEnqueue(EVENT_APP_BASIC_START_JOB);
    ZAF_JobHelperJobEnqueue(EVENT_APP_NOTIFICATION_START_JOB);
    ZAF_JobHelperJobEnqueue(EVENT_APP_START_TIMER_EVENTJOB_STOP);
  }
```

In state *STATE_APP_TRANSMIT_DATA*, shown next, the event *EVENT_APP_NEXT_EVENT_JOB* starts the job by fetching the first job event from the job event queue.

Each *send*-function is provided with a pointer to the call-back function *ZCB_JobStatus()* that will be called by the framework when the transmission has completed. *ZCB_JobStatus()* will simply place a *EVENT_APP_NEXT_EVENT_JOB* event on the application event queue to trigger the processing of the next job event.

```
void ZCB_JobStatus(TRANSMISSION_RESULT * pTransmissionResult)
{
  if (TRANSMISSION_RESULT_FINISHED == pTransmissionResult->isFinished)
  {
    ZAF_EventHelperEventEnqueue(EVENT_APP_NEXT_EVENT_JOB);
  }
}
```

8.6 Power Manager

The Z-Wave chip provides several low energy modes (EM). Each energy mode manages whether the CPU and its various peripherals are available. The energy modes range from EM0 Active to EM4 Shutoff. EM0 Active mode provides the highest number of features, enabling the CPU, Radio, and peripherals with the highest clock frequency. EM4 Shutoff Mode provides the lowest power state, allowing the chip to return to EM0 Active on a wake-up condition.

To achieve the longest possible battery life, it is essential that Z-Wave applications on battery powered devices reduce the time spent in the higher energy modes as much as possible.

The Z-Wave Power Manager owned by the Z-Wave protocol thread controls what energy mode the RTOS should go to when idle. It uses the concept of *Power Locks* where regions of code can lock the chip from entering a specific energy mode. The Power Manager keeps track of all power locks and ensures that the chip does not at any time enter a power mode lower than what is currently requested.

To interact with the Power Manager from an application, use the Power Management API in *ZAF_PM_Wrapper.h*. To use it, first initialize the interface with *API_IF_init()*, and then register a power lock with *ZAF_PM_Register()*.

```
void API_IF_init(SApplicationHandles* pAppHandles);
void ZAF_PM_Register(SPowerLock_t* handle, pm_type_t type);
```

Use the power lock type pm_type_t to tell the Power Manager if the power lock should "protect" code that need access to the radio (PM_TYPE_RADIO: do not enter EM2/EM3/EM4) or just some peripherals (PM_TYPE_PERIPHERAL: don't enter EM3/EM4).

After the power lock has been registered, *ZAF_PM_StayAwake()* and *ZAF_PM_Cancel()* can be used to wrap the code that needs access to the radio or peripherals:

```
void ZAF_PM_StayAwake(SPowerLock_t* handle, unsigned int msec);
void ZAF_PM_Cancel(SPowerLock_t* handle);
```

If the *msec* parameter to *ZAF_PM_StayAwake()* is non-zero, the power lock will automatically be cancelled after the specified time (in that case, a call to *ZAP_PM_Cancel()* is not required).

```
static SPowerLock_t m_RadioPowerLock;

static void
ApplicationTask(SApplicationHandles* pAppHandles)
{
  API_IF_init(pAppHandles);

  ZAF_PM_Register(&m_RadioPowerLock, PM_TYPE_RADIO);
  :
}

void SomeFunction(void)
{
  ZAF_PM_StayAwake(&m_RadioPowerLock, 0);
  :
  /* Do something where the radio module is required */
  :
  ZAF_PM_Cancel(&m_RadioPowerLock);
  :
}
```

Several callbacks from the protocol to the framework supported enabling execution of specific code as the last thing before entering sleep mode. Up to three callbacks are available. For details refer to function ZAF_PM_SetPowerDownCallback() in ZAF_PM_Wrapper.h.

The applications DoorLockKeyPad (FLiRS node) and SensorPIR (battery operated node) both use power locks. SensorPIR goes all the way to EM4 when idle, but since a FLiRS node must wake up every 250 ms or 1000 ms, the DoorLockKeyPad application only sleeps in EM2 to achieve a fast wakeup time.

8.7 Application Timers

The AppTimer module provides an application interface for software timers.

```
File: AppTimer.h

void AppTimerInit(uint8_t iTaskNotificationBitNumber, TaskHandle_t ReceiverTask);
void AppTimerSetReceiverTask(TaskHandle_t ReceiverTask);
bool AppTimerRegister(SSwTimer* pTimer, bool bAutoReload, void(*pCallback)(SSwTimer*
pTimer) );
void AppTimerNotificationHandler(void);
```

All software timers are essentially FreeRTOS timers running in the high priority FreeRTOS timer task. Any timer callbacks are normally executed in the context of the timer task. Using the *AppTimer* module, you can ensure that timer callback functions are executed in the context of the application task.

> ***Note:** If a timer expires while e.g., a battery-operated device, is deep sleeping in energy mode EM4 Shutoff, the device will wake up, but the timer callback function will* ***not*** *be executed, since the device will be going through a full startup initialization.*

To start using application timers, you need a *SSwTimer* instance and you need to initialize the *AppTimer* module. That is usually done in *ApplicationInit()*:

```
#include <AppTimer.h>

static SSwTimer myAppTimer;

ZW_APPLICATION_STATUS ApplicationInit(EResetReason_t eResetReason)
{
  AppTimerInit(EAPPLICATIONEVENT_TIMER, NULL);
}
```

See Section 8.5.1 for a description of the *EAPPLICATIONEVENT_TIMER* notification bit number and the framework function *AppTimerNotificationHandler()*.

In the *ApplicationTask()* function, you need to register the handle of the application task with the *AppTimer* module and also register all your *SSwTimers* (it is possible to register up to eight application timers):

```
void ApplicationTask(SApplicationHandles* pAppHandles)
{
  m_AppTaskHandle = xTaskGetCurrentTaskHandle();

  AppTimerSetReceiverTask(m_AppTaskHandle);
  AppTimerRegister(&myAppTimer, false, ZCB_MyAppTimerCallback);
}
```

The timeout callback could be implemented like this:

```
void ZCB_MyAppTimerCallback(SSwTimer *pTimer)
{
  UNUSED(pTimer);
  ZAF_EventHelperEventEnqueue(EVENT_APP_MY_TIMER_TIMEOUT);
}
```

To start and stop an application timer, simply use the functions from the SwTimer module (all timeout values in milliseconds):

```
File: SwTimer.h

ESwTimerStatus TimerStart(SSwTimer* pTimer, uint32_t iTimeout);
ESwTimerStatus TimerStartFromISR(SSwTimer* pTimer, uint32_t iTimeout);
ESwTimerStatus TimerRestart(SSwTimer* pTimer);
ESwTimerStatus TimerRestartFromISR(SSwTimer* pTimer);
ESwTimerStatus TimerStop(SSwTimer* pTimer);
ESwTimerStatus TimerStopFromISR(SSwTimer* pTimer);
bool TimerIsActive(SSwTimer* pTimer);
```

When one or more application timers expire, the framework function *AppTimerNotificationHandler()* is called, which in turn calls the timer callback function for each of the timers that have expired.

If an *autoloading* timer (i.e., restarts automatically) is configured with a very small timeout value, it is possible it can expire multiple times before *AppTimerNotificationHandler()* is called. In that case, the timeout events will *not* be queued; *AppTimerNotificationHandler()* will only be called a single time.

9 Firmware Update Images and Bootloader

The purpose of this section is to describe how to generate and manage firmware update images. The ZW700 SDK comes with three bootloader images. One is the OTW image for the FG14 module. This bootloader is only meant for FG14 devices, which runs the SerialAPI.

The other two bootloaders are the OTA bootloader for the ZGM130S module and an OTW bootloader for the ZGM130S. The OTA bootloader is needed for all ZW700 based devices, which implement firmware updates; the OTW bootloader is for devices, which update firmware using the serial port from another host controller. The OTA bootloader is triggered when an image has been transferred over the air using the FIRMWARE_UPDATE command class. The transferred image must be an image in Gecko Boot Loader(GBL) format. The bootloaders provided in the SDK require the GBL image to be signed.

Three steps are needed for performing an OTA update:

1. The OTA bootloader must be flashed.
2. The Signing keys and optionally an encryption key must be flashed.
3. A signed image must be transferred using the firmware update command class.

The OTA key locations are as follows depending on SoC:

- For ZGM13: protocol\z-wave\BootLoader\sample-keys
- For EFR32ZG14P: protocol\z-wave\BootLoader\ZG14_keys

Further information about the bootloaders can be found in [17].

9.1 Generate GLB Files

To generate the GBL files needed for the OTA update, a signing keypair must first be created. It is the intention that a vendor will keep the signing keypair for the lifetime of the product. These keys are used to sign all firmware versions for the whole lifetime of the product. An encryption key must also be created, this key is intended for encrypting the GBL file. Encryption makes it harder for a bootlegger to copy the product.

The signing keys can be created by using Simplicity Commanders command line interface.

```
commander.exe gbl keygen --type ecc-p256 -o vendor_sign.key
```

This step will create 3 files:

1. *vendor_sign.key* - This is the private key and must be kept safely by the product manufacturer.
2. *vendor_sign.key.pub* - This is the public key.
3. *vendor_sign.key-tokens.txt* - This is the public key in another format which can be programmed into the device at manufacturing using simplicity commander.

A vendor may choose to have a key pair like this for all his products, or one for each product type.

An encryption key can be generated as follows:

```
commander.exe gbl keygen --type aes-ccm -o vendor_encrypt.key
```

Once the two keys have been obtained, a GBL file may be produced as follows:

```
commander.exe gbl create appname.gbl  --app appname.hex  --sign
vendor_sign.key --encrypt vendor_encrypt.key  --compress lz4
```

This should be done each time a new firmware is produced.

9.2 Flashing the Boot Loader and App

It is possible to flash the boot loader including the public signing key and the encryption key using commander.exe. The list below shows args to commander.exe for a board having SN 440049475 as an example:

1. Erase Flash args:device masserase -s 440049475 -d Cortex-M4
2. Reset args:device reset -s 440049475 -d Cortex-M4
3. Erase Bootloader args:device pageerase --region @bootloader -s 440049475 -d Cortex-M4
4. Erase Lockbits args:device pageerase --region @lockbits -s 440049475 -d Cortex-M4
5. Program Bootloader args:flash C:\dk2\Apps458\BootLoader_7.11.0_458\OTA-bootloader-fg13-combined.s37 -s 440049475 -d Cortex-M4
6. Program Keys args:flash --tokengroup znet --tokenfile C:\dk2\Apps458\BootLoader_7.11.0_458\sample_encrypt.key --tokenfile C:\dk2\Apps458\BootLoader_7.11.0_458\sample_sign.key-tokens.txt -s 440049475 -d Cortex-M4
7. Erase Flash args:device masserase -s 440049475 -d Cortex-M4
8. Reset args:device reset -s 440049475 -d Cortex-M4
9. Program Flash args:flash "C:\dk2\Apps458\PowerStrip\ZW_PowerStrip_7.11.0_458_ZGM130S_REGION_IN.hex" --address 0x0 --serialno 440049475 --device Cortex-M4
10. Reset args:device reset -s 440049475 -d Cortex-M4

Minor modifications are needed in steps 3 and 4 if the commands are run on a Windows Powershell:

3. Erase Bootloader args:device pageerase --region "@bootloader" -s 440049475 -d Cortex-M4
4. Erase Lockbits args:device pageerase --region "@lockbits" -s 440049475 -d Cortex-M4

Note that the bootloader and keys are not erased by a normal mass erase.

References

[1] Silicon Labs, SDS11847, Software Design Specification, Z-Wave Plus Device Type Specification.
[2] Silicon Labs, SDS11846, Software Design Specification, Z-Wave Plus Role Type Specification.
[3] Silicon Labs, INS1480, Instruction, Z-Wave 700 Getting Started for End Devices.
[4] Silicon Labs, ZAD13111, Z-Wave Alliance Document, Z-Wave Plus Assigned Icon Types.
[5] Silicon Labs, SDS11274, Software Design Specification, Security 2 Command Class.
[6] Silicon Labs, APL12956, Application Note, Z-Wave Association Basic.
[7] Silicon Labs, SDS13321, Software Design Specification, Supervision Command Class.
[8] Silicon Labs, APL13475, Application Note, Z-Wave Development Basics.
[9] Silicon Labs, SDS13826, Software Design Specification, Z-Wave Smart Start Requirements.
[10] Silicon Labs, SDS13968, Software Design Specification, Smart Start User Input Identifier Registry.
[11] Silicon Labs, SDS13781, Software Design Specification, Z-Wave Application Command Class Specification.
[12] Silicon Labs, SDS13782, Software Design Specification, Z-Wave Management Command Class Specification.
[13] Silicon Labs, SDS13783, Software Design Specification, Z-Wave Transport-Encapsulation Command Class Specification.
[14] Silicon Labs, SDS13784, Software Design Specification, Z-Wave Network-Protocol Command Class Specification.
[15] Silicon Labs, SDS14224, Software Design Specification, Z-Wave Plus v2 Device Type Specification.
[16] Silicon Labs, SDS14222, Software Design Specification, Association-Command-Class, Association Command Class, list of mandatory commands for the Lifeline Association Group.
[17] Silicon Labs, UG266, Gecko Bootloader User's Guide.
[18] Silicon Labs, AN1135, Using Third Generation NonVolatile Memory (NVM3) Data Storage.
[19] Silicon Labs, EFM32 Gecko Platform API Reference, Gecko Platform driver library, NVM3: http://devtools.silabs.com/dl/documentation/doxygen/5.7/efm32g/html/group__NVM3.html
[20] Silicon Labs, INS14280, Instruction for Z-Wave 700 Getting Started for End Devices.
[21] Silicon Labs, INS14278, How to Use Certified Apps in Z-Wave 700.
[22] Silicon Labs, AN0045, Application Note, USART/UART – Asynchronous mode