Instruction

Z-Wave Plus V2 Application Framework GSDK

Document No.:	INS14259
Version:	24
Description:	-
Written By:	PSH;COLSEN;NOBRIOT;GEKOCZIA;ANERDO;BSANTOK;TASTUMME; ALMUNKHA
Date:	
Reviewed By:	NTJ;COLSEN;SCBROWNI;CAOWENS;PSH;BSANTOK;GEKOCZIA;TASTUMME;MAMIH ALI
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	6.4	Added a True Status description in the Architecture section.
1	20181213	ESOSTERG	9	Updated Utilities section.
1	20181214	JESMILJA	8	Updated Command Classes description
2	20181217	JFR	6.1 9.4 10	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	6.3	Updated description of Power Management
4	20190315	JOROSEVA	7.2	Added description of NVM3 File System
4	20190315	JESMILJA	6.1 7	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	7.4	Updated description of ApplicationInit() and ApplicationTask()
4	20190320	JFR	0 9.4.3 7.2 All	Updated flashing of bootloader and app. Added ADC driver. Added how to adjust Tx power. Minor typos.
5	20190325	JFR	7.3	Added watchdog.
6	20190704	CHOLSEN	7.2	Minor text changes and fix of example code.
7	20190712	SCBROWNI	All	Minor typos
8	20190812	PESHORTY	6.2	Added description of ZAF_SetMaxInclusionRequestIntervals
8	20190829	PESHORTY	7.3	Added note about watchdog always being enabled in production code
8	20190829	CHOLSEN	9.6	Added section “Callback before entering sleep”
8	20190903	JFR	9.4.1.1	GPIO port usage in serial API application.
8	20190903	JOROSEVA	6.2.1	Learn mode status events.
8	20190903	SCBROWNI	6.2	Editorial review of Section 6.2
9	20191022	CHOLSEN JFR	9.4.2 4	Minor correction in source code. Added “Introduction to the Z-Wave Technology”.
10	20200518	JFR	10	Added location of OTA key.
11	20200611	SCBROWNI	All	Tech Pubs review of new sections since version 7
12	20201002	JFR	9.7 4.3.7	Timer usage in ISRs. Multicast with single case follow up mandatory.
13	20201123	JFR	All	Added Z-Wave Long Range
14	20201124	SCBROWNI	All 9.7 4.3.7	Reviewed new sections and corrected Header capitalization and figure title formatting
14	20210120	LAMICCON	9.4.2.1	Added UART communication settings for serial API application.
14	20210121	SAGHASEM LAMICCON	7.4 7.4.2 9.5.2 9.4.2	Added documentation on User Tasks (available APIs and limitations). Replaced ZW_ApplicationRegisterTask with ZW_UserTask_ApplicationRegisterTask. Removed parameter (1536/4) in function call ZW_UserTask_ApplicationRegisterTask example. Removed sentence “It is not recommended to add more threads to the application.” from section 9.4.2.
15	20210311	SAGHASEM JFR	6.1	Updated how to extract code size information as part of the build.

REVISION RECORD

Doc. Rev	Date	Author	Pages Affected	Brief Description of Changes
16	20210415	JFR	All	References changed to Z-Wave Alliance website.
16	20210609	PSH	10	Added information about building a bootloader for external flash
17	20220303	SRNARUMA	10	Added instructions for testing the bootloader on 700 and 800 series
18	20220524	CHOLSEN	7, 8, 9	Updated and removed description about changed/removed APIs.
19	20220725	MNPALANI	10, 10.2	Updated with references to gecko bootloader documentation for 700 and 800 series
20	20220808	JSMILJANIC	10	Add location of gbl sample files
21	20220808	MNPALANI	10	Updated Acronym and section 10 with clarification about pre-built bootloader
21	20220907	MNPALANI	10.2	Updated with information related to bootloader's compression algorithm and application version check feature for 800 series
22	20221011	CHOLSEN	2 7 7.4.1 8.6	Added reference to Doxygen output. Removed outdated description and added reference to Doxygen output.
22	20221201	CHOLSEN	9.1	Removed outdated content and added reference to 8.3
22	20221209	CHOLSEN	9.2 9.3	Removed outdated content and added reference to Doxygen output.
22	20221209	CHOLSEN	6	Added diagram showing ZAF's relation to other elements in Z-Wave. Removed other elements from the detailed diagram of ZAF architecture.
23	20221209	JSMILJANIC	8.7	Removed outdated content and added reference to Doxygen output.
23	20230130	CHOLSEN	7.4.1	Removed outdated content and added reference to Doxygen output.
24	20230711	BSANTOK	7 7.2	Removed mentions to deprecated modules: config_app, config_rf, Command Class Lists Configuration. Updated section 7.2.

Table of Contents

Table of Figures

1 Definitions, acronyms and abbreviations

Abbreviation	Explanation
AGI	Association Group Information
AL	Always Listening
APL	Application Layer
CC	Command Class
DLPDU	Data Link Protocol Data Unit
FL	Frequently Listening
GBL	Gecko Bootloader
ISM	(unlicensed) Industrial Scientific and Medical
ISR	Interrupt Service Routine
MAC	Medium Access Control
MPDU	MAC Protocol Data Unit
NIB	Network Information Base
NIF	Node Information Frame Command. Refer to [22]
NLDE	Network Layer Data Entity
NL	Non-Listening
NPDU	Network Layer Protocol Data Unit
NLME	Network Layer Management Entity
NSDU	Network Service Data Unit
NWE	Network Wide Exclusion
NWI	Network Wide Inclusion
NWK	Network Layer
NLDE-SAP	Network Layer Data Entity - Service Access Point
NLME-SAP	Network Layer Management Entity - Service Access Point
MLDE-SAP	MAC Layer Data Entity - Service Access Point
MLME-SAP	MAC Layer Management Entity - Service Access Point
OSI	Open System Interconnection
OTA	Over The Air
PLDE-SAP	Physical Layer Data Entity – Service Access Point
PLME-SAP	Physical Layer Management Entity – Service Access Point
PHY	Physical layer
S0	Security 0 Command Class
S2	Security 2 Command Class. Refer to [13]
S2 DSK	Security 2 Device Specific Key. Refer to [13]
SAP	Service Access Point
SAR	Segmentation and Reassembly
SDK	Software Development Kit
SIS	SUC ID Server
SUC	Static Update Controller
ZAF	The Z-Wave Plus Application Framework

2 Introduction

This document describes the Z-Wave Plus V2 Application Framework (ZAF) version 10.1x.x distributed on Z-Wave 700/800 SDK 7.1x.x.

Some content was moved to Doxygen, and this content will be shipped with the GSDK beginning with 7.19.0.

3 Purpose

The purpose of the ZAF is to facilitate the implementation of robust Z-Wave Plus V2 compliant products in a fast and cost-effective manner. The ZAF include 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 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. The Z-Wave logo is only granted to products passing certification.

4 Introduction to the Z-Wave Overview

The Z-Wave protocol is a low bandwidth half duplex protocol designed for reliable wireless communication in a low-cost control network. The main purpose of the protocol is to enable short message transportation in a reliable manner. The Z-Wave protocol is not designed to transfer a large amount of data or any kind of streaming or timing critical data.

4.1 The Z-Wave Protocol Stack Architecture

The Open System Interconnection (OSI) reference model is a representation system for characterizing and standardizing the functions of a communication system in terms of abstraction layers. This allows us to describe similar communication functionalities into logical layers. The 7 layers of the OSI model are regarded by many as an idealized model; too abstract and fine-grained for most real-world protocols. It is however useful to refer to the OSI model when describing a given communication protocol framework. With respect to that, the Z-Wave protocol stack would be described using the model as shown in Figure 1. Note that the Z-Wave application layer consists of the OSI stack layers knows as transport, session, presentation, and application.

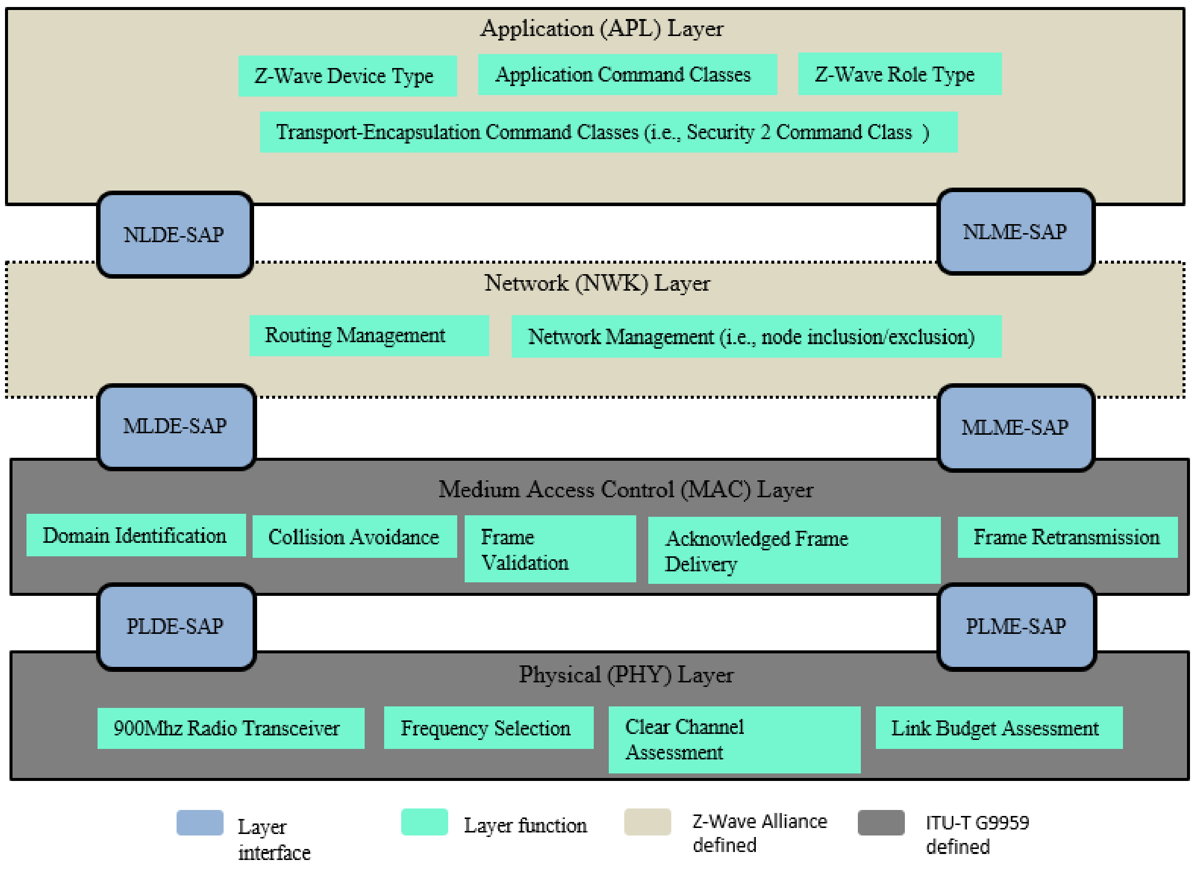


Figure 1. Z-Wave Protocol Stack Architecture

As depicted in Figure 1, the Z-Wave protocol stack is made up of OSI layers where each layer performs set of services for the upper layer. Each layer has two main interfaces to facilitate the communication with upper layers through a Service Access Point (SAP). The interfaces are described as a data entity and management entity that provide a data transmission service and all other services, respectively.

ITU-T G.9959 [25] defines the physical and medium access control layers.

- The physical layer offers a data flow control between the MAC and PHY layers and adds PHY-related management headers. The PHY layer is responsible for activation and deactivation of the radio transceiver, data transmission and reception, frequency selection, clear channel assessments, and the link budget assessment of received frames.
- The MAC layer defines the Z-Wave data transfer model and frame structure. During a Z-Wave frame transmission, the MAC layer takes the payload data from higher layers and construct the MAC data payload (MPDU) and the MPDU header. The header comprises addresses, frame control and frame length information. The frame control field is about 16 bits in length and contains information about the frame type and other control flags that can be used by higher layer.

On the foundation of those two lower layers, the Z-Wave alliance defines the Network layer (NWK) and application layers.

The Z-Wave Network Layer (NWK) defines a multi-hop routing protocol, that is employed by Z-Wave nodes to extend their communication range. It means that the Z-Wave nodes can therefore send frames to nodes that are not in direct radio communication range. Besides, the Z-Wave NWK layer is responsible for network formation (i.e., inclusion/exclusion of nodes to/from a network) and its maintenance. The Z-Wave NWK layer manages the network establishment using command frames known as the Z-Wave Protocol Command Class (Refer to [22] in section “Command frames”). These Z-Wave NWK commands are designed for network formation specific purposes.

The Z-Wave application layer is responsible for building applications using dedicated Command Classes, (defined in [11]-[14]).In order to be certifiable, applications **shall** comply with Z-Wave device types defined in [1] and [15]. Finally, the applications layer is also responsible for providing some network management functionalities using the NWK interface (for details, refer to [2]). All the Z-Wave specifications can be found on the Z-Wave Alliance website (Member login - Select ‘Specifications’ under the ‘Home’ tab).

4.2 Network Layer Reference Model

The Network Layer (NWK) provides an interface between the application layer and the MAC layer. The NWK layer relies on services provided by the MAC layer and offers services to higher layers though the Network Layer Data Entity (NLDE) and Network Layer Management Entity (NLME) service point interfaces. The NLME provides management service interface where the NWK layer management functionalities can be invoked. The NLME is responsible for maintaining a Network Information Base (NIB) that contains the routing information of the network. Figure 2 illustrates the components and interface of NWK layer.

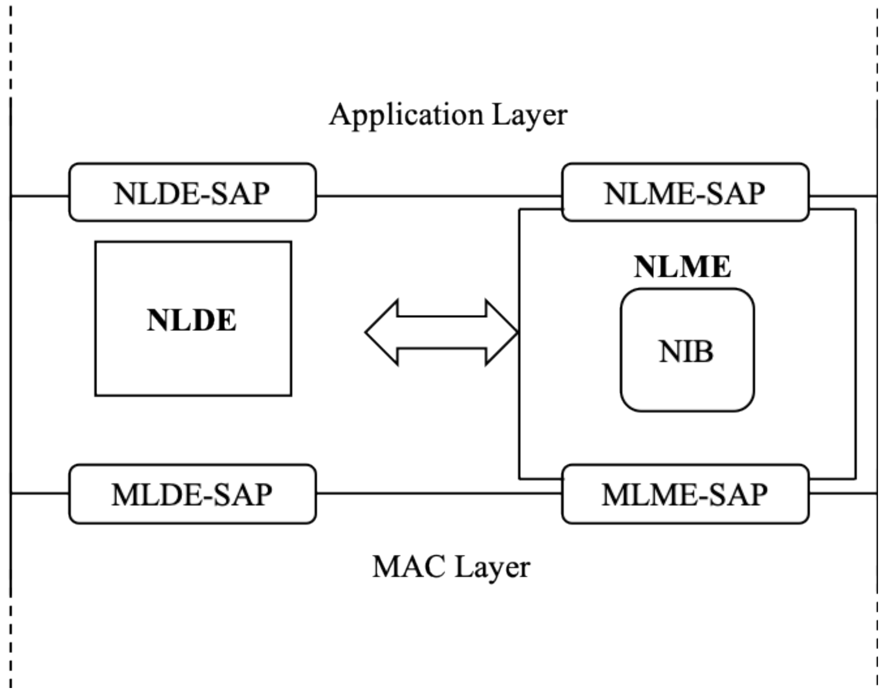


Figure 2. The Network Layer Reference Model

The Z-Wave NWK layer **shall** provide two services to the Application layer that are accessed through two SAPs:

- The data service, accessed through NLDE-SAP, and
- The network management service accessed through the NLME-SAP.

The detailed description of the Z-Wave NWK functional model is presented in [22], chapter "Z-WAVE NETWORK LAYER SPECIFICATION".

4.3 Z-Wave Definitions

4.3.1 Z-Wave Network Topology Basic Principles

The following is a summary of the basic Network topology principles established by ITU-T G.9959 [25]:

1. Groups of nodes are divided into domains:
    - The division of physical nodes into domains is logical. Domains may fully or partially overlap each other's radio frequency ranges.
    - The Z-Wave Network Layer supports up to 2^{32} domains.
    - Each domain is identified by a unique **HomeID**.
    - Management of different domains in the same physical media is handled by individual domain masters.
2. The domain is a set of nodes connected to the same medium:
    - One node in the domain operates as a domain master, known as the Primary Controller.
    - Each domain may contain up to *232* nodes (including the domain master).
    - Each node in the domain is identified by a **NodeID** that is unique within the actual domain.
    - Nodes of the same domain can communicate with each other either directly or via other nodes in the same domain.
3. Nodes of different ITU-T G.9959 [25] domains:
    - The Z-Wave Network Layer provides connectivity within one domain.
      In some cases, frames from a foreign domain are repeated into the current domain.
4. The network is self-healing:
    - Nodes may autonomously establish new routes on demand.
      Full mesh routing is supported. There is no requirement for star or tree network topologies.

4.3.2 Controller and End Devices

The Z-Wave network layer defines two networking node types: controller and end devices.

The controller nodes are responsible for setting up and maintaining the Z-Wave network. They can include or exclude nodes and they are aware for the network topology. This allows controllers to determine the possible routes between any two nodes in the network. Controllers can exchange network topology with each other.

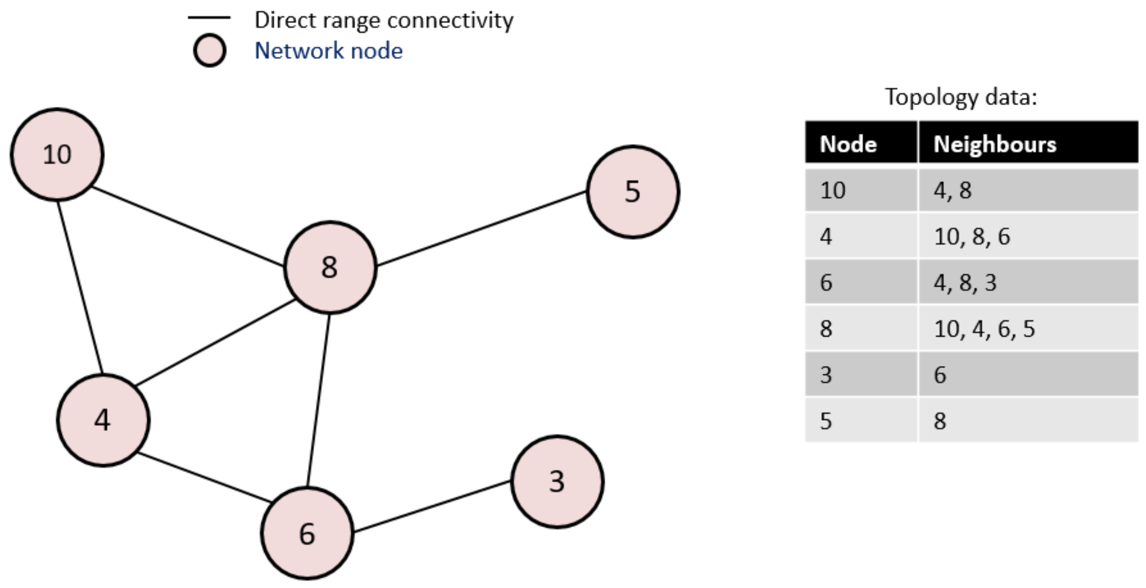
End devices can only be added or removed from a network by a controller, they do not calculate routes, and they simply rely on route information provided by the controllers. Note that end devices can send commands to other nodes and “control” other nodes functionalities at the application level.

Both controller and end devices can participate in mesh routing. It enables nodes within a network to communicate with each other even when they are out of direct communication range.

4.3.3 Network Topology

The network topology refers to the list of nodes present in a network as well the list of direct range neighbors for each node.

Figure 3 illustrates the concept of network topology.



Topology data:

Node	Neighbours
10	4, 8
4	10, 8, 6
6	4, 8, 3
8	10, 4, 6, 5
3	6
5	8

Figure 3. Network Topology Example

4.3.4 Z-Wave Controller Roles

A Z-Wave controller is a node that has the capability to provide network management functionalities such as adding/removing nodes to/from a network and distributing network topology to other controllers. The NWK layer defines several controller roles in a network:

4.3.5 Primary Controller

The Primary Controller is by default the controller that starts the network. The Primary Controller can always be used to set up and maintain a network. It can add/remove nodes and knows the network topology.

There can be only one Primary Controller in a network.

The Primary Controller may offer additional services to other controllers:

A Primary Controller can handover the Primary Controller (and/or SUC/SIS) role to another controller.

4.3.5.1 Secondary Controller

All controllers that are not the Primary Controller are Secondary Controllers.

If the Primary Controller does not have any SUC/SIS capability, a Secondary Controller cannot include nodes, exclude nodes, or receive updated network topology automatically.

If a SUC is present in the network, a Secondary Controller can request updated network topology at any time.

4.3.5.2 SUC Controller

A Static Update Controller (SUC) is the controller that has the responsibility to keep the network topology and distribute it to other controllers, on demand.

There can be only one SUC in a network. Both Primary Controllers and Secondary Controllers can be SUC.

4.3.5.3 SIS Controller

A SUC ID Server (SIS) Controller is a controller that has both the SUC Controller role and the Primary Controller role. In addition, it provides the SIS functionality. The SIS functionality consists in the ability to reserve NodeIDs to other controllers for enabling them to include and exclude nodes.

4.3.5.4 Inclusion Controllers

If the Primary Controller is the SIS, Secondary Controllers in the same network become Inclusion Controllers. Inclusion Controllers are secondary controllers that can include and exclude nodes on behalf of the SIS.

4.3.6 Node Operation Modes

Z-Wave nodes may operate in three different receiving modes.

4.3.6.1 Always Listening (AL)

AL nodes' RF receiver is always on and these nodes participate in mesh routing by repeating Routed NPDUs and Explore NPDUs. Refer to [22] in section "Routed NPDUs" and "Explore NPDUs".

4.3.6.2 Frequently Listening (FL)

FL nodes' RF receiver is turned off most of the time. The RF receiver is turned on at regular intervals for a short duration to listen the Wake Up Beams. AL nodes can reach FL nodes by issuing a Wake Up Beam prior to issuing commands to the FL nodes.

FL nodes do not participate in routing and do not repeat Routed NPDUs and Explore NPDUs. Refer to [22] in section "Routed NPDUs" and "Explore NPDUs".

ITU-T G.9959 [25] defines two possible Wake Up intervals FL nodes: 250ms and 1000ms, and three channel configurations.
When operating with a channel configuration 1 and 2, some NWK commands/frames indicate which setting to use (250ms or 1000ms).
When operating with channel configuration 3, the 1000ms setting is always used in frames/commands, despite using fragmented beams.

For more details, refer to [25].

4.3.6.3 Non-Listening (NL)

NL nodes cannot be awakened by another node. They wake up and transmit frames at fixed configured intervals to a single NodeID destination.

NL nodes reporting can be configured using the Wake Up Command Class. For more details, refer to [12].

4.3.7 Network Addressing

Z-Wave supports the following types of addressing:

- Singlecast
- Multicast
- Broadcast

The type of addressing and its frame format are defined in the MPDU Header (refer to [25]). 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 devices only. It is mandatory that multicast always use single cast follow up to ensure devices are reached out of direct range.

5 Z-Wave Long Range Protocol Overview

5.1 The Z-Wave Long Range Protocol Stack Architecture

The Z-Wave Long Range protocol stack is similar to the Z-Wave protocol stack. This is illustrated in Figure 4.

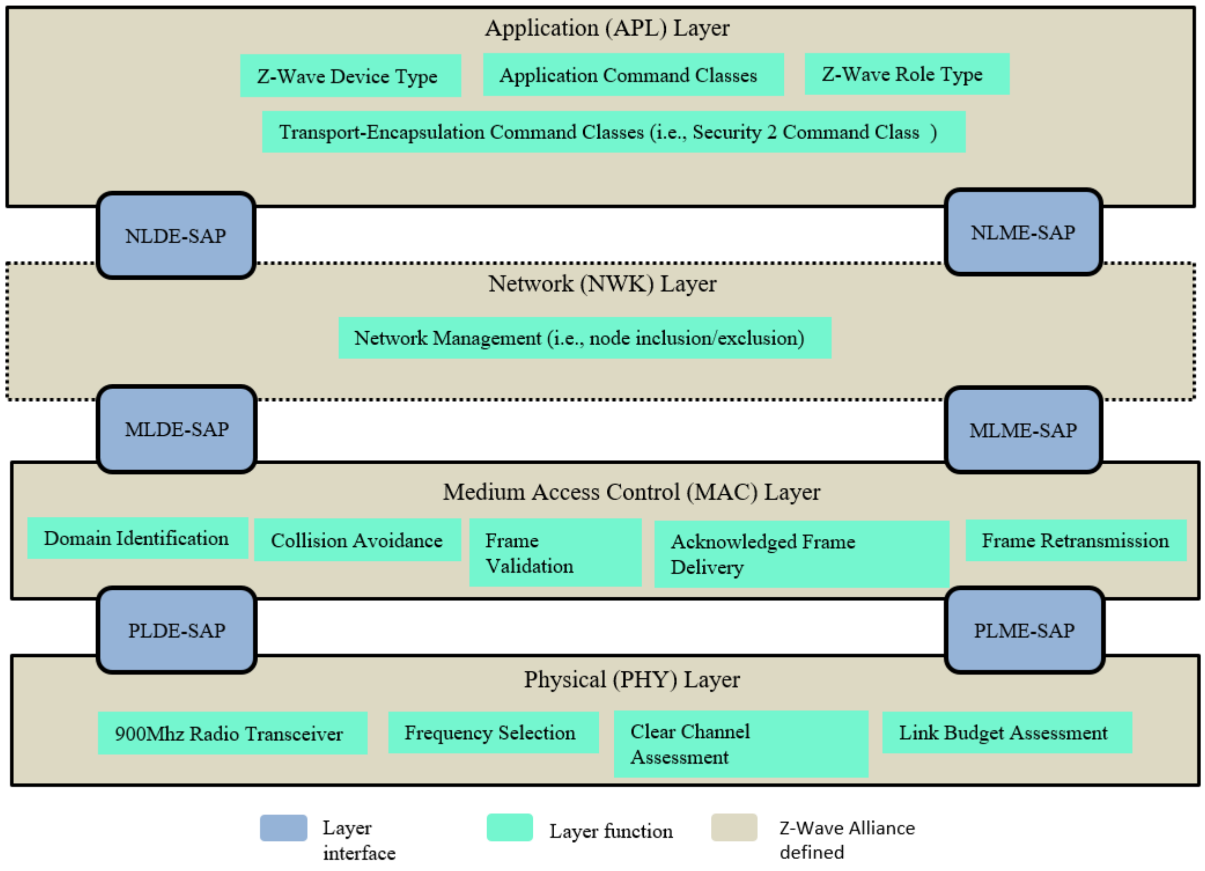


Figure 4. Z-Wave Long Range Protocol Stack Architecture

Each layer has two main interfaces to facilitate the communication with upper layers through an SAP. The interfaces are described as a data entity and management entity that provide a data transmission service and all other services, respectively.

“Z-Wave Long Range PHY layer specifications” [23] defines the physical layer and “Z-Wave Long Range MAC layer specifications” [24] defines the medium access control layer.

On the foundation of those two lower layers, the Z-Wave alliance defines the Network layer (NWK) and application layers.

The Z-Wave Long Range NWK layer is responsible for network formation (i.e., inclusion/exclusion of nodes to/from a network). The Z-Wave Long Range NWK layer manages the network establishment using command frames known as the Z-Wave Long Range Command Class (refer to [22] in section “Command frames”). These NWK commands are designed for network formation specific purposes.

The Z-Wave application layer is responsible for building applications using dedicated Command Classes, (defined in [11]-[14]).In order to be certifiable, applications **shall** comply with Z-Wave device types defined in [1] and [15]. Finally, the applications layer is also responsible for providing some network management functionalities using the NWK interface (for details, refer to [2]).

5.2 Z-Wave Long Range Network Layer Reference Model

The Z-Wave Long Range NWK layer provides an interface between the application layer and the MAC layer. The Z-Wave Long Range NWK layer relies on services provided by the MAC layer and offers services to higher layers though the Network Layer Data Entity (NLDE) and Network Layer Management Entity (NLME) service point interfaces. Figure 2 illustrates the components and interface of the Z-Wave Long Range NWK layer.

The Z-Wave Long Range NWK layer **shall** provide two services to the Application layer that are accessed through two SAPs:

- The data service, accessed through NLDE-SAP, and
- The network management service accessed through the NLME-SAP.

For a detailed description of the Z-Wave Long Range NWK functional model refer to [22] in section "Z-WAVE LONG RANGE NETWORK LAYER SPECIFICATION".

5.3 Z-Wave Long Range Definitions

5.3.1 Z-Wave Long Range Network Principles

The following is a summary of the network principles established by [23] and [24]:

1. Groups of nodes are divided into domains:
   - The division of physical nodes into domains is logical. Domains may fully or partially overlap each other's radio frequency ranges.
   - The Z-Wave Network Layer supports up to 2^{32} domains.
   - Each domain is identified by a unique **HomeID**.

2. The domain is a set of nodes connected to the same medium:
   - Each domain may contain up to *4000* nodes.
   - Each node in the domain is identified by a **NodeID** that is unique within the actual domain.
   - Nodes of the same domain can only communicate with the controller using direct range transmissions.

5.3.2 Controller and End Devices

Refer to 4.3.4 Z-Wave Controller Roles

5.3.3 Network Topology

Refer to 4.3.3 Network Topology

Nodes added to a network using Z-Wave Long Range will only have one known neighbor, which is the Primary Controller.

5.3.4 Z-Wave Controller Roles

Refer to 4.3.4 Z-Wave Controller Roles

A controller starting a Z-Wave Long Range network **shall** assume the Primary controller role.

The SUC/SIS functionalities will not be used in a Z-Wave Long Range network and included controllers will be Secondary Controllers.

5.3.5 Node operation modes

Refer to 4.3.6 Node Operation Modes

5.3.6 Network Addressing

Z-Wave Long Range supports the following type of addressing:

- Singlecast
- Broadcast

The type of addressing and its frame format are defined in the MPDU Header (refer to [24]).

6 Architecture

Figure 5 shows how the application and ZAF interfaces with other elements in Z-Wave.

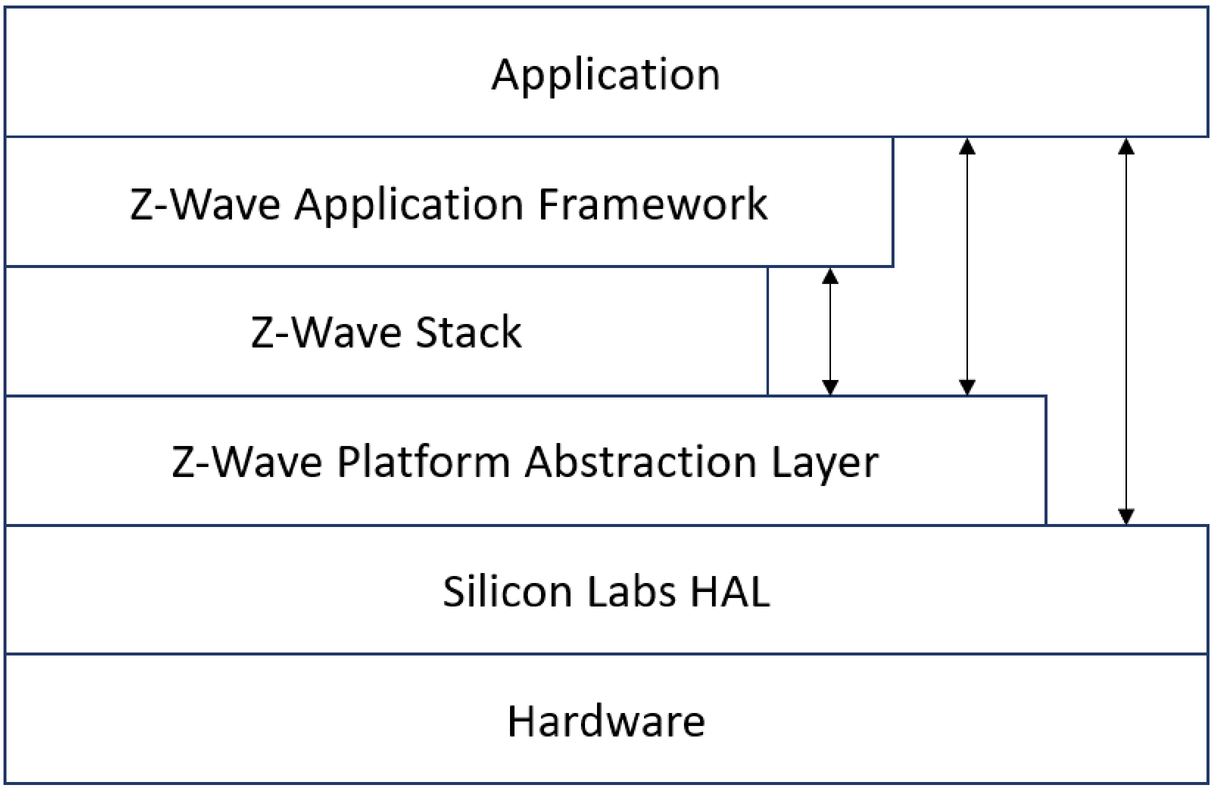

Application
Z-Wave Application Framework
Z-Wave Stack
Z-Wave Platform Abstraction Layer
Silicon Labs HAL
Hardware

Figure 5. Shows how the application and ZAF interfaces with other elements in Z-Wave.

Figure 6 shows the architecture of the ZAF.

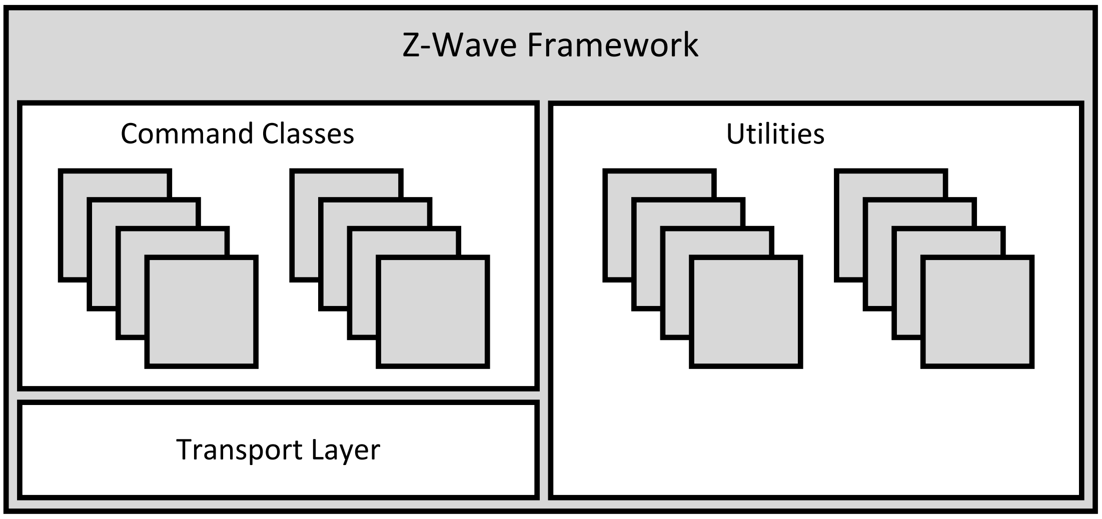


Figure 6. The Z-Wave Plus Application Framework Architecture

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.

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

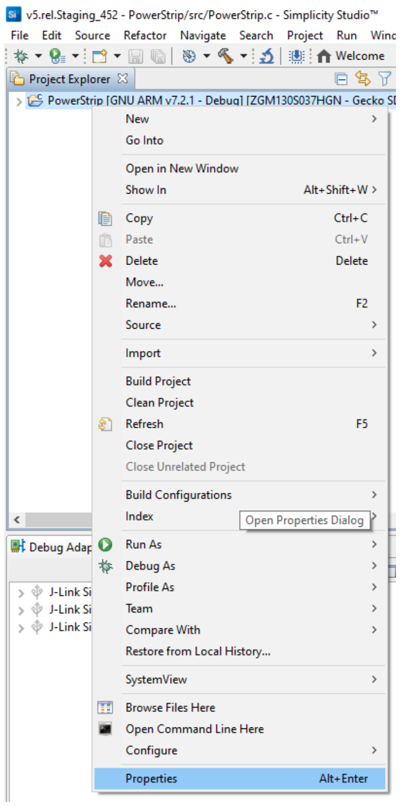
It’s possible to enable a feature that calculates and outputs the code size and non-volatile memory allocation as well as the SRAM memory usage as at the end of the build process.

As a prerequisite Python 3.x must be installed and available on the system path.

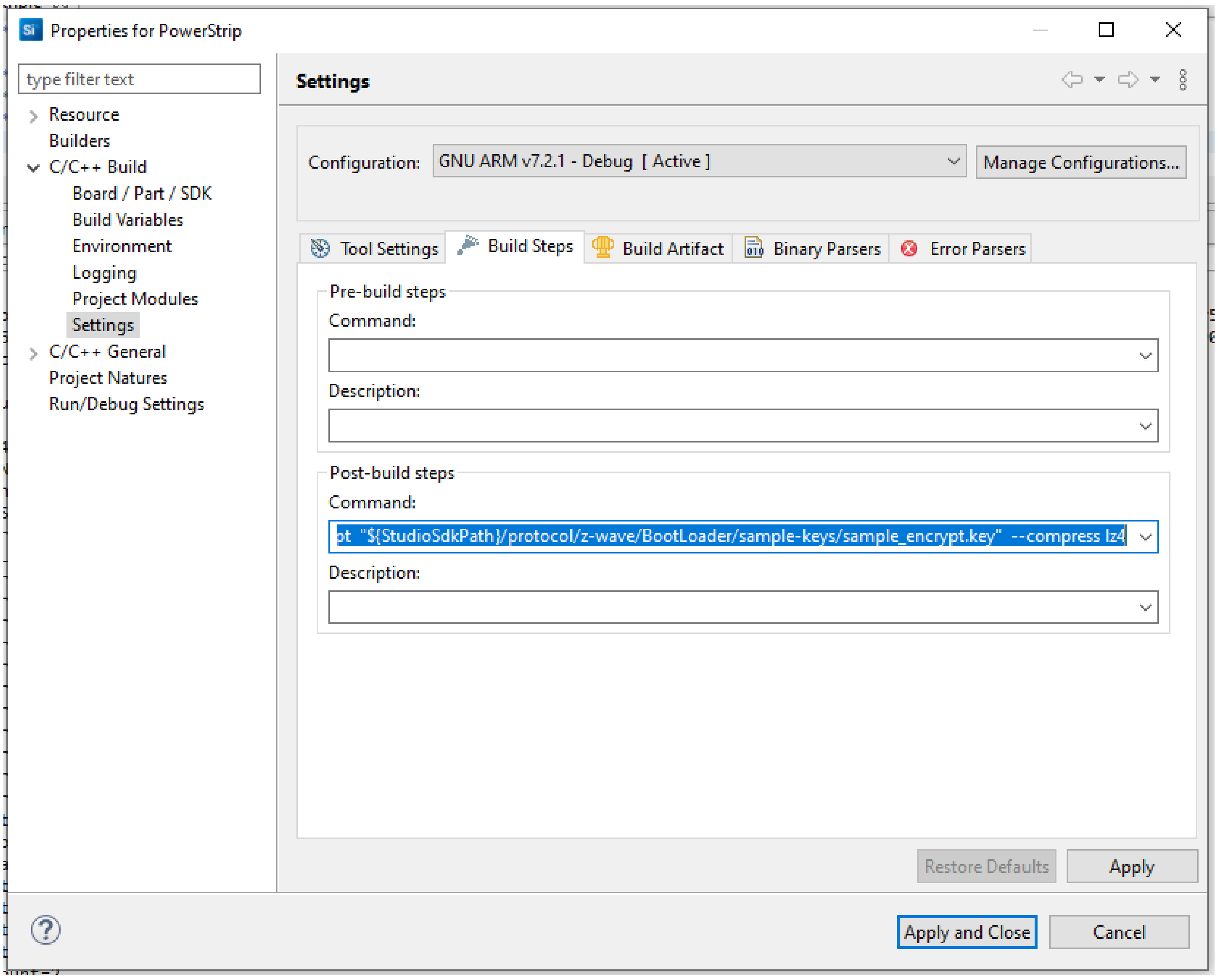
The changes made are only applied on a specific project. If multiple projects exist, then follow the steps for each project.

After having created a project, or already having a project do the following:

- Right-click on the project folder as in the picture, and select 'properties':



- The below dialog box appears:



Configuration: GNU ARM v7.2.1 - Debug [Active]

Post-build steps Command:

```
ot "${StudioSdkPath}/protocol/z-wave/BootLoader/sample-keys/sample_encrypt.key" --compress lz4
```

- Select C/C++ Build → Settings.
- And then ‘Build Steps’.

In this tab, one can run commands in a bash environment. As seen above, there are no pre-build commands, but we have a post-build command needed for signing the build binary.

Here we are going to append the following to the end of this already present command.

(remove ‘old commands’)

<old_commands> && arm-none-eabi-size -A "${BuildArtifactFileBaseName}.axf" > "${BuildArtifactFileBaseName}_codesize.txt" && python "${StudioSdkPath}/protocol/z-wave/Scripts/size_info_gen.py" -p -i "${BuildArtifactFileBaseName}_codesize.txt"

- The full string is ‘To enable’:

"${CommanderAdapterPackPath}" gbl create "${BuildArtifactFileBaseName}.gbl" --app "${BuildArtifactFileBaseName}.hex" --sign "${StudioSdkPath}/protocol/z-wave/BootLoader/sample-keys/sample_sign.key" --encrypt "${StudioSdkPath}/protocol/z-wave/BootLoader/sample-keys/sample_encrypt.key" --compress lz4 && arm-none-eabi-size -A "${BuildArtifactFileBaseName}.axf" > "${BuildArtifactFileBaseName}_codesize.txt" && python "${StudioSdkPath}/protocol/z-wave/Scripts/size_info_gen.py" -p -i "${BuildArtifactFileBaseName}_codesize.txt"

- The full string is 'To disable':

"${CommanderAdapterPackPath}" gbl create "${BuildArtifactFileBaseName}.gbl" --app "${BuildArtifactFileBaseName}.hex" --sign "${StudioSdkPath}/protocol/z-wave/BootLoader/sample-keys/sample_sign.key" --encrypt "${StudioSdkPath}/protocol/z-wave/BootLoader/sample-keys/sample_encrypt.key" --compress lz4

- Output:

The demo applications have a corresponding code size information generated for them and these files are available at the following location in the SDK:

C:\SiliconLabs\SimplicityStudio\vX\developer\sdks\gecko_sdk_suite\vX.X\protocol\z-wave\Apps\bin\codesize

The full path may differ on your system and the X's are version numbers.

6.2 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.

6.2.1 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.

6.2.2 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)

6.3 Power Management

Power management is handled by the Z-Wave protocol. All listening devices are prevented by the protocol from entering any energy mode other than EM0 and EM1. Listening Sleeping End Devices can enter energy modes EM0-EM2, while non-listening devices 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 Listening Sleeping End Device 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 Listening Sleeping End Devices and non-listening devices from entering other energy modes than EM0 and EM1. This means that the radio transceiver will stay operational.
PM_TYPE_PERIPHERAL	Prevents non-listening devices 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 9.6.

6.4 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.

6.4.1 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).

6.4.2 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:

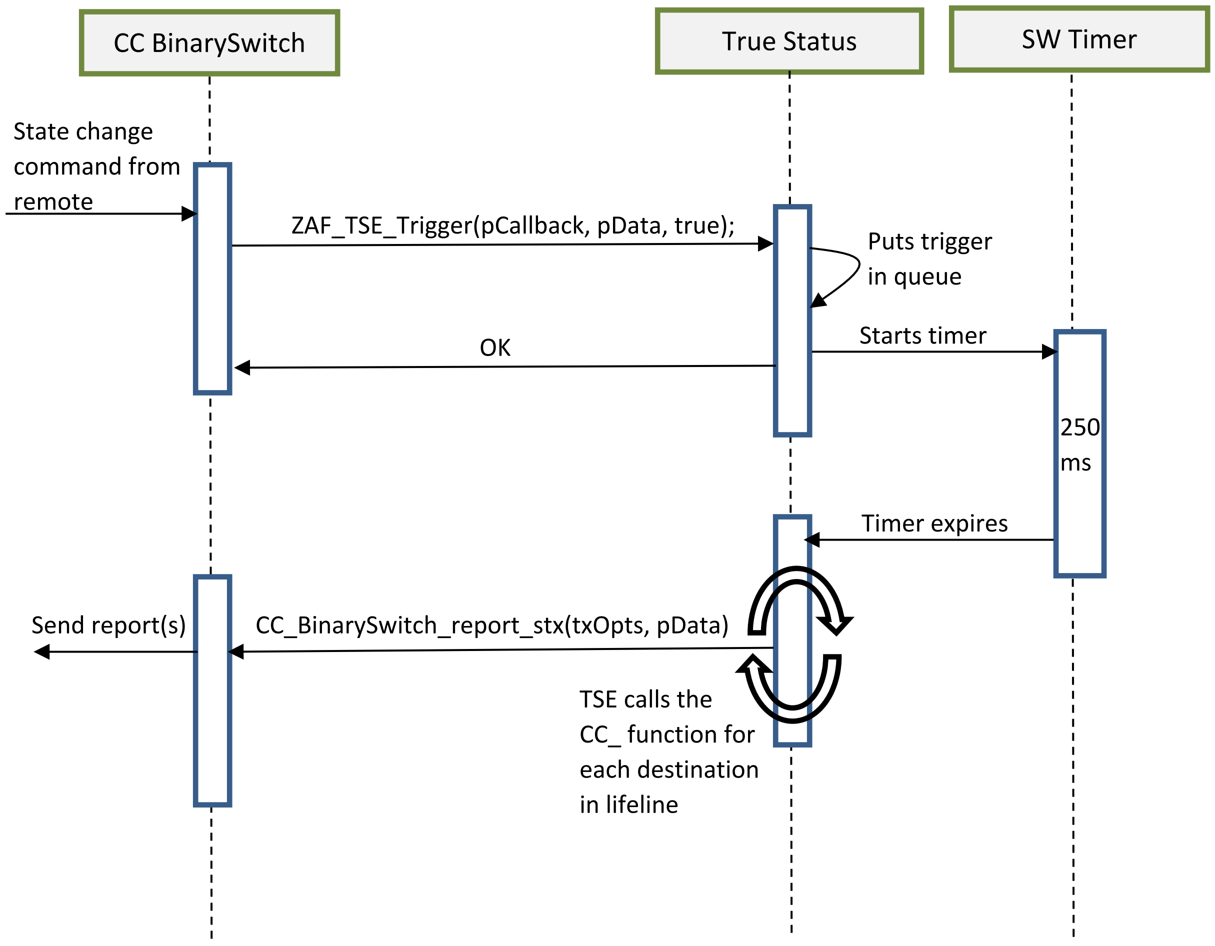
- **txOptions:** Tx Options passed from the True Status Engine with the required destination parameters for sending the state change report to a given lifeline member.
- **pData:** Pointer to the data struct previously registered at the call to ***ZAF_TSE_Trigger()***. Contains the command data for the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

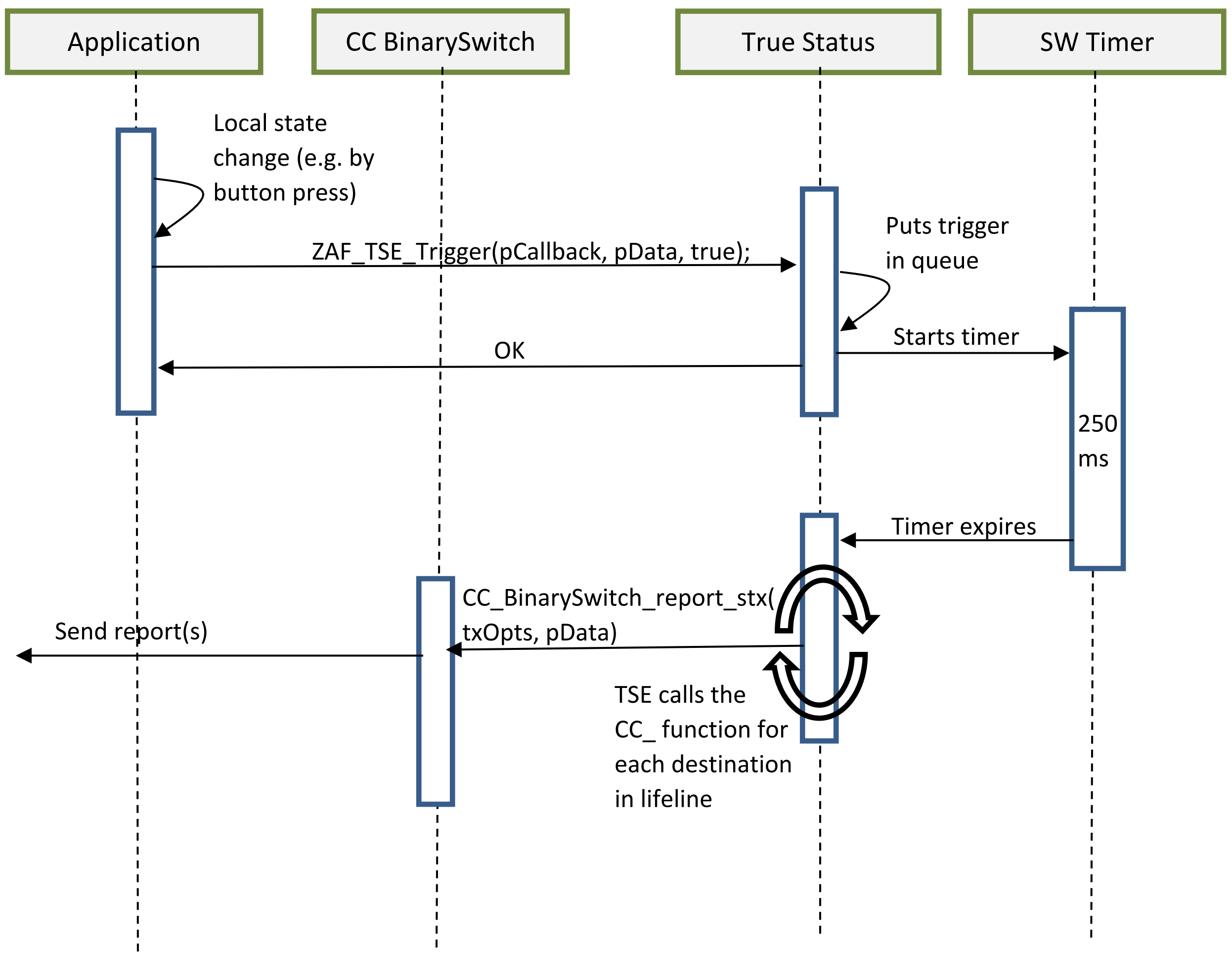
6.4.3 True Status Sequence Flows

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).

6.4.3.1 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

6.4.3.2 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
Starts timer
OK
250 ms
Timer expires
CC_BinarySwitch_report_stx(txOpts, pData)
Send report(s)
TSE calls the CC_ function for each destination in lifeline

7 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.

7.1 Create Application Folder and Set Up Build Environment

7.1.1 Select the Application to Start With

Pick one of existing software samples and modify it to match your needs. Refer to chapter 5 in [20] for more details. Selection should be made mainly based on Role Type:

- Always On End Device: SwitchOnOff, WallController, PowerStrip
- Reporting Sleeping End Device: SensorPIR
- Listening Sleeping End Device: DoorLock

In addition, there are some additional options to consider:

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

7.1.2 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 [3].
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.

7.2 Setting Up Files in Non-Volatile Memory

An 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 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 zpal_nvm.c must be increased accordingly, or the file system will slow down considerably.

Each file in an NVM3 file system is assigned an individual 20-bit object key as file identifier [18]. Application specific files should use identifiers in the range 0x00000—0x0FFFF, so the effective identifier address range is reduced to 16 bits.

File identifiers used by the ZAF are listed in ZAF_file_ids.h. These are guaranteed not to clash with the Application specific files, as the Z-Wave Platform Abstraction layer assigns these files a handle in a range that is different from the Application specific files (the exact range varies based on the actual hardware used – see zpal_nvm.c for more details).
All application file systems must have a file with file identifier ZAF_FILE_ID_APP_VERSION that contains the version number of the application. On startup, the application checks if this file is present. If it is missing, the file system is considered unwritten or corrupt, leading to reformatting of the file system and all files being reset to their default state.

To introduce an application-specific file to the file system, just define a new 16-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 ZAF_nvm_app_write() [19].

```
#define FILE_ID_MYNEWFILE 0x0112

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.
ZAF_nvm_app_write(FILE_ID_MYNEWFILE, &wrtData, sizeof(wrtData));

SMyAppData rdData;

/*
 * This call will read the content of the file to rdData. Reading only part of a file
 * is also possible, using the function ZAF_nvm_app_read_object_part().
 */
ZAF_nvm_app_read(FILE_ID_MYNEWFILE, &rdData, sizeof(rdData));
```

It is recommended that one write default content to application-specific files in the function zaf_nvm_app_set_default_configuration(), that gets called when a device is excluded or included into a new network.

When performing a 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 application's SLC project (.slcp) file. The bootloader does not accept files having a lower or equal version number. Then, on the first startup 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. The function zaf_nvm_app_load_configuration_migration() can then be used to perform the necessary migration steps.

7.3 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
```

7.4 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_UserTask_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 infinite loop where it lets the event distributor handle any events. See the description in section 9.5. The FreeRTOS scheduler will ensure that the application task is given processor time according to its priority.

7.4.1 Endpoint Configuration

Please see "When and how to create a Multi Channel application" in the Doxygen output.

7.4.2 Multi-threaded applications

A Z-Wave application runs by default the following threads:

- The Z-Wave protocol thread for the execution of the protocol stack
- The main application thread for the execution of the application code, and as an interface to the Z-Wave protocol thread.

An application can be extended creating additional FreeRTOS threads\tasks. These threads are called User Tasks to differentiate them from the main application thread (responsible for the communication with the Z-Wave protocol thread).

The User Tasks are created using the ZW_UserTask module (*ZW_UserTask.h*), which contains the definition of the ZW_UserTask_t data type and the available APIs to create a task.

ZW_UserTask_t is defined as follows:

```
typedef struct {
  TaskFunction_t         pTaskFunc;
  char*                  pTaskName;
  void*                  pUserTaskParam;
  ZW_UserTask_Priority_t priority;
  ZW_UserTask_Buffer_t*  taskBuffer;
} ZW_UserTask_t;
```

Where

- pTaskFunc is the function executed by the task
- pTaskName is the name of the task
- pUserTaskParam is the pointer to the parameter passed to the task function
- priority is the priority level of the task. The available priorities are:
  - USERTASK_PRIORITY_BACKGROUND
  - USERTASK_PRIORITY_NORMAL
  - USERTASK_PRIORITY_HIGHEST
- taskBuffer is the statically allocated memory required by FreeRTOS to handle the task.

The following APIs are available:

```
ReturnCode_t ZW_UserTask_Init(void);
```

- **ZW_UserTask_Init()** function is used to initialize the ZW_UserTask module. It does not need to be explicitly called as it is already called at boot time.
  - If this function is called again, it will return Code_Fail_InvalidState.

```
ReturnCode_t ZW_UserTask_CreateTask(ZW_UserTask_t* task, TaskHandle_t* xHandle);
```

- ZW_UserTask_CreateTask() can be used to create additional tasks.
  - This function requires the following input parameters:
    - task is the pointer to the ZW_UserTask task parameters.

        - xHandle is the returned task handle by FreeRTOS.
    - This function can return the following codes:
        - Code_Fail_InvalidState – if the module is not yet initialized.
        - Code_Fail_InvalidOperation – if the maximum of user tasks is already created.
        - Code_Fail_InvalidParameter – if the priority setting is out of bounds or any of the input parameters are NULL.
        - Code_Fail_Unknown – if FreeRTOS fails to create the task.

```
bool ZW_UserTask_ApplicationRegisterTask(
                                  VOID_CALLBACKFUNC(appTaskFunc)(SApplicationHandles*),
                                  uint8_t iZwRxQueueTaskNotificationBitNumber,
                                  uint8_t iZwCommandStatusQueueTaskNotificationBitNumber,
                                  const SProtocolConfig_t * pProtocolConfig
);
```

- ZW_UserTask_ApplicationRegisterTask() is used to create the main application thread. For additional details on the function usage and behavior see Sections 7.4 and 9.5.2.

For additional information on the APIs, consult the *ZW_UserTask.h* file. For detailed information on the available return codes see *ReturnCode_t* defined in *ZW_global_definitions.h*.

An example of how to use these APIs is available in the SensorPIR application (see ApplicationInit(…), in *SensorPIR.c*).

7.4.2.1 Limitations and recommendations

The following limitations and recommendations apply to the User Tasks:

- A maximum of three User Tasks can be created in addition to the main application thread.
- The User Task priorities are allocated such that they are all below the priority of the Z-Wave protocol thread (this is enforced within the ZW_UserTask module, where the available priority levels are defined).
- User Tasks must be created (using function ZW_UserTask_CreateTask) before the FreeRTOS scheduler starts, therefore, their creation is only possible within ApplicationInit(…) (see Section 7.4).
- The communication between the main application thread and the User Tasks is not provided. However, the use of FreeRTOS queues and events is recommended.
- The User Tasks cannot communicate directly with the Z-Wave protocol thread. They need to use the main application thread as intermediary.
    - With the exception of the power manager APIs that can be called from all tasks.

8 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.

8.1 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

8.1.1 Unsolicited Transmission

Migrated to Doxygen: “How to implement a new command class”.

8.1.2 Respond to Received Command

Each CC implementation defines a handler, e.g. CC_ManufacturerSpecific_handler(). This function extracts the received frame for a given Command Class. The function must be registered using a REGISTER_CC macro, preferably the latest. Normally, the frame is carrying a “Set” or “Get” Command that results in a function call for reading or writing data. For some command classes, these commands are handled by the command class itself, but for others, it is up to the application to implement these functions provided as external functions in the CC header file.

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;
}
```

8.1.3 Reporting the version of a command class

Migrated to Doxygen: “How to implement a new command class”.

8.1.4 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 6.4.

8.2 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.

8.3 Association Group Information CC (AGI CC)

The implementation of AGI CC serves two purposes:

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

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

Refer to *Z-Wave Command Class Configurator* in the Doxygen output for a description on how the AGI CC is configured.

8.4 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.

8.5 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.

8.6 Notification CC Version 8

This information was moved to the Doxygen output.

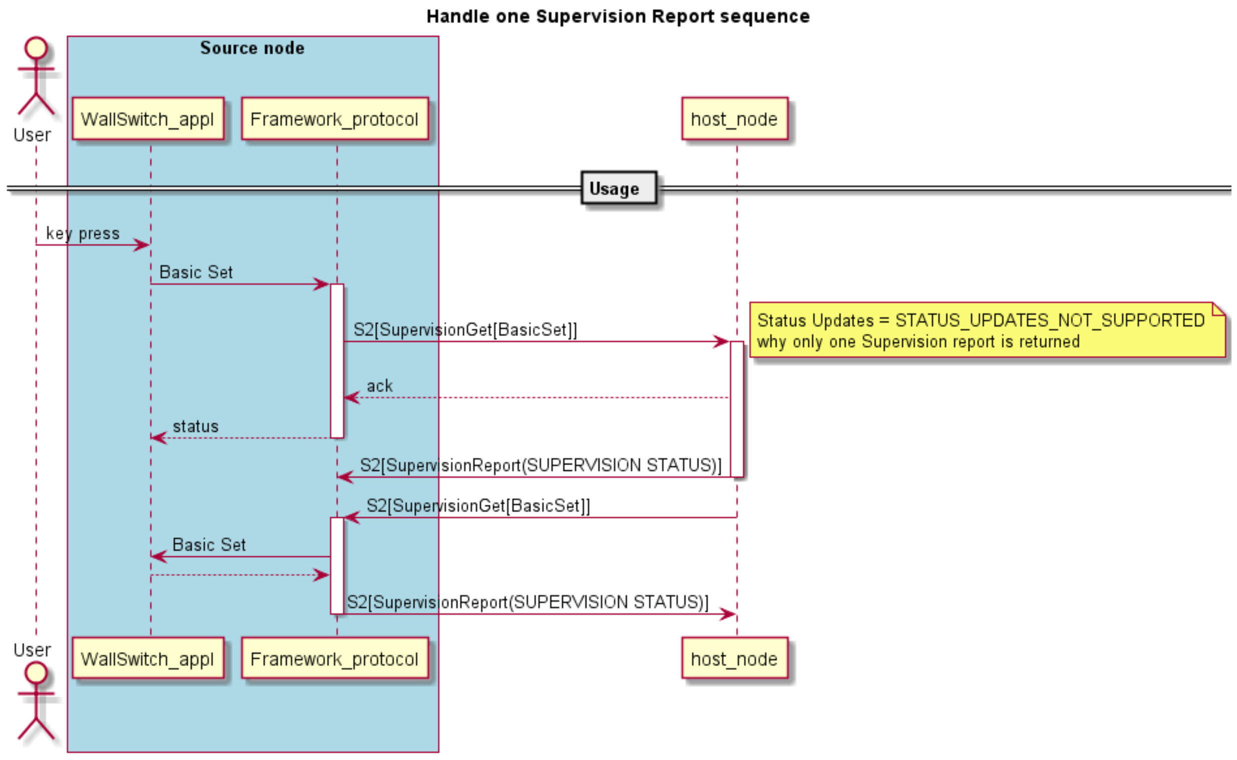
8.7 Supervision CC

This information was moved to the Doxygen output.

8.7.1 Configuration Scenarios

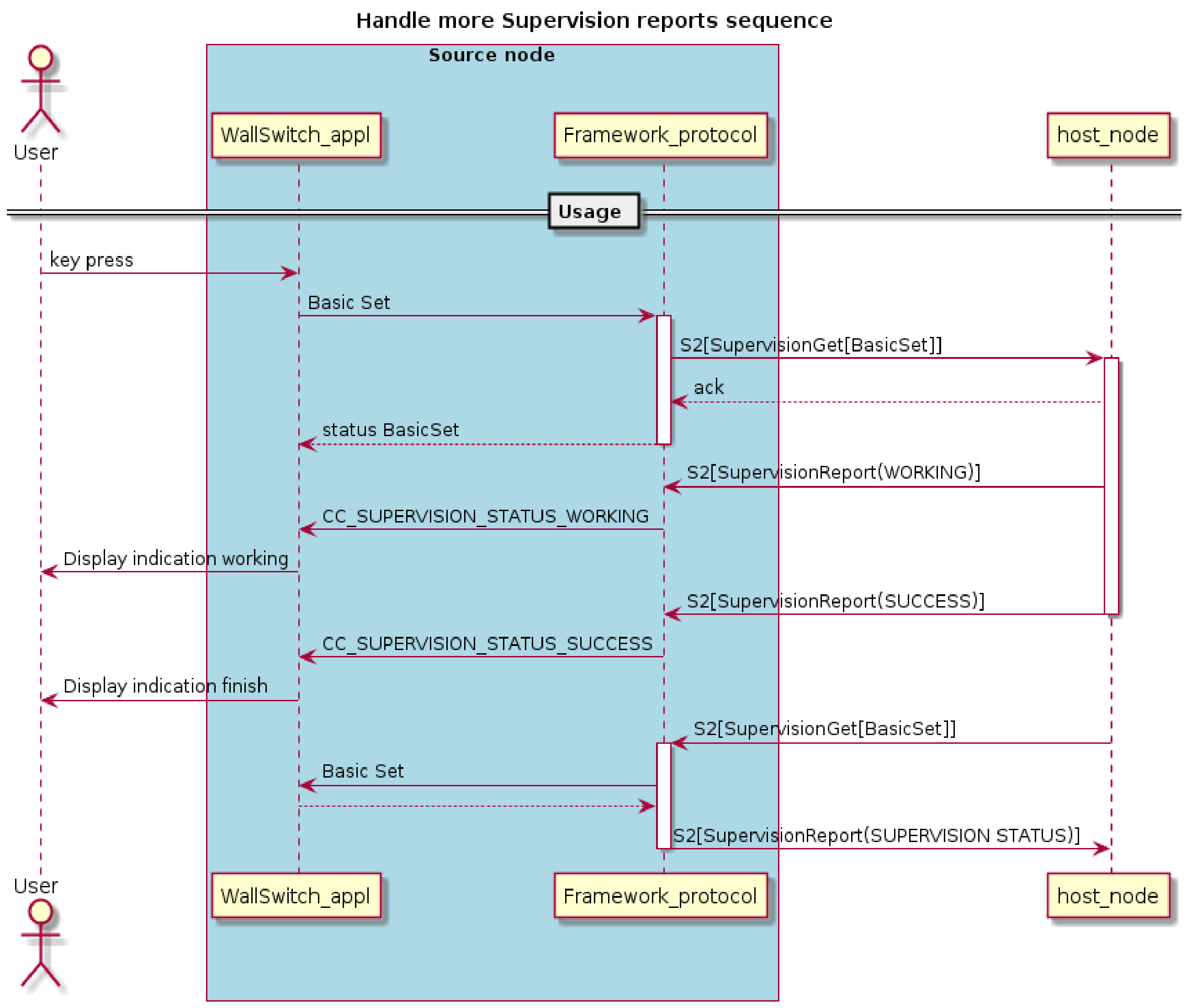
8.7.1.1 Default Configuration

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.


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

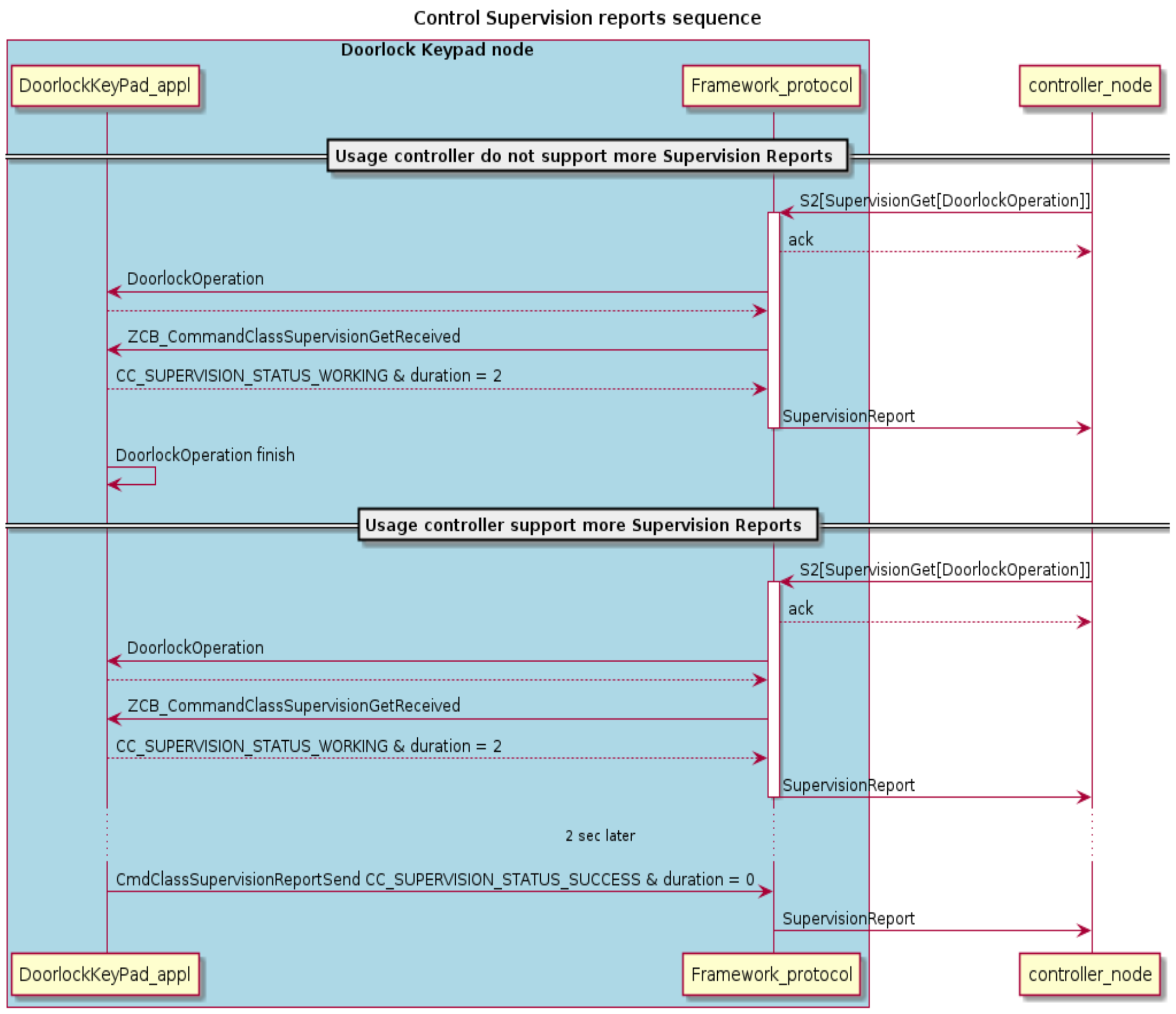
8.7.1.2 Handle More Supervision Reports

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).


Handle more Supervision reports sequence
Source node
User
WallSwitch_appl
Framework_protocol
host_node
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)]

8.7.1.3 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.


Control Supervision reports sequence
Doorlock Keypad node
DoorlockKeyPad_appl
Framework_protocol
controller_node
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

9 Utilities

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

9.1 AGI Module

The AGI module is part of CC AGI. Please see section 8.3.

9.2 Association Module

The Association Module is a common module to CC Association and CC Multi Channel Association. Please see the doxygen output.

9.3 Interfacing Firmware Update Module "ota_util"

ota_util is a part of Command Class Firmware Update Meta Data. Please see the Doxygen output for information about how to use this command class.

9.4 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 9.4.1). 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

9.4.1 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);
```

9.4.1.1 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*

9.4.2 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 [21]).

The event handler is called in the application thread context, which ensures thread safety in the application.

9.4.2.1 UART communication settings

The UART interface of the serial API application uses the following settings:

Table 1. Serial API UART (RS-232) Settings

Parameter	Value
Baud rate	115200 bits/s
Parity	No
Data bits	8
Stop bits	1

The least significant bit (LSB) of each byte MUST be transmitted first on the physical wire.

9.4.3 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.

9.5 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.

9.5.1 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.

9.5.2 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_UserTask_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_UserTask_ApplicationRegisterTask(ApplicationTask,
                                      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);
```

9.5.3 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);
  }
}
```

9.5.4 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);
```

9.5.5 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);
  }
  :
```

9.5.6 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);
  }
}
```

9.6 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 (Listening Sleeping End Device) and SensorPIR (Reporting Sleeping End Device) both use power locks. SensorPIR goes all the way to EM4 when idle, but since a Listening Sleeping End Device must wake up every 250 ms or 1000 ms, the DoorLockKeyPad application only sleeps in EM2 to achieve a fast wakeup time.

9.7 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 9.5.1 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.

Be careful when using timers inside Interrupt Service Routines (ISR). As TimerStartISR uses xTimerChangePeriodFromISR (freeRTOS) https://www.freertos.org/FreeRTOS-timers-xTimerChangePeriodFromISR.html, it is not advisable to start more than ONE timer from ISR with TimerStartISR. Finally, use only the dedicated ISR functions in the ISR.

10 Firmware Update Images and Bootloader

This section is split into two parts - for the 700 and 800 series. Due to changes in silicon not all steps are the same between the two series. Further the guide assumes that the user is running the latest version of Simplicity Studio (which at the time of writing the guide is V5).

Note: Detailed information regarding bootloaders for the 700 and 800 series is available in https://www.silabs.com/documents/public/user-guides/ug103-06-fundamentals-bootloading.pdf

https://www.silabs.com/documents/public/user-guides/ug489-gecko-bootloader-user-guide-gsdk-4.pdf

Before going further into this guide, it is important to know a few files and locations that are crucial in following the subsequent steps.

Default Simplicity Studio Installation Location : C:\SiliconLabs\SimplicityStudio\v5

Default SDK Installation Location : C:\Users\{Username}\SimplicityStudio\SDKs

Default Workspace Location: C:\Users\{Username}\SimplicityStudio\v5_workspace

Sample encryption keys - (sample_sign.key, sample_sign.key.pub, sample_sign.key-tokens.txt, sample_encrypt.key)

OTA gbl file 255 version - for purposes of this exercise, let it be the SwitchOnOff file. (SwitchOnOff_brd4204d_REGION_EU_v255.gbl).
Sample gbl files are located in {SDK Installation Location}\gecko_sdk_4.1.0\protocol\z-wave\Apps\bin\gbl

Bootloader image files: ota-EFR32ZG23_BRD4204D-crc.s37

sample keys location : {SDK Installation Location}\gecko_sdk\protocol\z-wave\BootLoader\sample-keys

bootloader location : {Simplicity Studio Installation Location}\offline\com.silabs.sdk.stack.super_4.0.1\protocol\z-wave\Apps\bin

Commander utility location : {Simplicity Studio Installation Location}\developer\adapter_packs\commander

Sample project location : {Default Workspace Location}\SwitchOnOff\GNU ARM v10.2.1 – Default

Note: The above paths are customizable by the user and as such must be adapted if they are different from the default paths given above.

10.1 For the 700 series

The purpose of this section is to describe how to generate and manage firmware update images. The SDK provides two bootloaders for a given board type – OTA and OTW. 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 Bootloader(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].

Generate GBL 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.

Flashing the Bootloader and App

It is possible to flash the bootloader 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.

10.2 For the 800 series

800 series is different from the 700 series. The process for preparing the eval board to test OTA is easier on the 800 series.

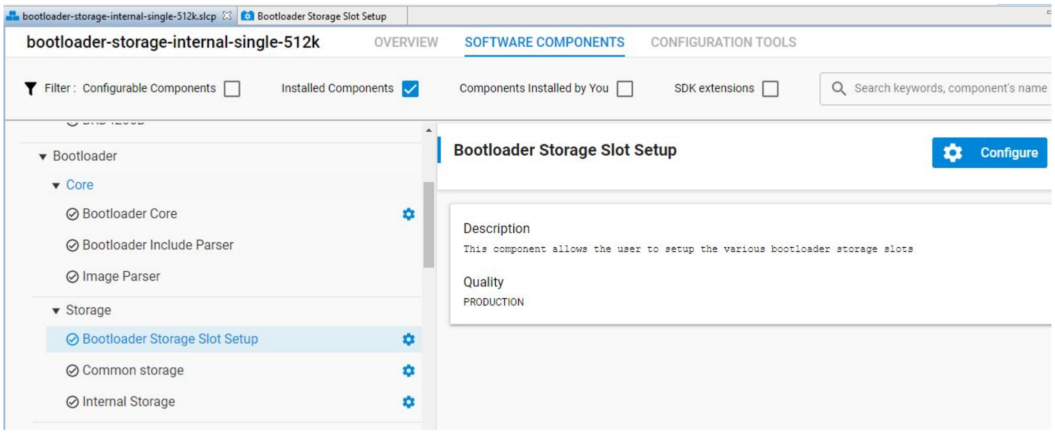
1. Download the bootloader image files. This is done by running the sample demo in Simplicity Studio. This will download the bootloader images to disk.
2. Create an example project using the same project above as a template.
3. Build the project in SimplicityStudio and generate the hex files.
4. Erase device
 "commander.exe device masserase -s {jlink_serial}"
5. Reset device
"commander.exe device reset -s {jlink_serial} "
6. Flash the appropriate OTA bootloader image – for purposes of this exercise, let it be eval board BRD4204.
"commander.exe flash {bootloader location}\ota-EFR32ZG23_BRD4204D-crc.s37 -s {jlink_serial}"
7. Flash initial device firmware built in step 3
"commander.exe flash "{sample project location}\SwitchOnOff.hex" --address 0x0 -s {jlink_serial}"
8. Flash the encryption keys.
"commander.exe flash --tokengroup znet --tokenfile sample_encrypt.key --tokenfile sample_sign.key-tokens.txt -s {jlink_serial}"

9. Reset device
"commander.exe device reset -s {jlink_serial}"
10. Connect a controller or a device running a controller firmware to the PC and start the PC controller application.
11. Include the node into the network and make sure the device is visible.
12. In the PC controller application initiate the OTA update using the OTA gbl file mentioned in the prerequisites.

Notes on Bootloader configuration for the 800 series

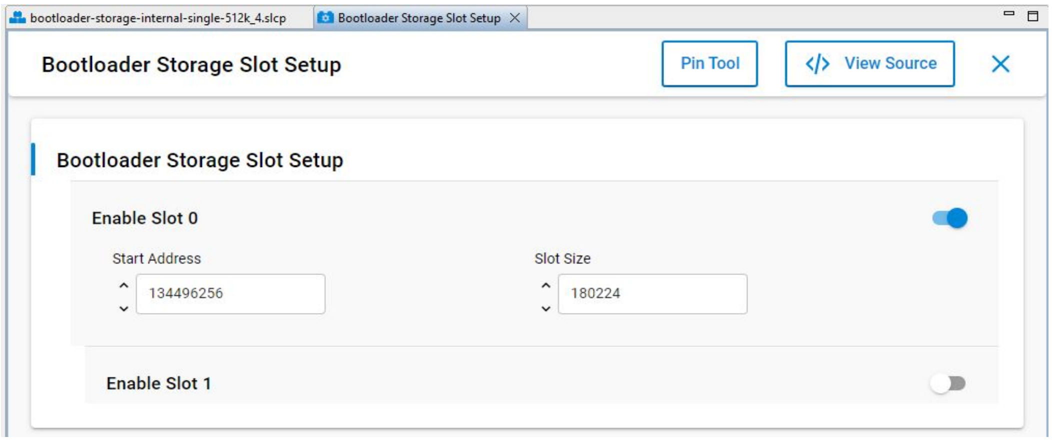
The bootloader resides at the start address 0x08000000 of the main flash and a fixed space of 24KB is reserved for this. Z-Wave applications will start from address 0x08006000.

The bootloader must be flashed first before the Z-Wave sample application is flashed. It is also possible to combine the bootloader and the Z-Wave application into a single image. *One can use the pre-built bootloader images available in Simplicity Studio or build a bootloader image by themselves using the bootloader sample applications in Simplicity Studio*. When building the bootloader, the OTA image storage information must be configured according to Figure 7 and Figure 8. Importantly, **this storage slot (slot0), start address and size must not be changed.**


bootloader-storage-internal-single-512k.slcp
Bootloader Storage Slot Setup
bootloader-storage-internal-single-512k
OVERVIEW
SOFTWARE COMPONENTS
CONFIGURATION TOOLS
Filter : Configurable Components
Installed Components
Components Installed by You
SDK extensions
Search keywords, component's name
Bootloader
Core
Bootloader Core
Bootloader Include Parser
Image Parser
Storage
Bootloader Storage Slot Setup
Common storage
Internal Storage
Bootloader Storage Slot Setup
Configure

Description
This component allows the user to setup the various bootloader storage slots
Quality
PRODUCTION

Figure 7 Bootloader Storage Slot Component - Configuration


bootloader-storage-internal-single-512k_4.slcp
Bootloader Storage Slot Setup
Bootloader Storage Slot Setup
Pin Tool
View Source
Bootloader Storage Slot Setup

Enable Slot 0: enabled
Start Address: 134496256
Slot Size: 180224
Enable Slot 1: disabled

Figure 8 Z-Wave OTA image storage information

The postbuild.sh script available in the Simplicity Studio project for Z-Wave sample applications can be used as an example to combine the bootloader and the Z-Wave application into a single image. This script also contains reference example for generating the OTA images.

Linker script for 800 series

The Z-Wave sample applications that are available in Simplicity Studio contain linker scripts that have been tuned to accommodate the OTA image related configuration also. It is a recommendation to not modify this linker script when developing applications.

Bootloader compression for 800 series

The bootloader compression type used for OTA is ***lzma*** compression. The corresponding component name to be selected in studio is ***bootloader_compression_lzma.***

Application upgrade version

The studio component ***bootloader_app_upgrade_version*** has to be selected for checking the application version during upgrades.

10.3 Building a custom bootloader

The bootloader provided with the SDK is built for using the internal flash for storing the downloaded OTA image. This design ensures the lowest possible HW cost of a product but also reduce the code space available for the application. In some products it might be desirable to have more code space and that can be achieved by having the OTA image in an external Flash. If this option is used, an alternative bootloader must be built and programmed onto the product.

The steps to use external flash are:

1. Use simplicity Studio to create a OTA bootloader for the external Flash you want to use. For details about how to configure the bootloader see [27]
    a. Make sure that the configuration fits the configuration in protocol\z-wave\ZAF\CommandClasses\FirmwareUpdate\ota_util.c
2. Modify linker script
    a. Copy the default linker script in protocol/z-wave/ZWave/linkerscripts/zgm13-zw700.ld to your local application project and set studio up to use that file for linking in Properties -> C/C++ Build -> Settings -> Memory Layout
3. Edit the new linker file
    a. Extend the FLASH section in the linker script from 232K up to a maximum of 264K to get more code space for the application.
4. Compile the application
    a. Remember to follow the steps in the previous chapters to generate a valid firmware update file.

10.4 Erase and read MFG frequency

There is a dedicated space in the flash memory where the Manufacturing Tokens datas can be stored. This area can be write once during firmware running. To save the new region the flash must be erased before.

Read token frequency: ”commander tokendump --tokengroup znet --token MFG_ZWAVE_COUNTRY_FREQ”

Write token frequency: ”commander flash --tokengroup znet --token MFG_ZWAVE_COUNTRY_FREQ:0xFF”

0xFF mean, this area is erased.

References

[1] Z-Wave Alliance, Software Design Specification, Z-Wave Plus Device Type Specification.
[2] Z-Wave Alliance, Software Design Specification, Z-Wave Plus Role Type Specification.
[3] Silicon Labs, INS14280, 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] Z-Wave Alliance, Software Design Specification, Z-Wave Application Command Class Specification.
[12] Z-Wave Alliance, Software Design Specification, Z-Wave Management Command Class Specification.
[13] Z-Wave Alliance, Software Design Specification, Z-Wave Transport-Encapsulation Command Class Specification.
[14] Z-Wave Alliance, Software Design Specification, Z-Wave Network-Protocol Command Class Specification.
[15] Z-Wave Alliance, Software Design Specification, Z-Wave Plus v2 Device Type Specification.
[16] Z-Wave Alliance, 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, INS14278, How to Use Certified Apps in Z-Wave 700.
[21] Silicon Labs, AN0045, Application Note, USART/UART – Asynchronous mode.
[22] Z-Wave Alliance, CSWG, Z-Wave and Z-Wave Long Range Network Specification.
[23] Z-Wave Alliance, CSWG, Z-Wave Long Range PHY layer specifications.
[24] Z-Wave Alliance, CSWG, Z-Wave Long Range MAC layer specifications.
[25] ITU-T G.9959, Short range narrowband digital radiocommunication transceivers – PHY & MAC layer specifications.
[26] Z-Wave Alliance, Node Provisioning Information Type Registry.
[27] UG266: Silicon Labs Gecko BootLoader User Guide
[28] Z-Wave Lock Bits and User Data Page Contents