Instruction

Working in 500 Series Environment User Guide

Document No.:	INS12366
Version:	22
Description:	Describes the 500 Series software development environment with respect to tools, installation and build in both uVision and makefile system. Application related matters are also described such as bootloader, OTA firmware update, interrupt service routines
Written By:	JFR;EFH;JSI;PSH;COLSEN;JBU;BBR
Date:	2018-03-06
Reviewed By:	JSI;PSH;SSE;NTJ;BBR;JFR;CRASMUSSEN
Restrictions:	Partners Only

Approved by:

Date	CET	Initials	Name	Justification
2018-03-06	14:02:07	NTJ	Niels Thybo Johansen	

SILICON LABS

CONFIDENTIAL

REVISION RECORD

Doc. Ver.	Date	By	Pages affected	Brief description of changes
1	20130517	JFR EFH	ALL	Initial draft
2	20130613	JFR EFH TRO JBU	ALL	Major revision
2	20130930	JSI	Section 4.3 & 4.4 Section 4.2	Added Bootloader description Added application interrupt service routines
3	20131016	JFR	Section 4.2.2 Section 4.5.1 Section 3.4	Updated application interrupt service routines description for makefiles Added external NVM application data Extended makefile parameter description
3	20131125	JSI	Section 4.3 & 4.4	Updated with regard to added OTA support for 256KB and NVM Added Figure 7
4	20131127	TRO	Section 4.4.3	Add description of OtaInit(..)
5	20140123	JSI	Section 4.2.1 Section 3.4	Updated IV address for uVision project LIB16 parameter added
6	20140130	JSI JFR EFH	Section 4.5.1 Section 3.2 Table 2 Section 4.6.2 Section 4.2.1	Added description of NVM_LIB_SIZE Removed KEIL_LOCAL_PATH Clarified targets Removed internal R&D note Removed IV(0x1800) insertion description (now automatically)
7	20140312	JFR	Section 4.2	EA global enable usage
8	20140404	PSH	Section 4.2	Removed unnecessary makefile step in interrupt handling
9	20150211	JFR EFH	Section 3.4 Section 4.5.2 Section 3.4	Added description of sticky make targets Added section 4.5.2 , External NVM Application data layout and location LIB16 parameter removed
10	20150517	JFR	Section 3.4.11	Removed generation of an uVision multi-project file
11	20160120	JFR	Section 3.1 Section 4.1 Section 4.5	Updated Keil PK51 Professional Developers Kit version Updated porting requirements Updated Z-Wave Plus application implementation
12	20160421	JFR	Section 3.1	Updated to Keil PK51 9.54A.
13	20160525	COLSEN	Section 3.4.10	Added TEST_INTERFACE parameter
13	20160530	TRO	Section 4.3	Change NVM decription for SDK 6.61/SDK 6.7x
13	20160801	TRO	Section 4.4.4	Firmware update SDK 6.51.x application to SDK with S2 support
13	20160816	TRO	Section 4.3	Bootloader description
13	20160817	EFH	Section 4.5.3	Added description of NVM data initialization
13	20160817	TRO	Section 4.4.4	Bullet points phase 2: add porting possibilities. Change Figure 11
14	20160919	JBU	Section 4.4.1 & 4.4.2	Described how to transfer OTZ files via Firmware Update Meta Data Command Class
15	20170112	EFH	Section 4.5.2	Added clarification of rules for adding new NVM variables
16	20170208	TRO	Section 4.4.3, 4.4.4, 4.4.4.1 and 4.4.4.2	Added intermediate description for Makefile and application source code
17	20170301	TRO	Section 4.4.4	Update section after review
17	20170328	TRO	Section 3.4.1	Update BOOT parameter for BOOTLOADER_UPDATE
18	20170619	PSH	Section 3.4.5 Section 4.5.1	Removed MY frequency and clarified that frequency parameter is required. Specified that 16KB EEPROM will not work on controllers
19	20170817	EFH	Section 3.4.12 and 3.4.13	Added description of make parameter options UVISIONPREBUILD and UVISIONPOSTBUILD
20	20171116	JSI	Section 4.5.2.1	Added description of SDK 6.6x+ application external NVM usage definition
20	20171116	JFR	Section 4.5.1	Removed NVM_LIB_SIZE = 0x6000 and 16KB EEPROM option as external NVM
21	20180228	JFR	Section 3.4.5	Clarified missing parameters
22	20180306	BBR	All	Added Silicon Labs template

Table of Contents

Table of Figures

List of Tables

1 ABBREVIATIONS

Abbreviation	Explanation
GNU	An organization devoted to the creation and support of Open Source software
IMA	Installation and Maintenance Application
NVM	Non-volatile Memory
OTA	Over The Air
SDK	Z-Wave Software Developer's Kit
WUT	Wake Up Timer

2 INTRODUCTION

2.1 Purpose

The purpose of this document is to guide the Z-Wave application programmer through the very first Z-Wave software system build. This programming guide describes how to build a complete program and load it on a 500 Series Z-Wave module. Refer to [1], [2] or [3] depending on SDK used regarding Z-Wave Plus applications hosted on the 500 Series Z-Wave module.

2.2 Audience and prerequisites

The audience is R&D software application programmers. The programmer should be familiar with the Keil PK51 Professional Developers Kit for the 8051 microcontroller and the GNU make utility.

3 DEVELOPMENT ENVIRONMENT SETUP AND EXECUTION

The developer can choose from two methods to develop, build, and download firmware to the Z-Wave 500 Series single chips:

- Use Keil uVision4 Integrated Development Environment
- Use a source code editor, make command line tool, and the Z-Wave Programmer GUI tool

3.1 3rd party tools to SDK

There is an additional 3rd party software tool that is required to develop Z-Wave applications that is not supplied with the SDK. That is the Keil PK51 Professional Developers Kit v9.54A for the 8051 microcontroller.

Notice that PK51 Professional Developers Kit v9.54A must be used due to bugs identified when using banking etc.

The Keil PK51 Professional Developers Kit v9.54A can be purchased through Digi-Key Corporation **www.digikey.com** as our Z-Wave SDK. Alternative distributors visit **www.keil.com** for details.

In the following it is assumed that PK51 Professional Developers Kit v9.54A is installed in the folder **C:\KEIL\C51**.

3.2 Environment Setup

A couple of environment variables must be defined before the sample applications can be built on the Z-Wave SDK:

- KEILPATH
- TOOLSDIR

The procedure on a Windows PC is performed as follows:

1. Select **Start**, **Control Panel** and **System**
2. Windows XP: Select **Advanced** tab and click the **Environment Variables** button
3. Windows 7: Open **Advanced system settings**, select **Advanced** tab and click **Environment Variables** button

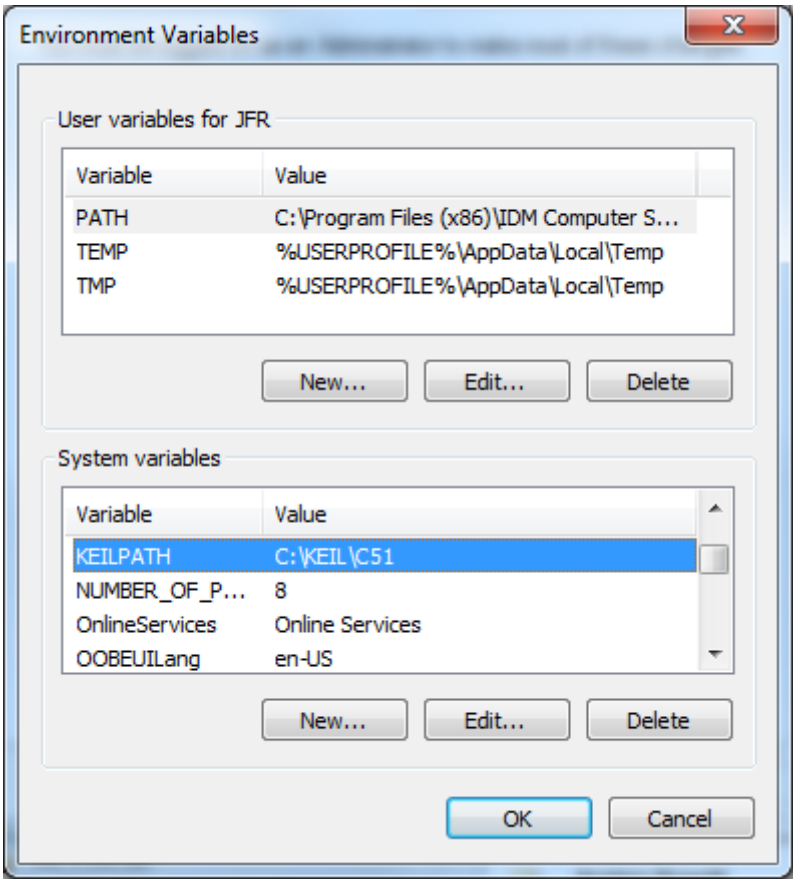

Figure 1, Configuring KEILPATH environment variables

4. Under **System variables** activate the **New** button
5. In the **Variable name** textbox enter **KEILPATH** (use capital letters because Windows is case sensitive)
6. In the **Variable value** textbox enter **C:\KEIL\C51** and activate the **OK** button

7. Under **System variables** activate the **New** button

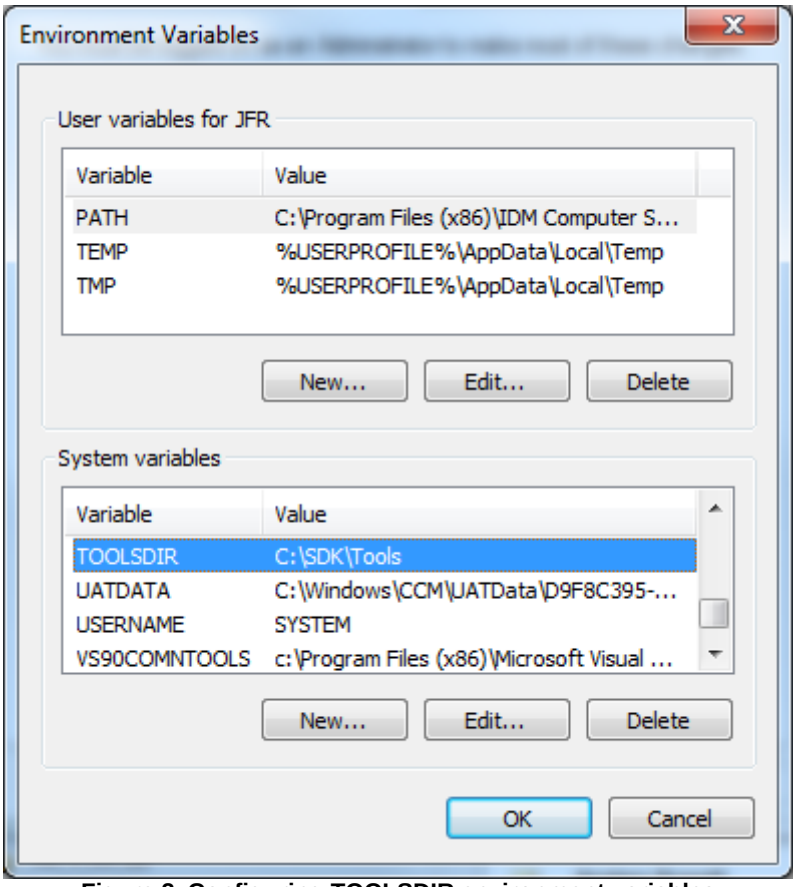

Figure 2, Configuring TOOLSDIR environment variables

8. In the **Variable name** textbox enter **TOOLSDIR** (use capital letters because Windows is case sensitive)
9. In the **Variable value** textbox enter **C:\SDK\TOOLS** and activate the **OK** button

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

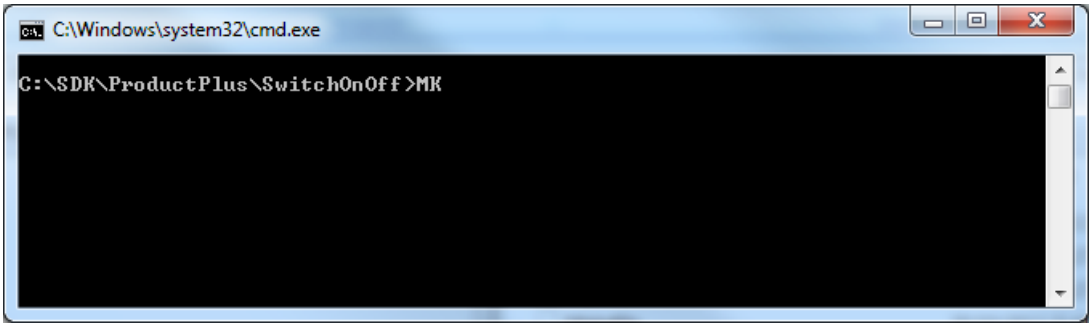

Figure 3, Building sample applications

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

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

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

3.3 Compiling from the Command Line

The command line batch file, MK.BAT, can build all versions with respect to device types (Portable Controller, Static Controller, Bridge Controller, Enhanced 232 Slave, Routing Slave, etc.) and RF frequencies (ANZ/EU/HK/IL/IN/JP/KR/MY/RU/US) at once, or the wanted target can also be entered as a parameter on the command line. The figure below displays the possible targets for a given product.

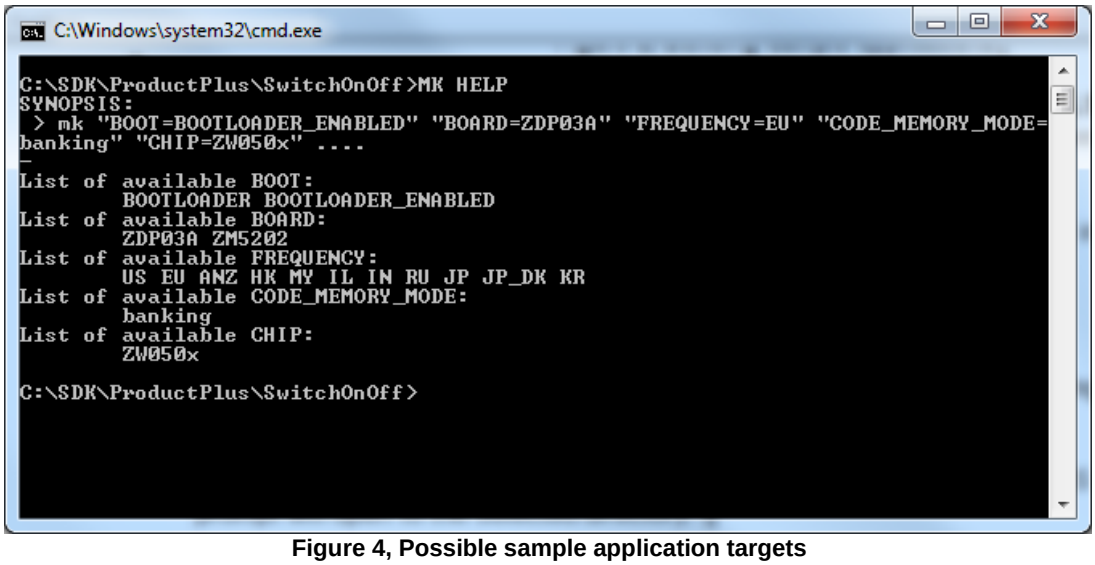

Figure 4, Possible sample application targets

Remember to enter the targets in upper and lower case as shown.

For every parameter, you can specify a single variant to build for in three different ways:

- By specifying the frequency in your command line, like:
  > mk “FREQUENCY=EU” ....

- By setting the parameter in the Makefile (it is prepared):
  FREQUENCY:=EU

- Alternatively you can do the same by setting your environment from the command line with:
  > SET FREQUENCY=EU

Remember to UNSET this when you jump to work on other things.

You can combine these methods in any way for the different parameters.

When MK.BAT is executed the following directory structure is created within the source code directory as depicted below:

```
- <appl>
    - build
        - <appl>_ZW050x_<freq. etc.>          - SD3502/ZDB3502 and ZM5101/ZDB5101 module
            - list                            - contains list files
            - rels                            - contains object files and map files
            <appl>_ZW050x_<freq. etc.>.hex    - application hex file
```

3.4 Makefile project

The file Makefile is initially read by the make tools that are called from mk.bat. It creates the directory structure and defines the build-targets and then calls the other makefiles in the build depending on the target.

Every sample application has a main Makefile describing what can be built. It also gives the developer an opportunity to limit what is built to a subset of this.

Targets can be built for lots of variants defined by the following parameters:

- BOOT
- BOARD
- CHIP
- CODE_MEMORY_MODE
- FREQUENCY
- HOST_INTERFACE
- IMA
- LIBRARY
- SENSOR_TYPE
- UVISION
- WATCHDOG

Not all of these parameters are relevant for all applications, but the irrelevant ones are set to a default selected value in the applications Makefile.

For every one of these parameters, there are three different ways to set which one you want. The three ways to do this is described in the Makefile for the application. You can leave parameters unspecified. Then make will build targets for all combinations of these parameters.

The applications main Makefile defines a list of modules, which are specific for the application, and which shall be included in the build.

The applications main Makefile also defines CDEFINES, which are specific for the application.

In a clean product directory, if you do not fully specify all parameters for the single target, you want to build, the make command will only give you a short help text listing what you can build, and an example command line fully specifying a single target. Copy and paste this example line and correct it to match your preferred single target.

Next time you rebuild your single target, you only need to execute the make command "MK" without any parameters. Then your single target will be rebuilt.

If you want to build a group of targets, then leave selected parameters unspecified, and add to the make command a pseudo target "ZW0x0x". In this way you can build all possible targets in just one command (without ending up in the help text).

In the process of building your single preferred target, a Makefile.sticky will be made in your project directory. This makefile is the one, which controls which target is your preferred one. If you decide for another target, you can delete the Makefile.sticky or just make a "MK clean". Or you can simply fully specify another single target. Then this one will be built, and it will be your preferred target onwards.

3.4.1 BOOT parameter

This parameter is used to build an application with/without a bootloader included when supporting OTA firmware update. The bootloader is used to copy firmware image from external NVM to internal code space.

Table 1. Description of BOOT parameters in command line

Parameter	Result
No parameters specified	Building all variants of the application and bootloader: * without a bootloader, * bootloader, * application with a bootloader and * application without a bootloader but taking into account that it already resides in FLASH memory.
BOOT=nonBOOT	Building application without a bootloader.
BOOT=BOOTLOADER	Building bootloader, which is a pre-requisite for BOOTLOADER_ENABLED.
BOOT=BOOTLOADER_ENABLED	Building application with a bootloader and the same application without a bootloader but taking into account that it already resides in FLASH memory. The application without a bootloader is used to make an OTA firmware update.
BOOT=BOOTLOADER_UPDATE	(Only SDK 6.61.01) Build intermediate application for updating SDK 6.61.01 target to SDK 6.71x (see 4.4.4).

3.4.2 BOARD parameter

This parameter specifies the target hardware platform when building the application.

Table 2. Description of BOARD parameters in command line

Parameter	Result
No parameters specified	Building for all hardware platforms.
BOARD=ZDP03A	Building for a ZDB3502 mounted on a ZDP03A
BOARD=ZM5101	Building for a ZDB5101 mounted on a ZDP03A
BOARD=ZM5202	Building for a ZDB5202 mounted on a ZDP03A

3.4.3 CHIP parameter

This parameter specifies the chip used. However, only the 500 Series chip is supported currently.

Table 3. Description of CHIP parameters in command line

Parameter	Result
No parameters specified	Building application for 500 Series chip
CHIP=ZW050x	Building application for 500 Series chip

3.4.4 CODE_MEMORY_MODE parameter

This parameter specifies the code memory layout used. This parameter supports only banking.

Table 4. Description of CODE_MEMORY_MODE parameters in command line

Parameter	Result
No parameters specified	Building application for a banked environment.
CODE_MEMORY_MODE=banking	Building application for a banked environment.

3.4.5 FREQUENCY parameter

This parameter specifies the RF frequency to be built. Some of the selections cover additional countries.

Table 5. Description of FREQUENCY parameters in command line

Parameter	Result
No parameters specified	Prompt frequency input in case the pseudo target “ZW0x0x” is not specified.
FREQUENCY=ANZ	Building targets using Australia and New Zealand frequency
FREQUENCY=EU	Building targets using European Union frequency
FREQUENCY=HK	Building targets using Hong Kong frequency
FREQUENCY=IL	Building targets using Israel frequency
FREQUENCY=IN	Building targets using India frequency
FREQUENCY=JP	Building targets using Japan frequency
FREQUENCY=JP_DK	Building targets using Japan frequency for testing only. This variant uses a lower “Listen Before Talk” threshold to allow testing in Denmark.
FREQUENCY=KR	Building targets using Korea frequency
FREQUENCY=RU	Building targets using Russia frequency
FREQUENCY=US	Building targets using United States frequency

3.4.6 HOST_INTERFACE parameter

This parameter specifies the serial API communication interface to be built. This parameter is only supported by the serial API applications.

Table 6. Description of HOST_INTERFACE parameters in command line

Parameter	Result
No parameters specified	Building application for all serial API communication interfaces.
HOST_INTERFACE=UART	Building application for an UART based serial API communication interfaces.
HOST_INTERFACE=USBVCP	Building application for an USBVCP based serial API communication interfaces.

3.4.7 IMA parameter

This parameter specifies if the IMA features are included. This parameter is only available for serial API applications based on static controllers. The IMA features support a service provider network installation and maintenance procedure.

Table 7. Description of IMA parameters in command line

Parameter	Result
No parameters specified	Building application based on static controller having IMA variant.
IMA=nonIMA	Skip building application based on static controller having IMA variant.
IMA=IMA_ENABLED	Building application based on static controller having IMA variant.

3.4.8 LIBRARY parameter

This parameter specifies the Z-Wave protocol to be used when building the application.

Table 8. Description of LIBRARY parameters in command line

Parameter	Result
No parameters specified	Building application based on all supported Z-Wave protocols.
LIBRARY=controller_bridge	Building application based on bridge controller.
LIBRARY=controller_portable	Building application based on portable controller.
LIBRARY=controller_static	Building application based on static controller.
LIBRARY=controller_static_norep	Building application based on static controller without repeater functionality.
LIBRARY=controller_static_single	Building application based on static controller used in ERTT measurements. This library can suppress retransmissions of a frame.
LIBRARY=slave_enhanced_232	Building application based on enhanced 232 slave, which use an external NVM
LIBRARY=slave_routing	Building application based on routing slave, which do not use an external NVM

3.4.9 SENSOR_TYPE parameter

This parameter specifies battery operated or always listening device when building application.

Table 9. Description of SENSOR_TYPE parameters in command line

Parameter	Result
No parameters specified	Building all applications.
SENSOR_TYPE=NON_BATT	Building application for always listening devices.
SENSOR_TYPE=BATTERY	Building application for battery operated devices, which typically sleeps.

3.4.10 TEST_INTERFACE parameter

The TEST_INTERFACE parameter enables a user to use the Test Interface in the Z-Wave Plus Framework. See the following table for possible values.

Table 10. Description of TEST_INTERFACE parameter in command line

Parameter	Result
No parameters specified	Building all combinations.
TEST_INTERFACE=YES	The application is built with the Test Interface.
TEST_INTERFACE=NO	The application is not built with the Test Interface.

3.4.11 UVISION parameter

The UVISION parameter enable a user to generate Keil uVision4 project files for the embedded sample application based on a 500 Series Z-Wave Single Chip. The uVision4 project files are generated by opening a command prompt (DOS box) in the relevant sample application directory and adding “UVISION=1” to command MK.

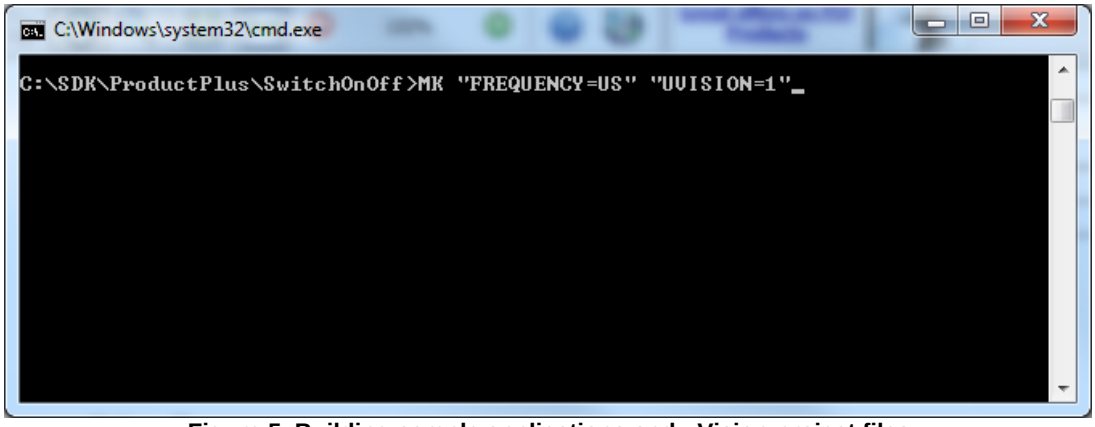

Figure 5, Building sample applications and uVision project files

3.4.12 UVISIONPREBUILD parameter

The UVISIONPREBUILD parameter is not intended for use directly by the customer. The option is used by a generated uVision project to fullfill the need for making things before uVision starts building.

In uVision terms: Options for target->User->Before Build/Rebuild->Run #1 or <BeforeMake><UserProg1Name> in the .uvproj file contains a make-command with this option set (UVISIONPREBUILD=1)

This could include generating some header files, which the target program depends on.

3.4.13 UVISIONPOSTBUILD parameter

The UVISIONPOSTBUILD parameter is not intended for use directly by the customer. The option is used by a generated uVision project to fullfill the need for making things after uVision has built the executable.

In uVision terms: Options for target->User->After Build/Rebuild->Run #1 or <AfterMake><UserProg1Name> in the .uvproj file contains a make-command with this option set (UVISIONPOSTBUILD=1)

This could include converting the absolute object file to a hex file, adding a checksum, adding a bootloader, or other things.

3.4.14 WATCHDOG parameter

By default, the watchdog is disabled in the Z-Wave Plus sample applications. This is an advantage during development and testing prior to final release testing. An enabled watchdog may prevent firmware crashes and stalls from being discovered during development and initial testing. As a side note, debugging a system with an enabled watchdog can be a challenge.

A released product SHOULD have the watch dog enabled. Remember to conduct a full system test on a product having the watchdog enabled.

Table 11. Description of WATCHDOG parameters in command line

Parameter	Result
No parameters specified	Building two targets: * watchdog disabled and * watchdog enabled
WATCHDOG=WATCHDOG_DISABLED	Building target having watchdog disabled.
WATCHDOG=WATCHDOG_ENABLED	Building target having watchdog enabled.

4 DEVELOPING APPLICATION CODE

All sample applications [1] for the 500 Series Z-Wave Single Chip contain source code and makefiles that allow the developer to modify and compile the applications without modifying makefiles, etc. Sample applications are built by calling the MK.BAT script file that is located in the sample application directory. Alternatively, use the integrated development environment uVision from Keil. However, the uVision project files must first be generated as described in section 3.4.11.

A 32kB code bank is allocated for application development and available data memory is 4KB. Internal NVM (same as MTP in 400 Series) now supports 64 Bytes. The application developer MUST NOT exceed the above limitations due to future protocol enhancements. External NVM depends on the Z-Wave Development Board used.

4.1 Porting Requirements

A Z-Wave application based on earlier Z-Wave Single Chips must first be ported to the 500 Series Single Chip. For details about porting, refer to [4], [5] and [6].

The Z-Wave Plus applications based on earlier SDKs may also require porting to a newer version of the Z-Wave Plus Framework. For details, refer to [7].

4.2 Application Interrupt Service Routine

The application Interrupt Service Routines (ISR) must for various reasons be located in the COMMON bank and ISR should therefore be defined in a specific module and explicitly located in COMMON.

Set the mask bit to enable the particular interrupt used by application in ApplicationInitSW. Set also edge/level flag in case an external interrupt is used. Do not set EA (global enable) bit to 1 to enable the interrupt system because this is taken care of by the Z-Wave protocol.

4.2.1 uVision project

For uVision project this can be done as follows:

- Use the uvision project generator to create uvision project.
- Open the project in uvision IDE
- Create a new c (for example int1.c) file and add your ISR to that file.
- Add the file under the Appl Source Code (COMMON) folder in uvision project browser.

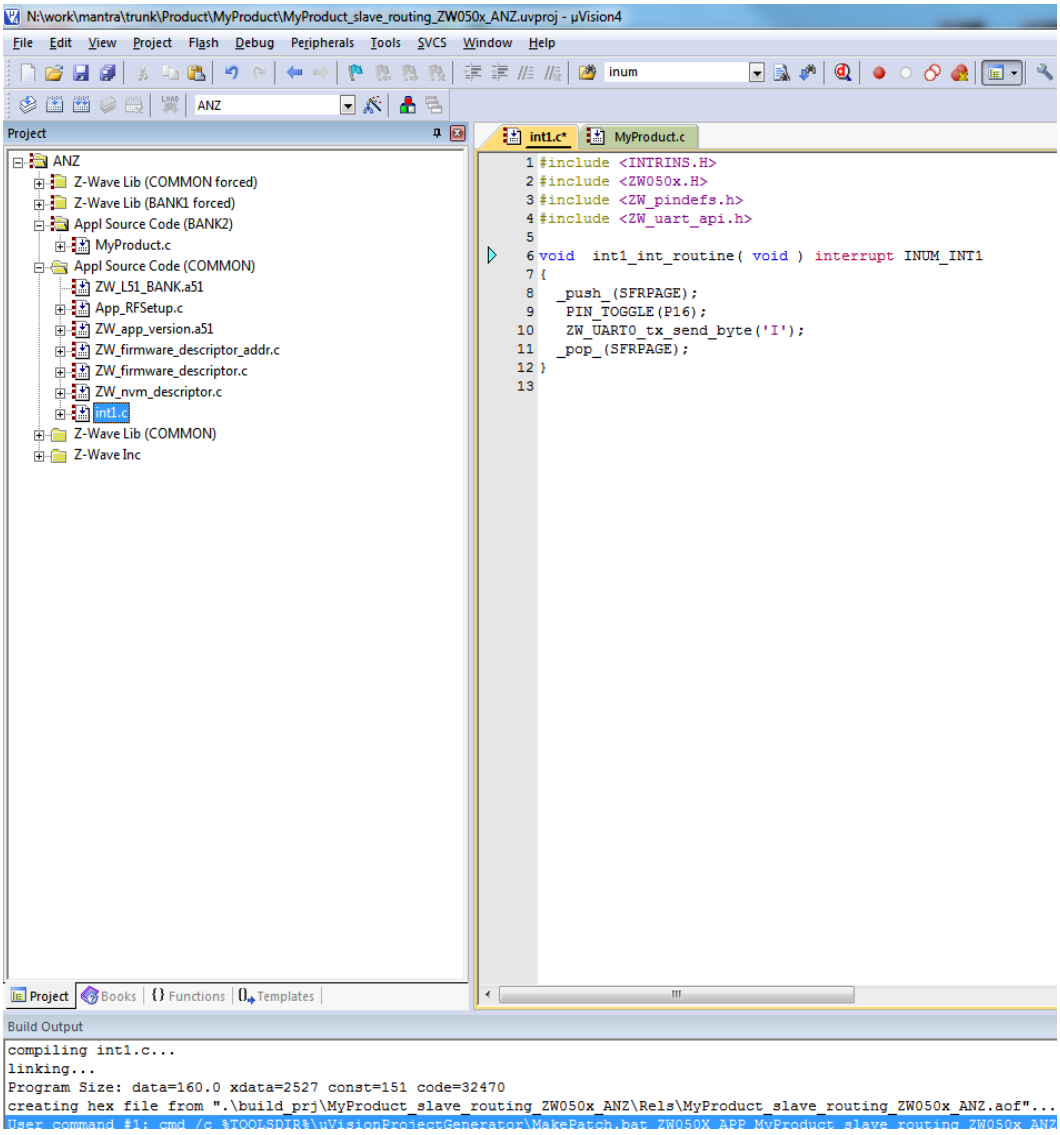

Figure 6, Adding interrupt module to uVision

Edit the int1.c options

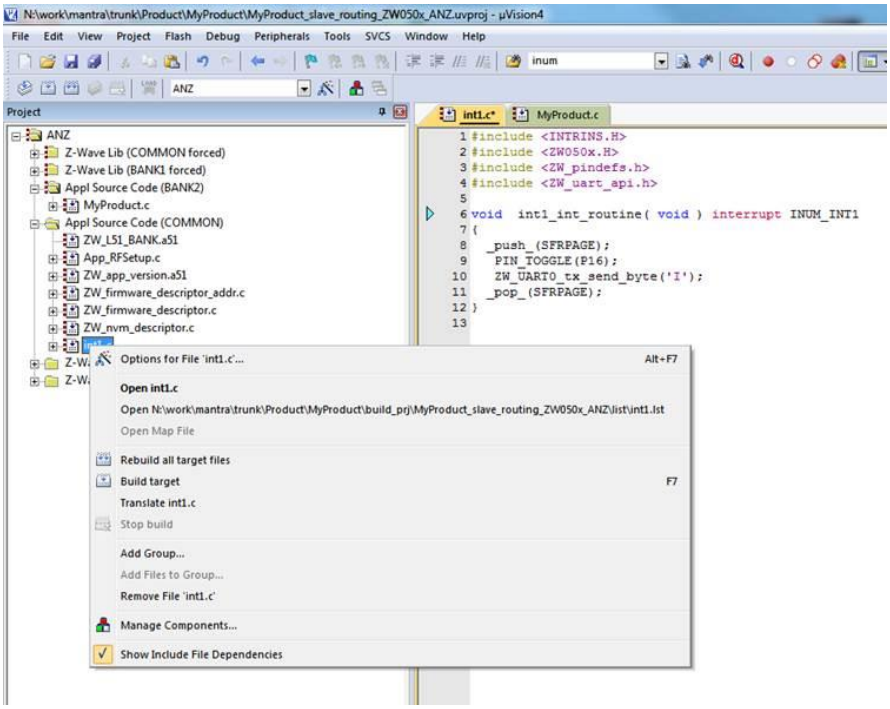

Figure 7, Opening file options for interrupt module

Under the C51 tab copy the content of the define box from another file to the int1.

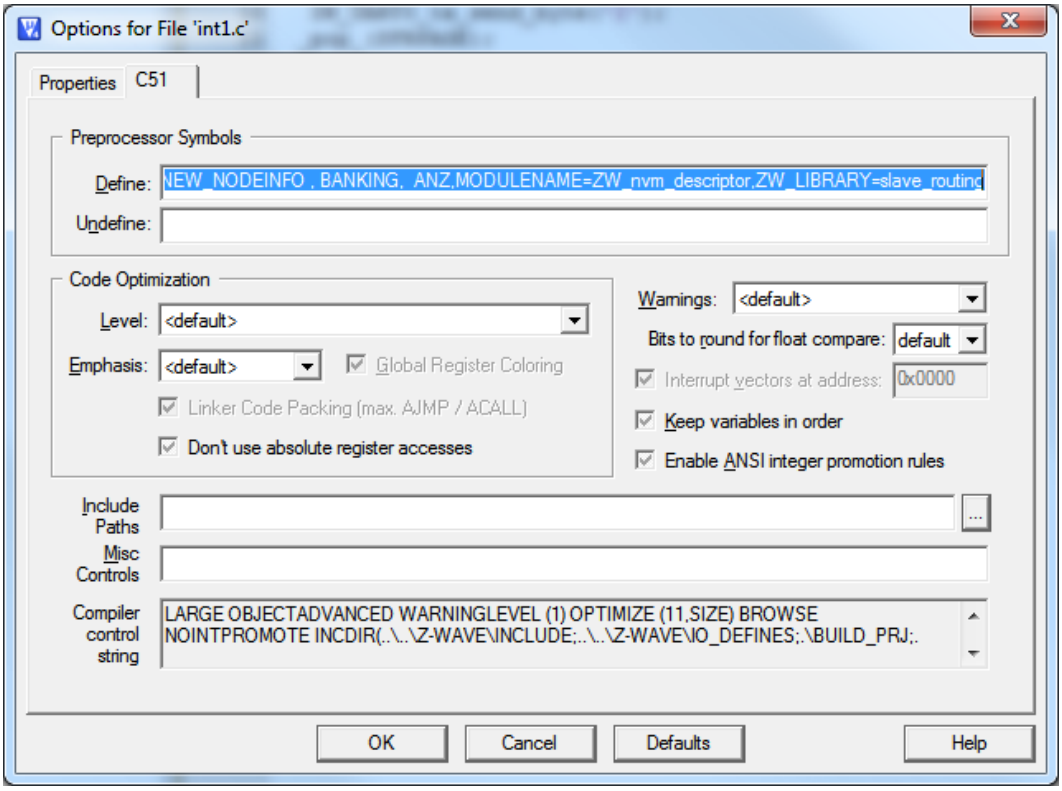

Figure 8, Adding preprocessor symbols to interrupt module

4.2.2 Makefile projects

For Makefile based projects a few changes must be made in some of the makefiles. First, add the following changes to the Makefile file in the ProductPlus or Product application folder.

```
119 #------------------------------------------------------------------------------ =  119 #------------------------------------------------------------------------------
    ------------------------------------                                               ------------------------------------
120                                                                                    120
121 ifeq ($(BOOTLOADER),)                                                            121 ifeq ($(BOOTLOADER),)
122 # List of all application modules.                                               122 # List of all application modules.
123 # For use in both Bootloader enabled mode and none Bootloader                    123 # For use in both Bootloader enabled mode and none Bootloader
    enabled mode                                                                       enabled mode
124 RELFILES:=\                                                                    124 RELFILES:=\
125 LEDdim.obj\                                                                    125 LEDdim.obj\
126 one_button.obj\                                                                126 one_button.obj\
127 led_control.obj\                                                               127 led_control.obj\
128 association.obj\                                                               128 association.obj\
129 slave_learn.obj\                                                               129 slave_learn.obj\
130 eeprom.obj                                                                     130 eeprom.obj
131 ifneq ($(SCHEME),)                                                            131 ifneq ($(SCHEME),)
132 RELFILES+=\                                                                    132 RELFILES+=\
133 ZW_Security_AES_module.obj\                                                    133 ZW_Security_AES_module.obj\
134 ZW_TransportSecurity.obj                                                       134 ZW_TransportSecurity.obj
135 endif                                                                         135 endif
136 ifeq ($(CODE_MEMORY_MODE),banking)                                           +-
137 # build the project the banked way.
138 COMMON_RELFILES:=\
139 int1.obj
140 endif
141 endif                                                                         =  136 endif
142                                                                                    137
143 #------------------------------------------------------------------------------ =  138 #------------------------------------------------------------------------------
    ------------------------------------                                               ------------------------------------
```

4.3 Bootloader

The bootloader is used to support firmware update in product and is included in product HEX-file by enabling bootloader functionality in build environment.

For SDK 6.5x:

1. mk “BOOT=BOOTLOADER” //Build bootloader
2. mk “BOOT=BOOTLOADER_ENABLED” //Build target and link bootloader into target

For newer SDK’s (not SDK 6.5x):

1. mk “BOOT=BOOTLOADER_ENABLED” //Build target with bootloader

The bootloader library implements a bootloader which resides in the lower 6Kbyte (SDK 6.5x) of the address space. On System reset the bootloader determines if a valid new firmware exists in the external NVM and if so, it determines if the firmware should be flashed to CPU Flash and booted. If the System reset reason (wakeup reason) is either external interrupt or WUT then the bootloader just boots the current firmware, thereby skipping the NVM firmware check. The bootloader vectors all other interrupts but the reset vector directly to any application or library defined interrupt vector.

The current bootloader implementation supports either a 128Kbyte or a 256Kbyte (NVM_SIZE) external NVM (see Figure 9).

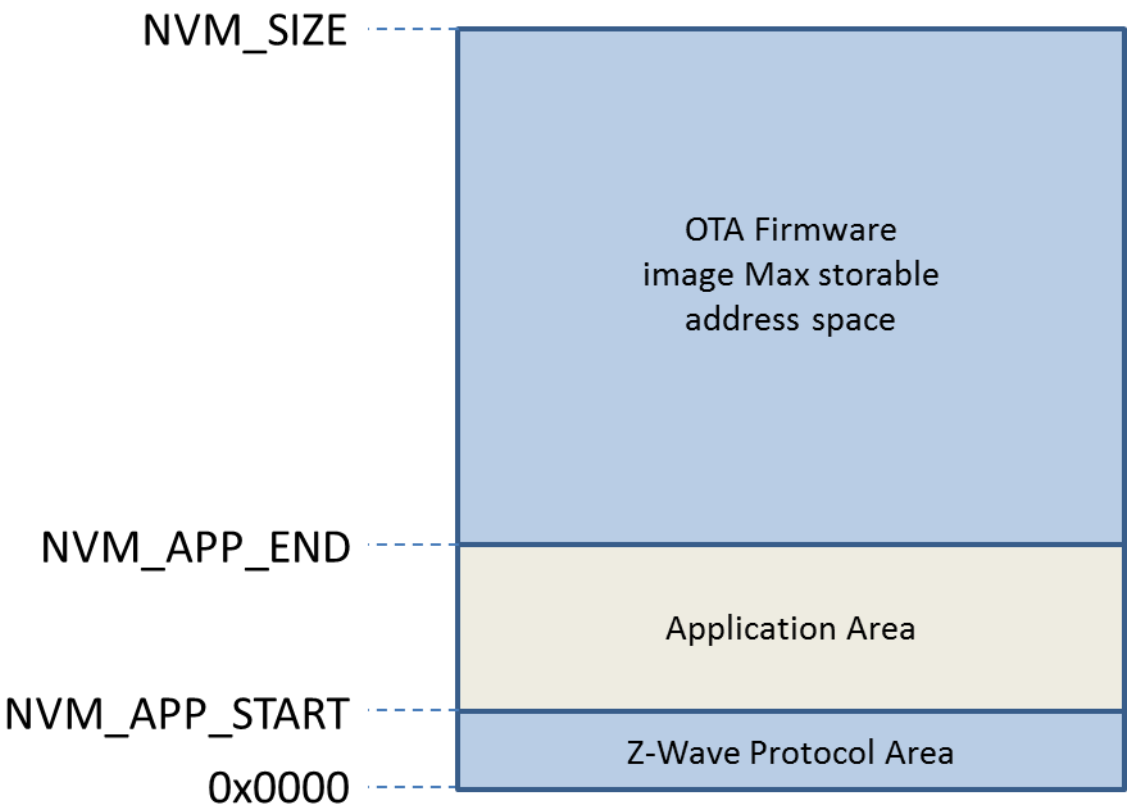

Figure 9, NVM layout for 128Kbytes and 256Kbyte EEPROM.

NVM application area start address (NVM_APP_START) is not fixed and changed dependent of SDK version. The bootloader handle moving Application area under a firmware update to new address. Table 12,describe how to find NVM_APP_START and application size dependent of SDK's.

Table 12, show how to find NVM_APP_START and NVM_APP_END address in map file.

SDK	Address name	Address of descriptor name in MAP-file	Comment
6.5x	NVM_APP_START	nvmApplDescriptor	NVM application data are part of nvmApplDescriptor and not shown as individual address in MAP-file. See application eeprom.h. (Fixed start address 6000h for SDK 6.5x)
	NVM_APP_END	nvmDescriptor	See application eeprom.c
6.61->	NVM_APP_START	nvmApplicationSize	See section 4.3.1.
	NVM_APP_END	nvmApplicationDescriptor	

NVM application area size depends on EEPROM size. For 128KB EEPROM is max address (NVM_APP_END) 0x77FE and 256KB EEPROM is max address (NVM_APP_END) 0x1FFFE.

4.3.1 SDK6.61 NVM descriptor layout

From SDK 6.61 a new NVM descriptor layout is introduced for a more compact NVM layout. NVM module variables are not included in a struct as in SDK 6.5.x (nvmApplDescriptor). This gives the possibility to see each application NVM variable in the map file. Figure 10 show the new NVM layout.

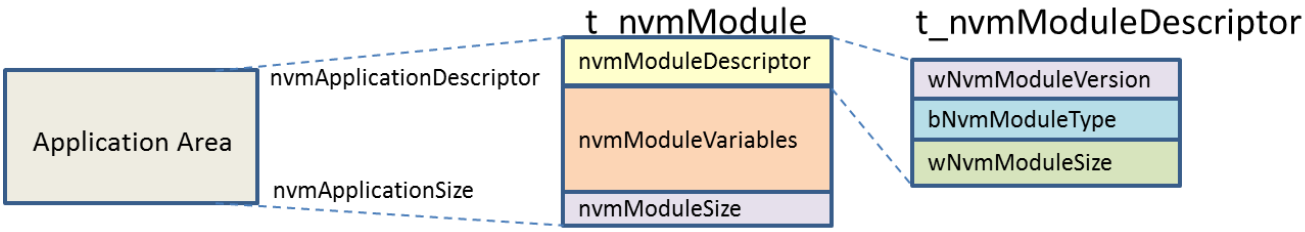

Figure 10, show SDK 6.61 how application NVM is configured.

4.4 OTA Firmware Update

Many sample applications already implements OTA firmware update support. However, the following sections describe how to enable OTA firmware update on a ported application to the 500 Series Single Chip.

The current implementation of Firmware Update Meta Data Command Class uses Version 2 and therefore the OTA transfers maximum 128 Kbyte (or from 0x00000 to last byte defined – last byte in Bank 3).

4.4.1 Handling uncompressed OTA files

As the Bootloader occupies the space from 0x00000-0x017FF (6Kbyte) the first 0x01800 bytes are all 0xFF in the Firmware image transferred. The first 0x01800 bytes are thrown away if a 128KB NVM are used but first after being included in the running CRC16 calculation. The running CRC16 calculation is a CRC16 calculation done on the full received Firmware Image and must match the Checksum received in the FIRMWARE_UPDATE_MD_REQUEST_GET frame transmitted by the HOST to initiate the OTA sequence.

4.4.2 Handling compressed OTZ files

Starting with SDK 6.70, over-the-air update files are compressed. Compressed OTA files can be recognized by their file extension .OTZ. OTZ files are Intel Hex encoded binary files. The OTZ file must be converted from Intel Hex to binary, and then transmitted unmodified as Firmware Update Meta Data Command Class-encapsulate payload [10]. The RECOMMENDED fragment size is 20 bytes.

Note: The padding mechanism used for uncompressed OTA files does not apply to compressed OTZ files. The usual Firmware Update Meta Data Command Class CRC16 calculation must cover the entire binary contents of the OTZ file.

4.4.3 Z-Wave Plus OTA Firmware Update implementation

Here are the steps to update a Z-Wave Plus sample application to enable support of OTA firmware update:

1. Add header files:
    a. #include <CommandClassFirmwareUpdate.h>
    b. #include <ota_util.h>
2. Add Cmd Class in nodeInfo[] list:
    a. COMMAND_CLASS_FIRMWARE_UPDATE_MD_V2
3. In ApplicationCommandHandler switch case add:

```
case COMMAND_CLASS_FIRMWARE_UPDATE_MD_V2:
  HandleCommandClassFWUpdate(txOption, sourceNode, pCmd, cmdLength);
  break;
```

4. Implement in *application* ***BOOL ZCB_OTAStart ()*** and ***Void ZCB_OTAFinish (OTA_STATUS otaStatus)***. Please see header documentation on ***OtaInit(..)***.
5. In ApplicationInitSW() call **OtaInit(..)** to initialize ota_util module and include ***ZCB_OTAStart*** and ***ZCB_OTAStart*** as input parameters.
6. In the makefile add CommandClassFirmwareUpdate.obj and ota_util.obj
7. In the makefile include BOOTLOADER and BOOTLOADER_ENABLED flag and build target. See how it is done in a working file.

Regarding an OTA firmware update implementation see for example in the SwitchOnOff.c file for the Z-Wave Plus Switch On/Off application.

```
/*============================ OTA_Finish ==============================
** Function description
** OTA is finish.
**
** Side effects:
**
**--------------------------------------------------------------------*/
void
OTA_Finish(OTA_STATUS otaStatus) /*Status on OTA*/
{
  /* Just reset node to start on new image or cleanup if update failed*/
  ZW_WatchDogEnable(); /*reset asic*/
  while(1);
}


/*============================ OTA_Start ==============================
** Function description
** Ota_Util calls this function when firmware update is ready to start.
** Return FALSE if OTA should be rejected else TRUE
**
** Side effects:
**
**--------------------------------------------------------------------*/
BOOL   /*Return FALSE if OTA should be rejected else TRUE*/
OTA_Start(WORD fwId, WORD CRC)
{
  return TRUE;
}
```

4.4.4 Firmware updating SDK 6.51.xx/6.61.xx to the SDK with S2 support (SDK 6.7x)

Firmware updates of an SDK 6.51.x application to the SDK with S2 support need to be done through an intermediate application based on SDK 6.6x (not SDK 6.61.00!). A SDK 6.6x intermediate application includes code for updating the bootloader (in node) to a SDK 6.71.01+ bootloader supporting compressed firmware update and firmware update code supporting this as this is needed for making the firmware update to the SDK with S2.

- Phase 1: Port product to SDK 6.6x.
  - Update application Makefile, see section 4.4.4.1.
  - Update application source file, see section 4.4.4.2
  - Build SDK 6.6x project with: >mk “BOOT=BOOTLOADER_UPDATE” …
    - Product build with “BOOT=BOOTLOADER_UPDATE” only supports firmware update with OTZ firmware files!
    - Output: OTA file.
  - Firmware Update target with OTA file.
    - Now product only supports firmware update with OTZ firmware files!
- Phase 2: Update product to target with S2 support.
  - Port product to SDK 6.7x. Customer have two possibilities:
    - Port SDK 6.5x product to SDK 6.7x
    - Port SDK 6.6x intermediate product to SDK 6.7x
  - Build SDK 6.7x project with: >mk “BOOT=BOOTLOADER_ENABLED”. …

- Output OTZ file.
- Firmware Update target with OTZ file.

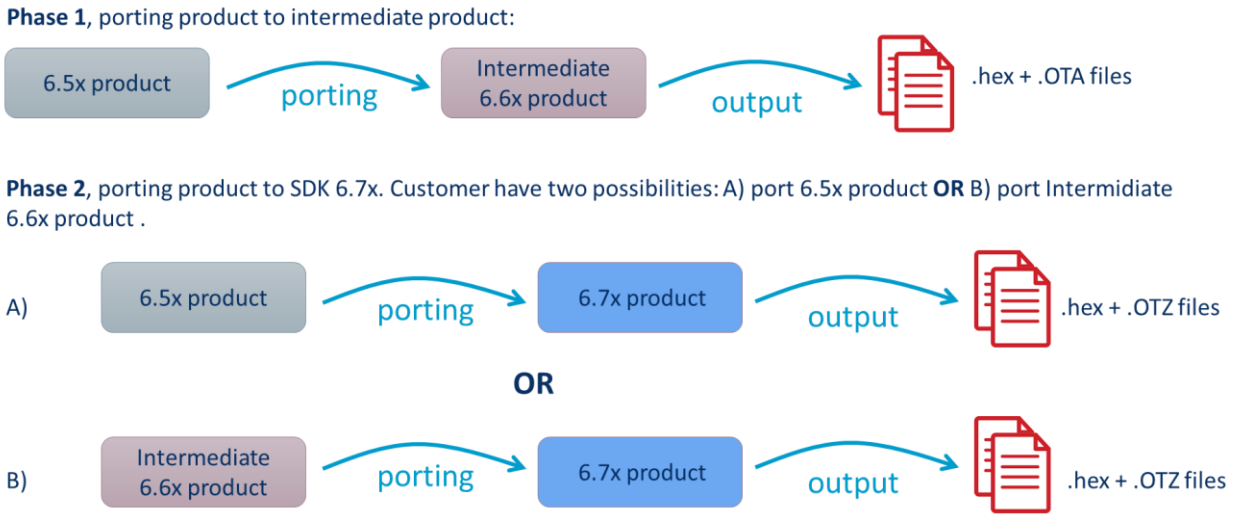

Figure 11, shows phases for updating sample application to support S2 based on the SDK 6.7x.

The intermediate application MUST be a certified application to guarantee application still works if firmware updates fails!

4.4.4.1 Intermediate application Makefile

Application Makefile need to be updated to support BOOTLOADER_UPDATE.

1, Add object files dependent of flags BOOTLOADER_ENABLED, BOOTLOADER_UPDATE:

```
ifneq ($(BOOTLOADER_UPDATE),)
RELFILES+=\
 firmware_update_nvm.obj\
 firmware_update_nvm_write.obj\
 otz_check_compressed_crc.obj
endif

ifneq ($(BOOTLOADER_ENABLED)$(BOOTLOADER_UPDATE),)
RELFILES+=\
 CommandClassFirmwareUpdate.obj\
 ota_util.obj
endif
```

2, Change target name from:

```
# Target name (Name of your target directory and base name of your target
files)

ifneq ($(BOOTLOADER_ENABLED),)
ifeq ($(LIBRARY),slave_routing)
LIBRARY:=slave_enhanced_232
ENH:=_enhanced_232
endif
TARGET:=$(APP)_$(LIB)_OTA_$(CHIPPACK)_$(COUNTRY)$(TEST)$(SEC)$(EP)
else
TARGET:=$(APP)_$(LIB)_$(CHIPPACK)_$(COUNTRY)$(TEST)$(SEC)$(EP)
endif
```

To:

```
# Target name (Name of your target directory and base name of your target
files)

ifneq ($(BOOTLOADER_ENABLED)$(BOOTLOADER_UPDATE),)
ifeq ($(LIBRARY),slave_routing)
LIBRARY:=slave_enhanced_232
ENH:=_enhanced_232
endif
endif
TARGET:=$(APP)_$(LIB)
ifneq ($(BOOTLOADER_ENABLED),)
TARGET:=$(TARGET)_OTA
endif
ifneq ($(BOOTLOADER_UPDATE),)
TARGET:=$(TARGET)_OTU
endif
TARGET:=$(TARGET)_$(CHIPPACK)_$(COUNTRY)$(TEST)$(SEC)
```

3, In section “Addition to the variants to build”.

Add flag BOOTLOADER_UPDATE to boot load list “LIST_OF_BOOT”:

```
LIST_OF_BOOT:=nonBOOT BOOTLOADER_ENABLED BOOTLOADER_UPDATE
```

Add boot options:

```
ifeq ($(BOOT),BOOTLOADER_ENABLED)
BOOT_OPTION:=BOOTLOADER_ENABLED=1
BOOTLOADER_ENABLED:=1
endif
ifeq ($(BOOT),BOOTLOADER_UPDATE)
BOOT_OPTION:=BOOTLOADER_UPDATE=1
BOOTLOADER_UPDATE:=1
endif
```

4, In section “Filtering out targets” add BOOTLOADER_UPDATE flag:

```
# Filtering out targets, which have no meaning, or are just not needed.
#
# The controlling variant name must be listed before the controlled variant
name in the LIST_OF_VARIANTS
# for this to work.
#
#LIST_OF_VARIANTS:=LIBRARY FREQUENCY CODE_MEMORY_MODE SENSOR_TYPE CHIP
ifneq ($(BOOTLOADER_ENABLED)$(BOOTLOADER_UPDATE),)
# BOOTLOADER_ENABLED has no meaning for slave_routing, because there is no
room for a new firmware image in NVM
LIST_OF_LIBRARY:=$(filter-out slave_routing, $(LIST_OF_LIBRARY))
# BOOTLOADER_ENABLED cannot build for some of the controller types, because
there is no room for code in COMMON
LIST_OF_LIBRARY:=$(filter-out controller_bridge controller_static,
$(LIST_OF_LIBRARY))
endif
```

4.4.4.2 Intermediate application source code

SDK 6.61.01 has support for building intermediate application and depends on application Makefile is change as described in section 4.4.4.1.

Search and replace flag “BOOTLOADER_ENABLED”:

```
#ifdef BOOTLOADER_ENABLED
```

To:

```
#if defined(BOOTLOADER_ENABLED) | defined(BOOTLOADER_UPDATE)
```

Module “ota.h” API is changed to use command class firmware update version 4, why application source code need to be updated to new API:

```
void
OtaInit(
  BYTE txOption,
  BOOL (CODE *pOtaStart)(void),
  VOID_CALLBACKFUNC(pOtaFinish)(BYTE val));
```

To:

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

void
OtaHostFWU_WriteFinish(void);

void
OtaHostFWU_Status( BOOL userReboot, BOOL status );

WORD
handleFirmWareIdGet( BYTE n);
```

Under initialization The Z-Wave Plus Applications Framework call Bootloader update function "`OverwriteBootloader()`" to start updating new bootloader. It is important application call "`Transport_OnApplicationInitSW()`" in "`ApplicationInitSW()`" to kick this process.

4.5 Z-Wave Plus Application implementation

The Z-Wave Plus Applications Framework facilitates implementation of robust Z-Wave Plus compliant products in a fast and cost effective manner. The Framework provides a number of modules and resides on top of the Z-Wave Protocol API calls. For details refer to [7].

The Z-Wave Plus applications must be implemented according to the Z-Wave Plus specifications [8]-[15].

4.5.1 External NVM Application data

To allow full utilization of the 500 series internal memory during OTA and future protocol feature roadmap results in the following recommendations for external NVM memory:

To enable full utilization of the 500 series with respect to future protocol features and OTA firmware update results in the following recommendations for external NVM:

Minimum requirements when selecting external NVM for devices without OTA firmware update support:

- 32KB – Required for slave and controller devices

Minimum requirements when selecting external NVM for devices without OTA firmware update support:

- 128KB – Required for slave devices
- 256KB – Required for controller devices

Initialization of the external NVM is completely handled by the Z-Wave protocol and thereby obsoleting the NVM initialization file extern_epp.hex. SDK 6.61 and later introduced a new NVM layout (see 4.3.1), which dynamically allocated the necessary address space for protocol and application respectively.

The former SDK 6.51 used fixed addresses for allocation of address space for protocol and application respectively.

Notice that NVM above 64K cannot be addressed with the old memory API calls (parameter offset is a WORD). To access NVM memory above 64KB use the NVM_ext_read_long_buffer() and NVM_ext_write_long_buffer() API calls. These two functions can handle the full NVM address space (0-NVM_SIZE) and therefore should be used with care because library resides in the beginning of external NVM.

4.5.2 External NVM Application data layout and location

In order to keep your application's NVM variables ordered and kept together, they must all be defined, and optionally initialized (once) in one file: eeprom.c, and declared in one file: eeprom.h. This is very important, when you want to facilitate Firmware Update. The NVM variables layout must be kept totally equal between versions of your software. You can only append new variables following the ones in the old version of your software. Even if you no longer use a variable in the new version of your software, the variable must be kept forever.

Make sure the compiler won't shuffle around with these variables, as there are external dependencies. You do this by using the compiler directive in the source file defining the NVM variables:

#pragma ORDER

Note from Keil C51 manual:
"Variables with memory type, initilalization, and without initilalization have all different tables. Therefore only variables with the same attributes are kept within order."

Note from Keil knowledgebase: (http://www.keil.com/support/docs/901.htm)
"The order is not necessarily taken from the variable declarations, but the first use of the variables."

Therefore, when using #pragma ORDER to order variables, declare them in the order they should be in a collection. And none of them may be declared or known in any way from other header files.

It is a bit confusing reading the Keil notes, because they are not fully aware of whether we speak of variable declarations or variable definitions. Somewhere in Keil's knowledge base, you can find an advice regarding ordering your varables, that tells you to put them into a struct, and thereby make their layout fixed. But this is not the best solution. If you want to make some kind of first time initialization of a specific one of your aplication's NVM variables, you can't do this if it resides in a struct. Then you have to initialize the whole struct.

Use the SerialAPIPlus application's eeprom.c and eeprom.h as a template, when you construct a new application.

To make sure, that your application NVM variable layout matches from your "old" application version to your "new" application version, always keep an eye on the map file.

You can make a listing of the NVM variables by using the following small script (or a subset of it):

```
@echo off & setlocal enableextensions enabledelayedexpansion
::
::============================================================
:: Windows batch script for generating NVM map for all variants of firmware.
:: Parameters:
cd build
for /D %%i in (*) do findstr /C:"PUBLIC    HDATA" %%i\Rels\%%i.map | sort /O ..\%%i.nvm_map
endlocal & goto :EOF
```

4.5.2.1 SDK 6.6x+ External NVM Application data layout and location

To accommodate the SDK 6.6x+ NVM layout an application must define needed external NVM variables in eeprom.c (example from SDK 6.61.01 SerialAPIPlus):

```
/* eeprom.c (SerialAPIPlus sample application) */
#pragma USERCLASS(CONST=NVM)

#pragma ORDER

#include "ZW_basis_api.h"
#include "eeprom.h"
#include "ZW_nvm_descriptor.h"

/*---------------------------------------------------------------------------*/
/* NVM layout SerialAPIPlus (embedded application part) (as in t_nvmModule) */
/* (begin)                                                                   */

/* Offset from &nvmModule where nvmModuleDescriptor structure is placed */
t_NvmModuleSize far nvmApplicationSize = (t_NvmModuleSize)&_FD_EEPROM_L_;

/* APPLICATION SPECIFIC PART START */

/* NVM variables for your application                                        */
BYTE far EEOFFSET_MAGIC_far;
#ifdef ZW_SLAVE
BYTE far EEOFFSET_LISTENING_far;
BYTE far EEOFFSET_GENERIC_far;
BYTE far EEOFFSET_SPECIFIC_far;
BYTE far EEOFFSET_CMDCLASS_LEN_far;
#else
BYTE far EEOFFSET_CMDCLASS_LEN_far;
#endif
BYTE far EEOFFSET_CMDCLASS_far[APPL_NODEPARM_MAX];
BYTE far EEOFFSET_WATCHDOG_STARTED_far;

/* APPLICATION SPECIFIC PART END */

/* NVM module descriptor for module. Located at the end of NVM module.      */
/* During the initialization phase, the NVM still contains the NVM contents */
/* from the old version of the firmware.                                    */
t_nvmModuleDescriptor far nvmApplicationDescriptor =
{
  (t_NvmModuleSize)&_FD_EEPROM_L_,      /* t_NvmModuleSize wNvmModuleSize   */
  NVM_MODULE_TYPE_APPLICATION,          /* eNvmModuleType bNvmModuleType    */
  (WORD)&_APP_VERSION_                  /* WORD wNvmModuleVersion           */
};

/* NVM layout SerialAPIPlus (as in t_nvmModule) (end)                        */
/*---------------------------------------------------------------------------*/

/* NVM module update descriptor for this new version of firmware. Located   */
/* in code space.                                                           */
const t_nvmModuleUpdate code nvmApplicationUpdate =
{
  (p_nvmModule)&nvmApplicationSize,     /* nvmModulePtr                     */
  /* nvmApplicationDescriptor is the first new NVM variable since devkit_6_5x_branch */
  (t_NvmModuleSize)((WORD)&nvmApplicationDescriptor),  /* wNvmModuleSizeOld */
  {
    (t_NvmModuleSize)&_FD_EEPROM_L_,    /* t_NvmModuleSize wNvmModuleSize   */
    NVM_MODULE_TYPE_APPLICATION,        /* eNvmModuleType bNvmModuleType    */
    (WORD)&_APP_VERSION_                /* WORD wNvmModuleVersion           */
  }
};
```

And the matching eeprom.h for making the NVM variables accessible for source code.

```
/* eeprom.h */
#include "ZW_nvm_descriptor.h"

/* NVM allocation declarations */

/* APPLICATION SPECIFIC PART START */

#define MAGIC_VALUE       0x42

/* NVM layout SerialAPIPlus (embedded application part) */
extern t_NvmModuleSize far nvmApplicationSize;
extern BYTE far EEOFFSET_MAGIC_far;
#ifdef ZW_SLAVE
extern BYTE far EEOFFSET_LISTENING_far;
extern BYTE far EEOFFSET_GENERIC_far;
extern BYTE far EEOFFSET_SPECIFIC_far;
extern BYTE far EEOFFSET_CMDCLASS_LEN_far;
#else
extern BYTE far EEOFFSET_CMDCLASS_LEN_far;
#endif
extern BYTE far EEOFFSET_CMDCLASS_far[];
extern BYTE far EEOFFSET_WATCHDOG_STARTED_far;

/* APPLICATION SPECIFIC PART END */

extern t_nvmModuleDescriptor far nvmApplicationDescriptor;

/* The starting address of the segment ?FD?EEPROM (to be used as a constant as (WORD)&_FD_EEPROM_S_) */
extern unsigned char _FD_EEPROM_S_;
/* The length of the segment ?FD?EEPROM in bytes (to be used as a constant as (WORD)&_FD_EEPROM_L_) */
extern unsigned char _FD_EEPROM_L_;
```

SerialAPIPlus also have allocated space in external NVM for use by the HOST connected to the SerialAPIPlus module. To accommodate the HOST external NVM usage SerialAPI uses the nvmHost.c to define the requirements of HOST accessible external NVM.

```
#pragma USERCLASS(CONST=NVM)

#include "ZW_basis_api.h"
#include "nvmHost.h"
#include "ZW_nvm_descriptor.h"

/*------------------------------------------------------------------------------*/
/* NVM layout SerialAPIPlus (host application part) (as in t_nvmModule)     */
/* (begin)                                                                  */

/* Offset from &nvmModule where nvmModuleDescriptor structure is placed */
t_NvmModuleSize far nvmHostApplicationSize = (t_NvmModuleSize)&_FD_NVMHOST_L_;

/* APPLICATION SPECIFIC PART START */

/* Host application non volatile variables */
BYTE far EEOFFSET_HOST_OFFSET_START_far[NVM_SERIALAPI_HOST_SIZE];

/* APPLICATION SPECIFIC PART END */

/* NVM module descriptor for module. Located at the end of NVM module.      */
/* During the initialization phase, the NVM still contains the NVM contents */
/* from the old version of the firmware.                                    */
t_nvmModuleDescriptor far nvmHostApplicationDescriptor =
{
  (t_NvmModuleSize)&_FD_NVMHOST_L_,     /* t_NvmModuleSize wNvmModuleSize   */
  NVM_MODULE_TYPE_HOST_APPLICATION,     /* eNvmModuleType bNvmModuleType    */
  (WORD)&_APP_VERSION_                  /* WORD wNvmModuleVersion           */
};
```

```
/* NVM layout SerialAPIPlus (as in t_nvmModule) (end)                    */
/*-------------------------------------------------------------------------*/

/* NVM module update descriptor for this new version of firmware. Located   */
/* in code space.                                                          */
const t_nvmModuleUpdate code nvmHostApplicationUpdate =
{
  (p_nvmModule)&nvmHostApplicationSize, /* nvmModulePtr                    */
  /* nvmHostApplicationDescriptor is the first new NVM variable since devkit_6_5x_branch */
  (t_NvmModuleSize)((WORD)&nvmHostApplicationDescriptor), /* wNvmModuleSizeOld */
  {
    (t_NvmModuleSize)&_FD_NVMHOST_L_,   /* t_NvmModuleSize wNvmModuleSize   */
    NVM_MODULE_TYPE_HOST_APPLICATION,   /* eNvmModuleType bNvmModuleType    */
    (WORD)&_APP_VERSION_                /* WORD wNvmModuleVersion           */
  }
};
```

And the matching nvmHost.h for making the NVM variables accessible for source code. Also here is the NVM_SERIALAPI_HOST_SIZE which defines how much external NVM are available for HOST.

```
#include "ZW_nvm_descriptor.h"

/* NVM allocation definitions */

/* APPLICATION SPECIFIC PART START */

#if defined(ZW_CONTROLLER) || defined(ZW_SLAVE_32)
/* NVM is 16KB, 32KB or even more (you decide the size of your SPI EEPROM or FLASH chip) */
/* Use only a reasonable amount of it for host application */
#define NVM_SERIALAPI_HOST_SIZE 2048
#else
/* For routing slaves the total number of NVM data bytes available is 254 Bytes */
#define NVM_SERIALAPI_HOST_SIZE 16
#endif

/* APPLICATION SPECIFIC PART END */

/* NVM layout SerialAPIPlus (host application part) */
extern t_NvmModuleSize far nvmHostApplicationSize;

/* APPLICATION SPECIFIC PART START */

extern BYTE far EEOFFSET_HOST_OFFSET_START_far[NVM_SERIALAPI_HOST_SIZE];
/* APPLICATION SPECIFIC PART END */

extern t_nvmModuleDescriptor far nvmHostApplicationDescriptor;

/* The starting address of the segment ?FD?NVMHOST (to be used as a constant as (WORD)&_FD_NVMHOST_S_) */
extern unsigned char _FD_NVMHOST_S_;
/* The length of the segment ?FD?NVMHOST in bytes (to be used as a constant as (WORD)&_FD_NVMHOST_L_) */
extern unsigned char _FD_NVMHOST_L_;
```

4.5.3 External NVM Application data initialization

As a part of program initialization, NVM data is initialized in a fashion like normal data is.

Initialized data is when the data is initialized in its definition. All data, which is not defined as initialized will be cleared to zero.

NVM data shall only be initialized or cleared the first time the program starts. So a validation of the old NVM data content is made, to decide whether it is a first time startup.

During startup of a new firmware upgraded software a distinction is made between old NVM data and new NVM data, so that the old NVM data is left untouched, and the new NVM data is handled like if it was a first time startup.

To control the distinction between old and new NVM data there is a chain of descriptors in the NVM data area (segment HDATA) for the old NVM data to be used during startup. And there is a chain of descriptors in code area (segment CONST_NVM) for the new firmware to be used during startup.

4.6 C Coding Requirements

The following sections describe important C coding requirements and guidelines

4.6.1 Indirect function pointers when using code banking

The 500 series chip uses code banking, which requires important modifications to the application with respect to indirect function pointers. One example is the callback function used in ZW_SendData. The function below:

```
/*========================   cbVoidByte   ======================
**
**  Function:  stub for callback
**
**  Side effects: None
**
**--------------------------------------------------------------*/
static void cbVoidByte(BYTE b)
{
}
```

must be change to:

```
code const void (code * ZCB_cbVoidByte_p)(BYTE b) = &ZCB_cbVoidByte;
/*======================   ZCB_cbVoidByte   ====================
**
**  Function:  stub for callback
**
**  Side effects: None
**
**--------------------------------------------------------------*/
void
ZCB_cbVoidByte(BYTE b)
{
}
```

This change enables the linker to add ZCB_cbVoidByte to the interbank call table. Another problem is that the static attribute is applied to the ZCB_cbVoidByte function, which must be removed. The linker cannot generate an interbank call table entry for non-public (static) functions. ZCB_ added for easy identification of the indirectly called functions. Forgetting above modification is not caught by compiler/linker and results in unpredicted behavior such as a non-responding application.

For details refer to: http://www.keil.com/support/docs/2486.htm

4.6.2 Testing for generic null pointers

Keil C51 has both generic and memory specific pointers. The memory specific pointers are a proprietary extension to C and do not follow standard rules. A memory specific pointer with value 0x0000 converted to a generic point will get the value 0xFF0000, 0x010000 or similar, depending on the type of memory specific pointer. Therefore, testing the generic pointer for NULL-ness with ‘if(ptr)’ will fail. Instead use the macros NON_NULL(p) and IS_NULL(p) to test the pointer. An example:

```
void
func(BYTE *ptr)
{
    if(NON_NULL(ptr)  // if(ptr) is wrong!!
    {
        *p = 42;
    }
}
```

For a discussion of this problem see http://www.keil.com/forum/3443 and http://www.keil.com/support/docs/2630.htm.

4.6.3 Function pointers must be code-specific

Function pointers MUST always be declared as code specific pointer. This can easily be done with the VOID_CALLBACKFUNC() macro:

```
VOID_CALLBACKFUNC(funcptr)(BYTE b);
```

For function pointers, this will avoid the problem described in section 4.6.2. Despite this, section 4.6.2 is still recommended for function pointer.

4.6.4 Code Space Shortage

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

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

REFERENCES

[1] Silicon Labs, INS12303, Instruction, Z-Wave 500 Series Developer's Kit v6.51.10 Contents.
[2] Silicon Labs, INS13050, Instruction, Z-Wave 500 Series Developer's Kit v6.61.01 Contents.
[3] Silicon Labs, INS13477, Instruction, Z-Wave 500 Series Developer's Kit v6.71.01 Contents.
[4] Silicon Labs, APL12444, Instruction, Porting Z-Wave Appl. SW from ZW0301 to 500 Series.
[5] Silicon Labs, APL12445, Application Note, Porting Z-Wave Appl. SW from 400 to 500 Series.
[6] Silicon Labs, INS13448, Instruction, Z-Wave 500 Series Application Programmers Guide v6.70.00.
[7] Silicon Labs, INS13427, Instruction, Z-Wave Plus Application Framework v6.70.0x.
[8] Silicon Labs, SDS11846, Software Design Specification, Z-Wave Plus Role Types Specification.
[9] Silicon Labs, SDS11847, Software Design Specification, Z-Wave Plus Device Types Specification
[10] Silicon Labs, SDS13781, Software Design Specification, Z-Wave Application Command Class Specification.
[11] Silicon Labs, SDS13782, Software Design Specification, Z-Wave Management Command Class Specification.
[12] Silicon Labs, SDS13783, Software Design Specification, Z-Wave Transport-Encapsulation Command Class Specification.
[13] Silicon Labs, SDS13784, Software Design Specification, Z-Wave Network-Protocol Command Class Specification.
[14] Silicon Labs, SDS13548, Software Design Specification, List of defined Z-Wave Command Classes.
[15] Silicon Labs, SDS10242, Software Design Specification, Z-Wave Device Class Specification.

INDEX