Instruction

Working in 400 Series Environment User Guide v6.02.00

Document No.:	INS11709
Version:	4
Description:	Describes the 400 Series software development environment with respect to normal and development mode.
Written By:	JFR;CHL;EFH;MVO
Date:	2012-05-25
Reviewed By:	SSE;TRO;PSH;CHL;MAM;JAD;EFH
Restrictions:	Partners Only

Approved by:

Date	CET	Initials	Name	Justification
2012-05-25	11:28:33	NTJ	Niels Thybo Johansen	

CONFIDENTIAL

REVISION RECORD

Doc. Ver.	Date	By	Pages affected	Brief description of changes
1	20110209	MVO JFR EFH	ALL	Initial draft
2	20110405	JFR	4.2.2 4.2.3 & 4.2.4	Clarified development process Added boundary cases
3	20120113	JFR	6.3	uVision Project file generation
3	20120117	EFH	6.2	The directory structure generated by the build process corrected
4	20120524	JFR	6.4	Debugging in the uVision environment

Table of Contents

Table of Figures

1 ABBREVIATIONS

Abbreviation	Explanation
OTP	One Time Programmable
SDK	Z-Wave Software Developer's Kit

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 400 Series Z-Wave module for the different code memory modes. Refer to [1] regarding Z-Wave sample applications hosted on the 400 Series Z-Wave module.

2.2 Audience and prerequisites

The audience is R&D software application programmers. The programmer should be familiar with the PK51 Keil Development Tool Kit for 8051 micro controllers and the GNU make utility.

3 BUILDING APPLICATION CODE

All the sample applications for the 400 Series Z-Wave Single Chip contains source code and makefiles that allows the developer to modify and compile the applications without modifying makefiles, etc. All 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.

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 in lots of variants with 5 varying parameters:

- FREQUENCY
- CODE_MEMORY_MODE
- LIBRARY
- HOST_INTERFACE
- SENSOR_TYPE

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

For every one of these parameters, there are 3 different ways to set which one you want. 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.

3.1 Specifying which subset of "everything", you wan't to build:

If you do not specify anything after the “mk” command, everything will be built.

The command “mk help” will give you a hint on what you can ask for:

```
...\Product\dev_ctrl>mk help
SYNOPSIS:
> MK ["FREQUENCY=EU"] ["CODE_MEMORY_MODE=normal"] ....
-
List of frequencies:        US EU ANZ HK MY IN JP JP_DK
List of code memory modes: normal devmode devmode_patch starter_devmode starter_devmode_patch
```

For every parameter, you can specify a single variant to build for in 3 different ways:

- 1. By specifying the code memory mode in your command line, like:
  > mk “CODE_MEMORY_MODE=devmode_patch” ....

- 2. By setting the parameter in the Makefile (it is prepared):
  CODE_MEMORY_MODE:=normal

- 3. Alternativly you can do the same by setting your environment from the command line with:
  > SET CODE_MEMORY_MODE=normal

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

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

4 MODES OF OPERATION

The 400 Series Z-Wave Single Chip build environment is different compared to previous 100/200/300 Series Z-Wave Single Chips since the ASIC has OTP memory instead of Flash memory in the code space. However, the 400 Series Z-Wave Single Chip supports a Development Mode (from now on called Development Mode) enabling application development in SRAM. The supported modes are shown on the figure below.

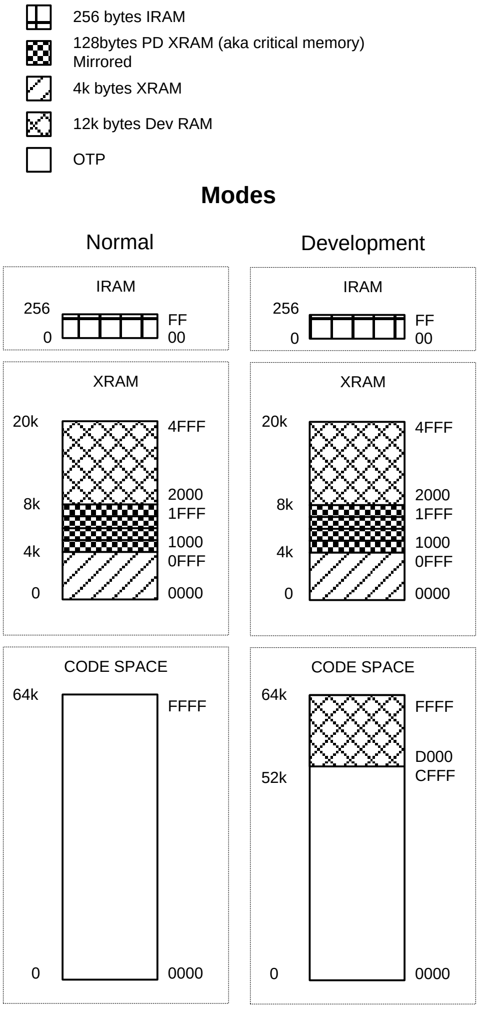


Figure 1, 400 Series Z-Wave Single Chip memory map in the different modes

In Normal Mode the code in located in OTP memory only.

Development Mode is used during application development. A 12kB SRAM is mapped into the upper part of the code space to allow modification of the code.

SDK sample applications supports both Normal and Development Mode enabling developers to easily start on new applications. However, read the following sections carefully because especially Development Mode contains a number of limitations and pitfalls.

4.1 Normal Mode

In Normal Mode all firmware code is located in OTP memory and is therefore targeted for final product runs typically. Since there is no way to erase the OTP code memory the developer must pick a new 400 Series Z-Wave Single Chip when running Normal Mode and a code change is required.

To ease the development process and re-use 400 Series Z-Wave Single Chip during several development coding/testing loops you can use Development Mode during development of the Z-Wave application.

4.2 Development Mode

Use Development Mode when creating a new and/or modifying an existing Z-Wave application. The same 400 Series Z-Wave Single Chip can be re-used as long as the Z-Wave application development can be constrained to the 12kB SRAM and OTP content can be kept unchanged. This shortens the development cycle considerably and reduces cost by avoiding replacement of chip.

4.2.1 Introduction

All sample applications in the SDK are prepared for application development in two different Development Modes supporting a SRAM based patch of all application functions in OTP.

- devmode
- starter_devmode

The devmode is best, when you are nearly finished with your application, or if you are making an application very similar to one of the sample applications in the SDK.

When you start your project from scratch, it is best starting with the starter_devmode. In this mode the patchable OTP target contains the entire Z-Wave library and an empty application with patchable versions of all the mandatory application functions shown below:

- ApplicationInitHW
- ApplicationInitSW
- ApplicationTestPoll
- ApplicationPoll
- ApplicationCommandHandler
- ApplicationNodeInformation
- ApplicationSlaveUpdate eller ApplicationControllerUpdate
- ApplicationSlaveCommandHandler (Bridge only)
- ApplicationSlaveNodeInformation (Bridge only)

You can still use any of the sample applications as a starting point. Start with the one with the library, which fits your applications needs.

Build the starter_devmode (OTP) target first. This can be programmed into the OTP of your first chip. If you can develop all of your application in 12K code memory, then you can use this first chip for the rest of your development work. All the sample applications in the SDK can be held in the 12K memory for development work.

When you have written your application, build the starter_devmode_patch target. This can be programmed into the SRAM of your starter_devmode development chip, and be run in development mode. All the sample applications in the devkit are prepared for this.

As your development work progresses, you can put more and more of your application into the OTP. Near the end of your project, you will see an advantage of using the devmode targets (without "starter_"). There is no way in between the two devmodes, devmode and starter_devmode. It will always be a little puzzle to bring you from your first attempts with the empty application, to the finished project in OTP.

As the last thing in your project, you build your project for normal mode, where all code is unpatchable in OTP. This is the code, which you should release for your final product.

If you use the patch macros declared in $(ZWLIBROOT)\include\patch.h like they are used in the Z-Wave sample applications, it will be possible to make source code compatible for working in all code memory modes.

4.2.2 Getting started…

We recommend using starter_devmode hex files because this option provides the greatest flexibility during development of a new application. The whole application can be changed because it is situated in the SRAM part mapped into OTP. The steps are as follows when selecting the LED Dimmer application as a starting point:

1. Developer's migrating from SDK's prior to 6.0x must change the code according to the steps in [2]. SDK 6.01.00 must also incorporate the standard patch system macro defines introduced in 6.01.01.

2. Make a new file called LEDdim_patch.c and copy entire content of LEDdim.c into it.

3. Build application in Development Mode and relevant frequency.

4. Program chip:
   leddimmer_ZW040x_ANZ_starter_devmode.hex to OTP
   leddimmer_ZW040x_ANZ_starter_devmode_patch_RAM.hex to SRAM mapped into OTP

5. Check that application runs as expected.

6. Make the necessary changes in LEDdim_patch.c

7. Build application in Development Mode and relevant frequency.

8. Program chip in Development Mode:
   leddimmer_ZW040x_ANZ_starter_devmode_patch_RAM.hex to SRAM mapped into OTP

9. Check application and in case additional changes are needed jump back to point 6.

10. Copy entire content of LEDdim_patch.c into LEDdim.c.

11. Build application in Normal Mode and relevant frequency.

12. Insert a new chip because OTP part have changed.

13. Program chip in Normal Mode:
    leddimmer_ZW040x_ANZ.hex to OTP

14. Check that application runs as expected.

4.2.3 Hitting 12KB SRAM mapped into OTP boundary

When LEDdim_patch.c exceeds the 12KB SRAM mapped into OTP boundary then it is necessary to reduce its size and use only devmode hex files going forward. The steps are now as follows:

1. Copy entire content of LEDdim_patch.c into LEDdim.c.
2. Remove some of the functions completely from LEDdim_patch.c that can stay unchanged in the following application development. Goal is to get under the 12KB boundary again.
3. Build application in Development Mode and relevant frequency.
4. If not under the 12KB boundary jump back to point 2.
5. Insert a new chip because OTP part have changed.
6. Program chip:
   leddimmer_ZW040x_ANZ_devmode.hex to OTP
   leddimmer_ZW040x_ANZ_devmode_patch_RAM.hex to SRAM mapped into OTP
   Notice: Do not use starter_devmode anymore!
7. Check that application runs as expected.
8. Make the necessary changes in LEDdim_patch.c
9. Build application in Development Mode and relevant frequency.
10. Program chip in Development Mode:
    leddimmer_ZW040x_ANZ_devmode_patch_RAM.hex to SRAM mapped into OTP
    Notice: Do not use starter_devmode anymore!
11. Check application and in case additional changes are needed jump back to point 8.
12. Copy content of LEDdim_patch.c into LEDdim.c but remember to preserve the functions that only exist in LEDdim.c.
13. Build application in Normal Mode and relevant frequency.
14. Insert a new chip because OTP part have changed.
15. Program chip in Normal Mode:
    leddimmer_ZW040x_ANZ.hex to OTP
16. Check that application runs as expected.

4.2.4 Hitting 52KB OTP boundary

When LEDdim.c and library exceeds the 52KB OTP boundary then it is necessary to introduce empty functions in LEDdim.c to reduce its size. The empty functions must then exist in LEDdim_patch.c including all the code to get it executed. Again it is necessary to use devmode hex files.

4.2.5 The patch macros

In the include file $(ZWLIBROOT)\include\patch.h are declared some very central macros for the patch system. Using these macros makes it possible to have identical source files for all the 5 different code memory modes. This makes it easier to synchronize the source codes with respect to equality of functi on headers between non-patch (OTP) and patch (SRAM).

- PATCH_FUNCTION_NAME
  This macro is used for adjusting a function name to be used as both patchable and as a patch.

- PATCH_FUNCTION_NAME_STARTER
  This macro is a variant of the above only to be used for the mandatory application functions referenced by the library.

- PATCH_FUNCTION_NAME_WRAPPER
  This macro is a variant of the above only to be used in the wrapper module …\util_func\starter.c used for starter_devmode targets.

- PATCH_TABLE_ENTRY
  This macro is used for generating a patch table entry for a function. It shall be put in every patchable function, and the patch for it. It shall be inserted before any code generating statements in the function, and after any declarations of variables. Observe that you can't initialize a local variable in the same statement as the declaration here.

- PATCH_TABLE_ENTRY_STARTER
  This macro is a variant of the above only to be used for the mandatory application functions referenced by the library.

- PATCH_TABLE_ENTRY_WRAPPER
  This macro is a variant of the above only to be used in the wrapper module …\util_func\starter.c used for starter_devmode targets.

- PATCH_VARIABLE
  This macro is used for declaration of global variables referenced by both non-patch (OTP) code and patch (SRAM) code.

- PATCH_VARIABLE_STARTER
  This macro is a variant of the above only to be used for variables nodeInfo and txBuf

4.2.6 Limitations in Development Mode

Remember the following important limitations when working with patchable functions in Development Mode:

- A patchable function, and the patch function for it, must have exactly the same declaration. This constraint goes for both the functions parameters and the functions local variables. You can have all the functions you like in the devmode_patch target. They do not need to be existent as a patchable function in OTP (but of course they can't be called from OTP code directly. They are not known). If you are working in starter_devmode this will be obvious, because the whole application resides in the starter_devmode_patch. Also, you can have all the functions you like in the patchable (OTP) part of your program. They do not need to have a corresponding patch function. But it is a good idea to synchronize the source files towards the end of your project. Then they can be copied back and forth between source.c and source_patch.c

- The functions must be declared reentrant to make the parameters and local variables allocated on the "Reentrant Stack". Also because of this, static variables are not allowed in these functions.

- Check carefully the variable data memory layout in the map files. They must not overlap.

- The patchable function and corresponding patch function must NOT be empty.
- If you make a time critical function patchable, you can experience problems, if you have too many other patch functions. I.e. if the patch-table grows too large, it can be too time consuming to search the table. So be careful with patching time critical functions.
- Interrupt vectors must be located in the OTP part of the program. If you are working in starter_devmode, and your application in the patch part of the program references a part of the Z-Wave library, which contains interrupt functions, which are not used by the empty application in OTP, Then you can experience problems. You then need to force load these Z-Wave library modules during linking of the OTP part of the program. This is done by assigning the module names in question to a variable called FORCE_LOADED_LIBRARY_MODULES in your Makefile, when building for starter_devmode. An example can be found in the Makefile for the application …\Product\Simple_AV_Remote:
  # Force loading of used library modules, which contains interrupt functions
  FORCE_LOADED_LIBRARY_MODULES:=ZW_PHY_KEYPAD_SCANNER_040X.
  This can be a comma delimited list of modules.

4.2.7 Patch and non-patch (RF parameters)

One more reason to work in Development Mode is when you need to adjust RF parameters in an existent program with the ZWaveProgrammer. In a pure normal mode OTP program you can't change anything, but in Development Mode you can change the RF parameters located near the top of the memory map in SRAM.

The patchable devmode targeted OTP program can be executed in normal mode. Just as the normal mode targeted OTP program. In both cases the RF parameters are located near the top of the memory map, and can't be changed.

If you need to change the RF parameters with the ZWaveProgrammer, and you don't have a patch for your program, then you need to program (write) a small file, only containing these RF parameter, to the development SRAM, and execute the program in Development Mode. This .hex file, <appl>_ZW040x_y_devmode_RAM.hex, is part of the build result, and is located as described in paragraph 6.2.

5 THE PATCH SYSTEM

The patch system consists of a patchcheck routine in the Z-Wave library, and macros inserted in the beginning of the patchable code (OTP code) and patch code (SRAM code). The inserted macros in the code will build a 'patch-table' for the 'patch-check' routine. Figure 2 shows an example of a patch system for a 400 Series Z-Wave Single Chip in Development Mode.

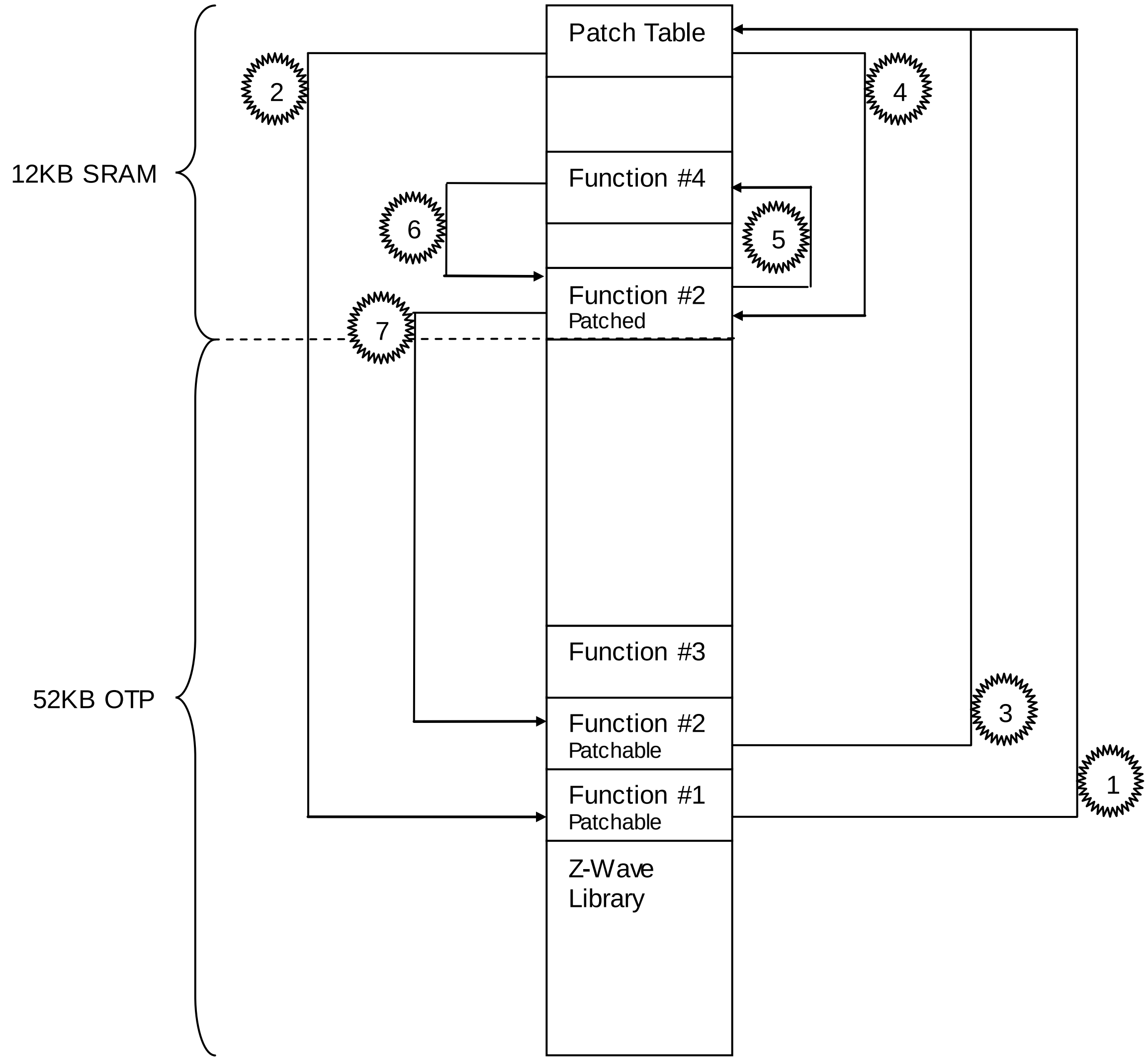


Figure 2, An example of the patch system execution process

The figure shows an example comprising of a Z-Wave library and three application functions residing in OTP. Modifying functions #1 and #2 as patchable allow patched functions in SRAM as an alternative. Function #3 is not enabled as patchable in the example. To make function #3 patchable will require a change in the source code, build, and download of the hex file into a new 400 Series ASIC. Function #4 is an application specific function that is only located in SRAM and that is only called from function #2. Note that functions, that are only located SRAM, can only be called from other functions in SRAM.

The code in the example is executed as described in the following:

1. The system starts to execute the main loop, which is located in the Z-Wave library. When various Z-Wave protocol tasks are completed, the application calls functions #1, #2 and #3 successively from the main loop. When the processor is executing function #1 it will discover that this function is patchable. The processor will then jump to the patch table in the SRAM to search for an alternative function in the SRAM.

2. As the patch table does not contain an entry for function #1, the processor will return to the start address of the OTP based function #1. The processor will then executes the function #1 in OTP memory and then return to the main loop.

3. When the processor is executing function #2 it will again discover that this function is patchable. As a result it will again jump to the patch table in the SRAM to search for an alternative function in the SRAM.

4. As the processor finds an alternative function in the patch table it will jump to the start address of function #2 in SRAM memory

5. The processor now executes function #2 in the SRAM memory where function #4 will be called

6. The processor returns to function #2

7. Then the processor returns to the main loop skipping function #2 in OTP memory. Finally, the program executes the non-patchable function #3 in OTP without any additional search overhead in the patch table.

The sample applications support both Normal Mode and Development Mode. A flag in the make files determine the mode to select, which is the only change the customer have to make. In Normal Mode the patch system is ignored and in Development Mode the patch system is enabled.

5.1 Patchable code example

For example, to make the application function, ApplicationInitSW, patchable it only requires minor changes in the source code as described in the following. First, you must insert some definitions for the patch system as the very first in your C-file:

```
#ifdef PATCH_ENABLE
/***************************************************************************/
/* Include assembly MACRO definitions for patch insertions.                */
/*                                                                         */
/* Define $SET (MAKE_PATCHABLE_CODE) for making patchable code destined   */
/* for OTP or ROM memory.                                                  */
/* Undefine $RESET (MAKE_PATCHABLE_CODE) for making code containing patch  */
/* code destined for RAM or FLASH memory.                                 */
/***************************************************************************/
#if defined(WORK_PATCH) || defined(STARTER_PATCH)
/* Making code containing patch code destined for development RAM memory. */
#pragma asm
$RESET (MAKE_PATCHABLE_CODE)
$INCLUDE (ZW_patch.inc)
#pragma endasm
/* Rename CODE class to CODE_PATCH */
#pragma userclass (code = PATCH)
/* Rename CONST class to CONST_PATCH */
#pragma userclass (const = PATCH)
/* Rename XDATA class to XDATA_PATCH */
#pragma userclass (xdata = PATCH)
#else
/* Making patchable code destined for OTP or ROM memory.                  */
#pragma asm
$SET (MAKE_PATCHABLE_CODE)
$INCLUDE (ZW_patch.inc)
#pragma endasm
#endif /* elsif defined(WORK_PATCH) || defined(STARTER_PATCH) */
#endif /* PATCH_ENABLE */
```

Second, modify the function ApplicationInitSW in the <application>.c file and in the <application>_patch.c[1] file before compiling and downloading them to respectively OTP and development SRAM::.

```
/*==========================   ApplicationInitSW   ========================

**    Initialization of the Application Software
**
**    This is an application function example
**
**--------------------------------------------------------------------------*/
BYTE                      /*RET  TRUE       */
PATCH_FUNCTION_NAME(ApplicationInitSW)( void ) /* IN   Nothing   */
#ifdef PATCH_ENABLE
reentrant
#endif
{
#ifdef PATCH_ENABLE
#pragma asm
PATCH_TABLE_ENTRY(ApplicationInitSW)
#pragma endasm
#endif
  .
  . /* Do the things which you thought was right in the first place */
  .
  return(TRUE);
}
```

Initialized global variables in the application module must be declared in the following way:

```
PATCH_VARIABLE BYTE receivedReport
#ifndef WORK_PATCH
 = TRUE
#endif
;
```

[1] <application>.c file and <application>_patch.c file can be identical

6 DEVELOPMENT TOOL SETUP AND EXECUTION

The developer can choose from two methods to compile and download firmware to the 400 Series Z-Wave single Chip:

- Use the make command line tool and the Z-Wave Programmer GUI tool
- Use Keil uVision development GUI

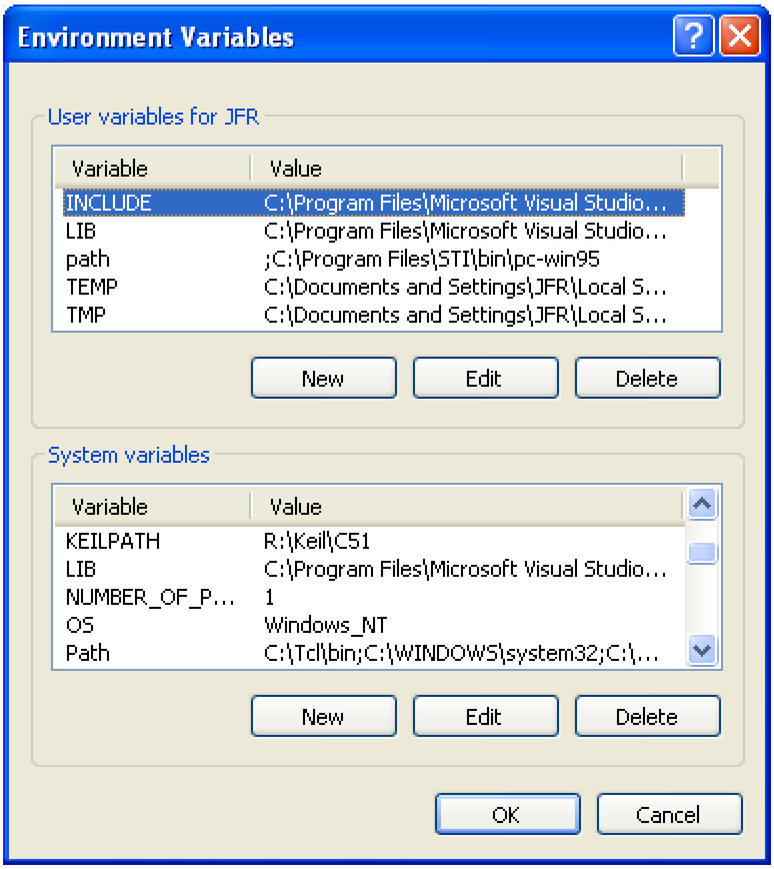
6.1 Environment Setup

A couple of environment variables must be defined before the sample applications can be build on the Z-Wave Developer's Kit:

- KEILPATH
- TOOLSDIR

The procedure on a PC using Windows XP is performed as follows:

1. Select **Start**, **Control Panel** and **System**
2. Select **Advanced** tab and activate the **Environment Variables** button



User variables for JFR

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

System variables

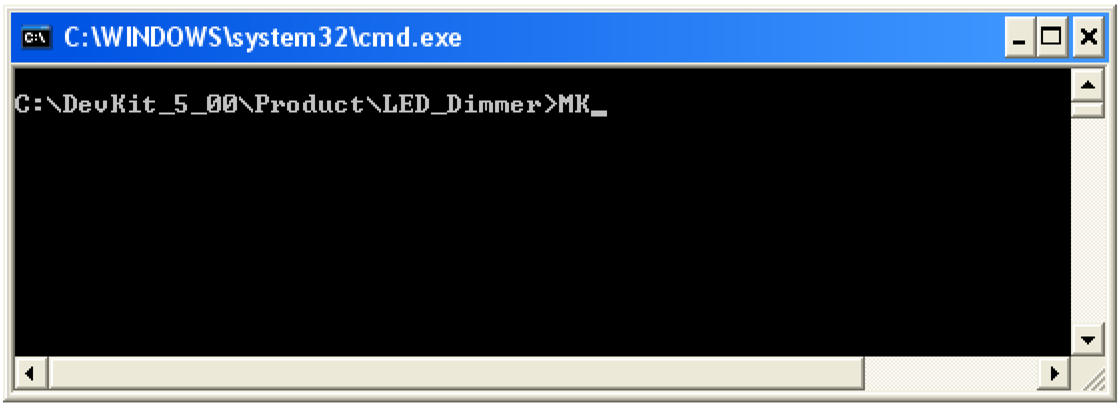
Variable	Value
KEILPATH	R:\Keil\C51
LIB	C:\Program Files\Microsoft Visual Studio...
NUMBER_OF_P...	1
OS	Windows_NT
Path	C:\Tcl\bin;C:\WINDOWS\system32;C:\...

Figure 3, Configuring environment variables

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

4. In the **Variable name** textbox enter **KEILPATH** (use capital letters because Windows XP is case sensitive)
5. In the **Variable value** textbox enter **C:\KEIL\C51** and activate the **OK** button
6. Under **System variables** activate the **New** button
7. In the **Variable name** textbox enter **TOOLSDIR** (use capital letters because Windows XP is case sensitive)
8. In the **Variable value** textbox enter **C:\Devkit_5_00\TOOLS** and activate the **OK** button

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



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

Figure 4, Building sample applications

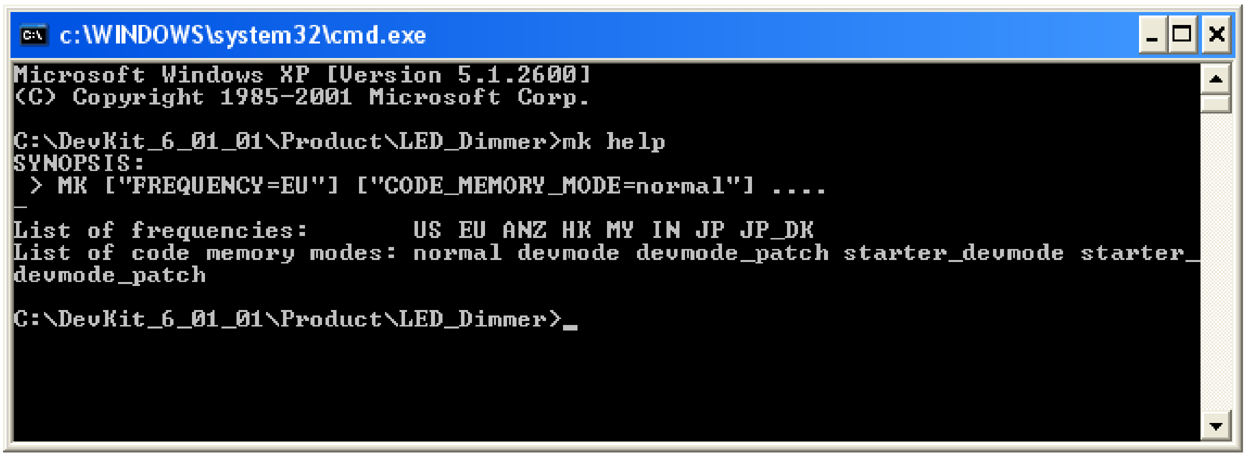
Remember to use upper case in **KEILPATH, KEIL_LOCAL_PATH** and **TOOLSDIR** when using Windows XP, because this operating system is case sensitive. If the environment variables are not defined then MK.BAT will prompt the user to define them.

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

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

6.2 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, Routing Slave, etc.) and RF frequencies (ANZ/EU/HK/IN/JP/MY/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.



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\DevKit_6_01_01\Product\LED_Dimmer>mk help
SYNOPSIS:
 > MK ["FREQUENCY=EU"] ["CODE_MEMORY_MODE=normal"] ....
_
List of frequencies:        US EU ANZ HK MY IN JP JP_DK
List of code memory modes: normal devmode devmode_patch starter_devmode starter_
devmode_patch

C:\DevKit_6_01_01\Product\LED_Dimmer>_
```

Figure 5, Possible sample application targets

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

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

```
- <appl>
    - build
        - <appl>_ZW040x_<frequency>
            - list                                         - contains list files
            - rels                                         - contains object files and map files
            extern_eep.hex                                 - EEPROM
            <appl>_ZW040x_<frequency>.hex                  - OTP (Normal Mode )
        - <appl>_ZW040x_<frequency>_devmode
            - list                                         - contains list files
            - rels                                         - contains object files and map files
            extern_eep.hex                                 - EEPROM
            <appl>_ZW040x_<frequency>_devmode.hex          - OTP (Development Mode )
            <appl>_ZW040x_<frequency>_devmode_patch_RAM.hex - SRAM (Development Mode )
        - <appl>_ZW040x_<frequency>_starter_devmode
            - list                                         - contains list files
            - rels                                         - contains object files and map files
            extern_eep.hex                                 - EEPROM
            <appl>_ZW040x_<frequency>_starter_devmode.hex  - OTP (Development Mode )
            <appl>_ZW040x_<frequency>_starter_devmode_patch_RAM.hex
                                                           - SRAM (Development Mode )
```

Actually, the makefile generates four hex-files, when working in Development Mode for the ZW040x target. The purpose of the hex-files is as follows:

<appl>_ZW040x_y_devmode.hex

Program this file into OTP memory in Development Mode.

This file is the result of the first linking. It is the program, which is prepared as patchable for further improvement and changes.

<appl>_ZW040x_y_devmode_patch.hex

This file is the result of the second linking. It is the patches to the original program above. This file is located in the directory …\build\Rels\

<appl>_ZW040x_y_devmode_patch_RAM.hex

Program this file into SRAM memory in Development Mode.

This file is the same as the previous. But it has been filtered so that it only contains contents destined for the SRAM memory (in Development Mode). However, if there were any RAM targeted program contents from the first linking, then it is also merged into this file.

<appl>_ZW040x_y_devmode_RAM.hex

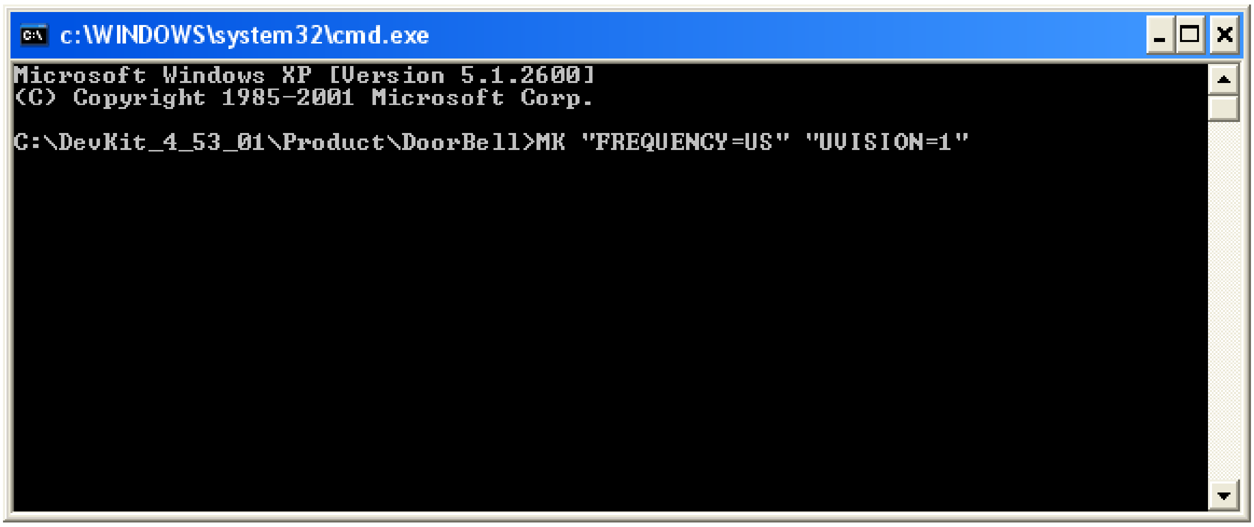
This file contains the frequency setup for the patchable program from the first linking. It will be needed for running the program in Development Mode without loading a patch.
This file is programmed into SRAM with the programmer, when you do not want to use a patch.
This file is located in the directory …\build\Rels\

6.2.1 Makefile

The file Makefile is initially read by the make tools that is 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.

6.3 uVision project file generation

The Developer's Kit contains tools for generation of Keil uVision4 project files for the embedded sample application based on a 400 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.



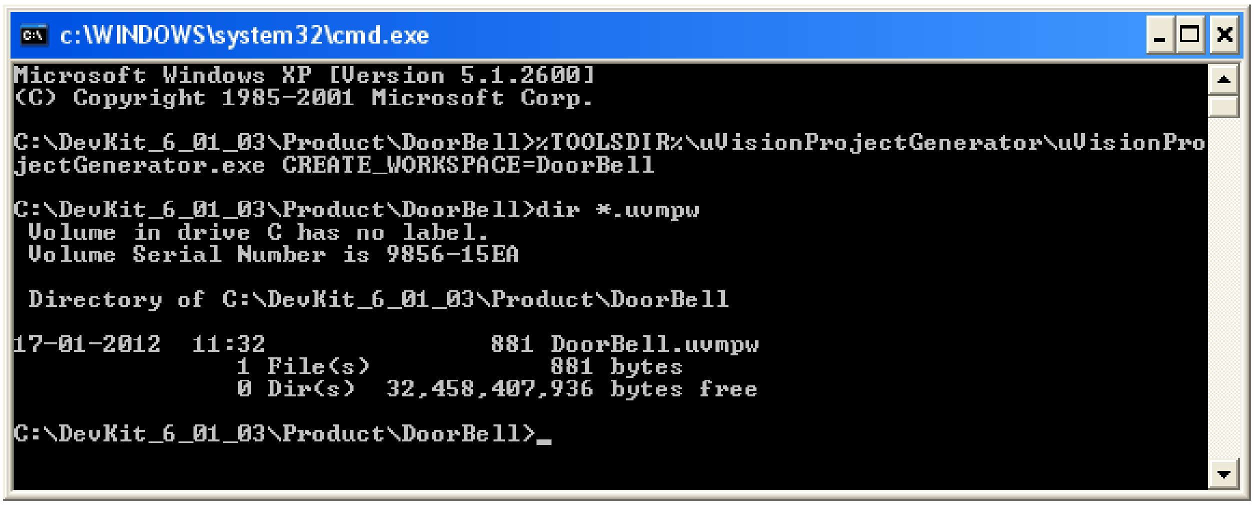
```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\DevKit_4_53_01\Product\DoorBell>MK "FREQUENCY=US" "UVISION=1"
```

Figure 6, Building sample applications and uVision project files

It is also possible to generate the uVision multi-project file (*.uvmpw) by calling

%TOOLSDIR%\uVisionProjectGenerator\uVisionProjectGenerator.exe CREATE_WORKSPACE=<workspace_name>

For example generate DoorBell uVision multi-project file as follows



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\DevKit_6_01_03\Product\DoorBell>%TOOLSDIR%\uVisionProjectGenerator\uVisionPro
jectGenerator.exe CREATE_WORKSPACE=DoorBell

C:\DevKit_6_01_03\Product\DoorBell>dir *.uvmpw
 Volume in drive C has no label.
 Volume Serial Number is 9856-15EA

 Directory of C:\DevKit_6_01_03\Product\DoorBell

17-01-2012  11:32               881 DoorBell.uvmpw
               1 File(s)            881 bytes
               0 Dir(s)  32,458,407,936 bytes free

C:\DevKit_6_01_03\Product\DoorBell>_
```

Figure 7, Generating uVision multi-project file

6.3.1 Normal Mode project

<appl>_ZW040x_US.uvproj: this project is the same as for old chip series. This project generates the HEX file to be written to the OTP of the ZW040x and to be run in the Normal Mode. No patch system is being used.

6.3.2 Development Mode project

<appl>_ZW040x_US_devmode.uvproj: this project generates the basic, patchable hex file, which is loaded to the OTP memory of the ZW040x in Development Mode.

<appl>_ZW040x_US_devmode_patch.uvproj: this project generates the patch hex file, which is loaded to the SRAM of the ZW040x in Development Mode.

To work with Development Mode, you must:

1) Open **<appl>.uvmpw**.
2) Activate the **<appl>_ZW040x_US_devmode** project
3) Build the project.
4) Press the “Download to Flash Memory” button (*Flash->Download*), and **<appl>_ZW040x_US_devmode.hex** will be written to the OTP.
5) Then activate the **<appl>_ZW040x_US_devmode_patch** project.
6) Build **<appl>_ZW040x_US_devmode_patch** project.
7) After this, you must press “Download to Flash Memory” button (*Flash->Download*), and **<appl>_ZW040x_US_devmode_patch_RAM.hex** will be written to the SRAM, and Development Mode will be activated.
8) Make changes in the **<appl>_patch.c**
9) Go to step (6).

Before using this scheme, you must fulfill these requirements:

1) At Project -> Options for target Release -> Utilities tab: set the COM port number to the port where Z-Wave Programmer ZDP03A board is connected.
2) If Keil uVision is not installed in default folder (C:\Keil): in *Project -> Manage -> Components, Environments, Books -> Project Components* tab: remove C51FPL.lib and add it from user’s Keil installation (keil folder\C51\LIB\ C51FPL.lib). Move the library using up and down buttons, so that it is placed between zw_controller_portable…. and init_vars.obj

6.4 Debugging with uVision IDE

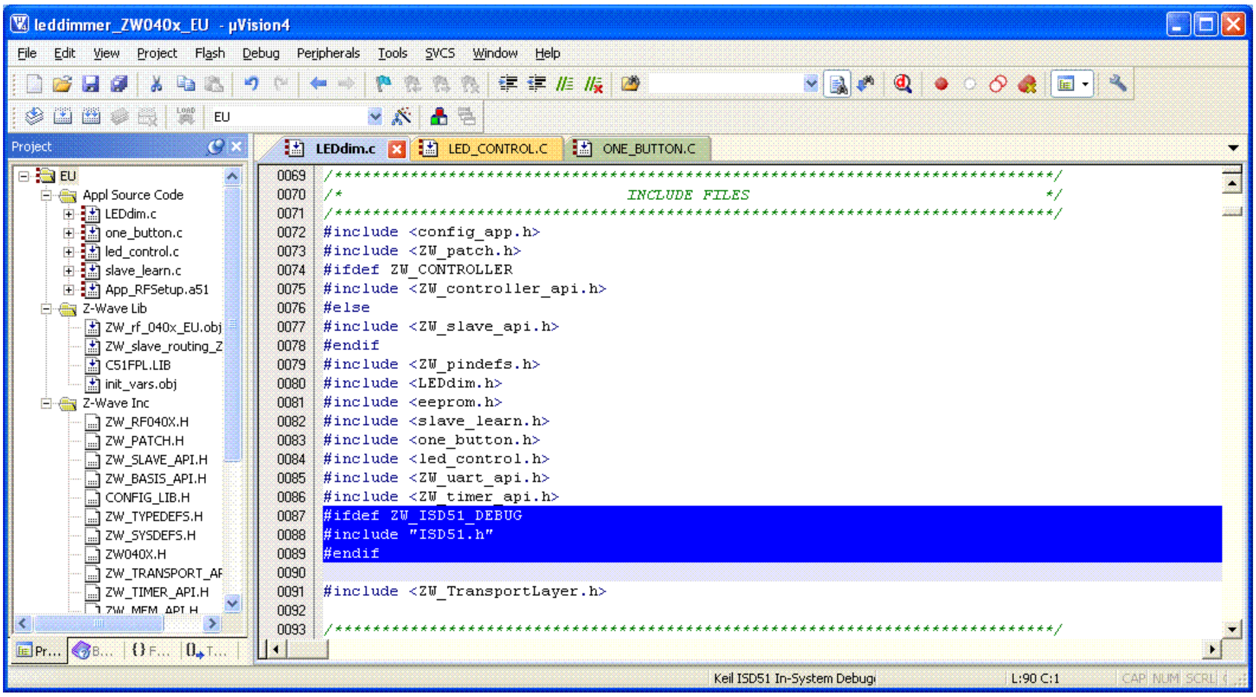
The uVision4 IDE support debugging of embedded applications including utilization of breakpoints supported by the 400 Series chip. The steps to enable debugging are as follows:

1. Make an .uvproj file for your application:

...\Product\LED_Dimmer>make FREQUENCY=EU CODE_MEMORY_MODE=normal BOARD=ZDP03A MAKESCHEME=NO_SCHEME UVISION=1

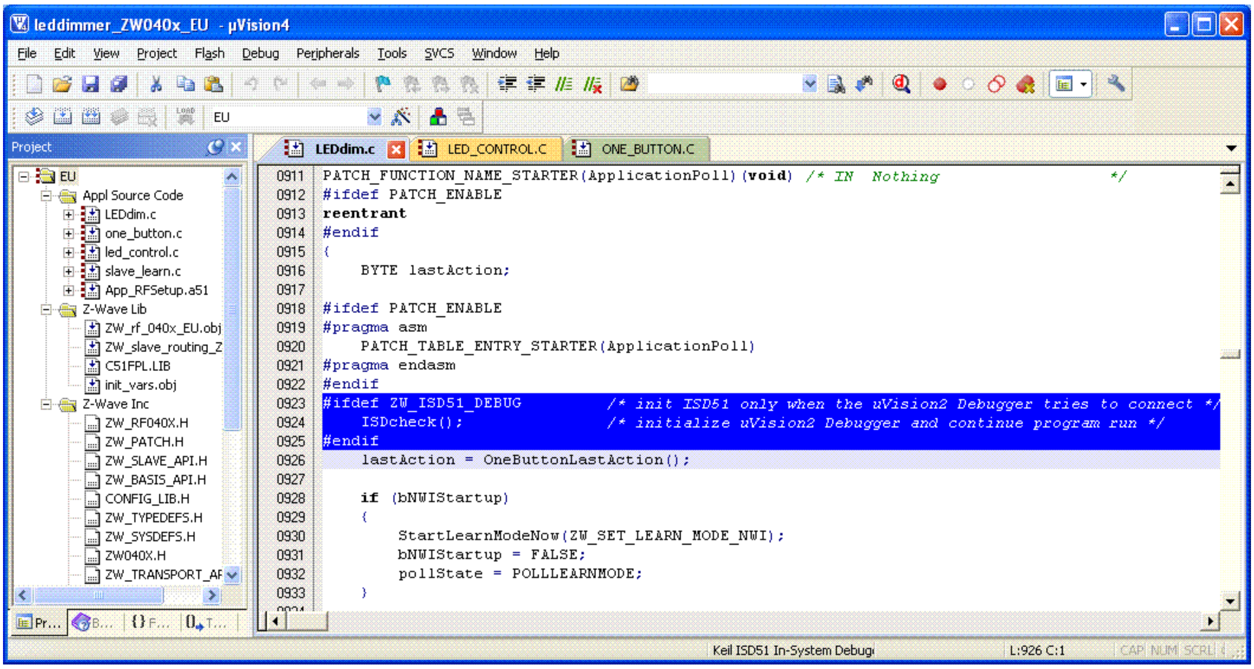
2. Open this uVision project in uVision4.

3. Add inclusion of ISD51.h into your applications main file like this:



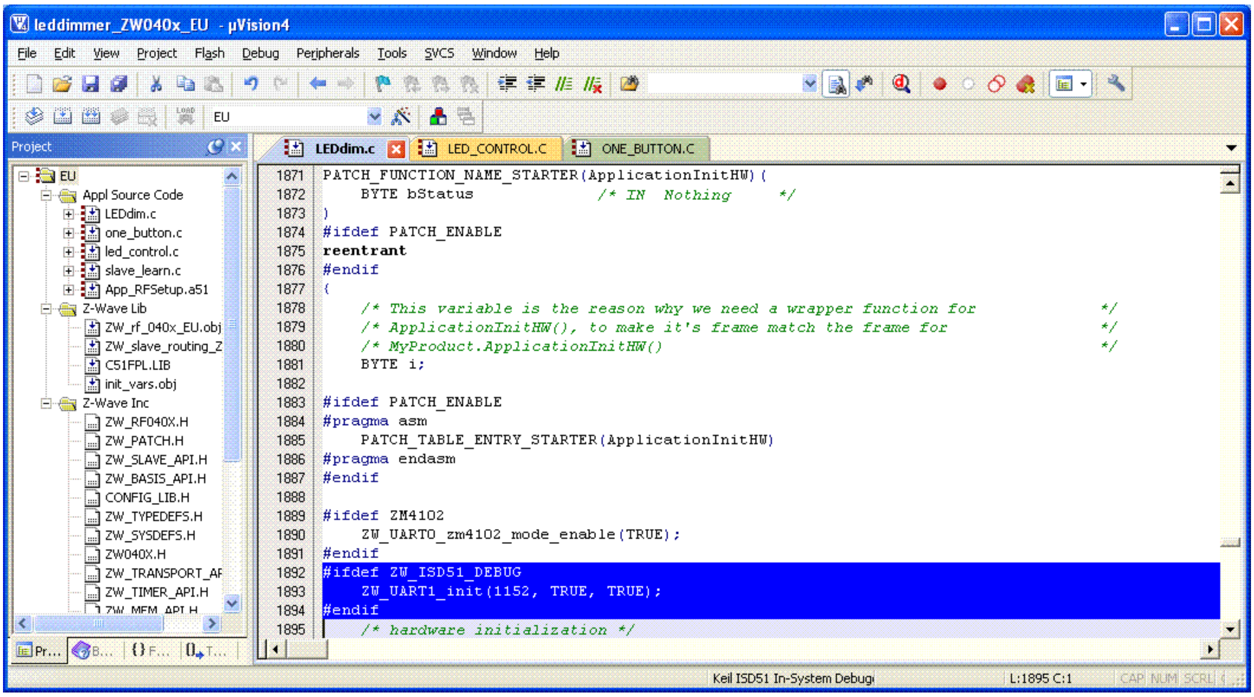
```
/****************************************************************************/
/*                              INCLUDE FILES                               */
/****************************************************************************/
#include <config_app.h>
#include <ZW_patch.h>
#ifdef ZW_CONTROLLER
#include <ZW_controller_api.h>
#else
#include <ZW_slave_api.h>
#endif
#include <ZW_pindefs.h>
#include <LEDdim.h>
#include <eeprom.h>
#include <slave_learn.h>
#include <one_button.h>
#include <led_control.h>
#include <ZW_uart_api.h>
#include <ZW_timer_api.h>
#ifdef ZW_ISD51_DEBUG
#include "ISD51.h"
#endif

#include <ZW_TransportLayer.h>

/****************************************************************************/
```

4. Add a polling statement into the main loop (in the beginning of ApplicationPoll() function) like this:



```
PATCH_FUNCTION_NAME_STARTER(ApplicationPoll)(void) /* IN  Nothing                */
#ifdef PATCH_ENABLE
reentrant
#endif
{
    BYTE lastAction;

#ifdef PATCH_ENABLE
#pragma asm
    PATCH_TABLE_ENTRY_STARTER(ApplicationPoll)
#pragma endasm
#endif
#ifdef ZW_ISD51_DEBUG           /* init ISD51 only when the uVision2 Debugger tries to connect */
    ISDcheck();                 /* initialize uVision2 Debugger and continue program run */
#endif
    lastAction = OneButtonLastAction();

    if (bNWIStartup)
    {
        StartLearnModeNow(ZW_SET_LEARN_MODE_NWI);
        bNWIStartup = FALSE;
        pollState = POLLLEARNMODE;
    }
```

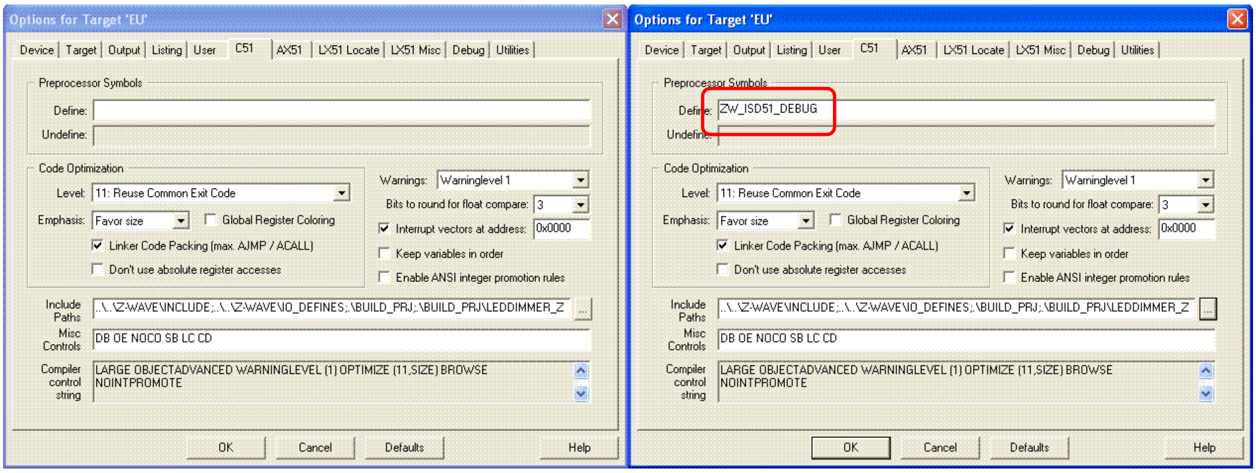
5. Add an initialization statement for UART1 in the ApplicationInitHW() function like this:



```
PATCH_FUNCTION_NAME_STARTER(ApplicationInitHW)(
    BYTE bStatus                /* IN  Nothing     */
)
#ifdef PATCH_ENABLE
reentrant
#endif
{
    /* This variable is the reason why we need a wrapper function for           */
    /* ApplicationInitHW(), to make it's frame match the frame for              */
    /* MyProduct.ApplicationInitHW()                                           */
    BYTE i;

#ifdef PATCH_ENABLE
#pragma asm
    PATCH_TABLE_ENTRY_STARTER(ApplicationInitHW)
#pragma endasm
#endif

#ifdef ZM4102
    ZW_UART0_zm4102_mode_enable(TRUE);
#endif
#ifdef ZW_ISD51_DEBUG
    ZW_UART1_init(1152, TRUE, TRUE);
#endif
    /* hardware initialization */
```

6. Add the options necessary for debugging like this: (The original is to the left and the new to the right)

Options for Target 'x' -> C51 -> Preprocessor Symbols -> Define: add "ZW_ISD51_DEBUG"


Options for Target 'EU'
Device | Target | Output | Listing | User | C51 | AX51 | LX51 Locate | LX51 Misc | Debug | Utilities
Preprocessor Symbols
Code Optimization
OK
Cancel
Defaults
Help

Define: ZW_ISD51_DEBUG

Undefine:

Level: 11: Reuse Common Exit Code

Emphasis: Favor size

□ Global Register Coloring

☑ Linker Code Packing (max. AJMP / ACALL)

□ Don't use absolute register accesses

Warnings: Warninglevel 1

Bits to round for float compare: 3

☑ Interrupt vectors at address: 0x0000

□ Keep variables in order

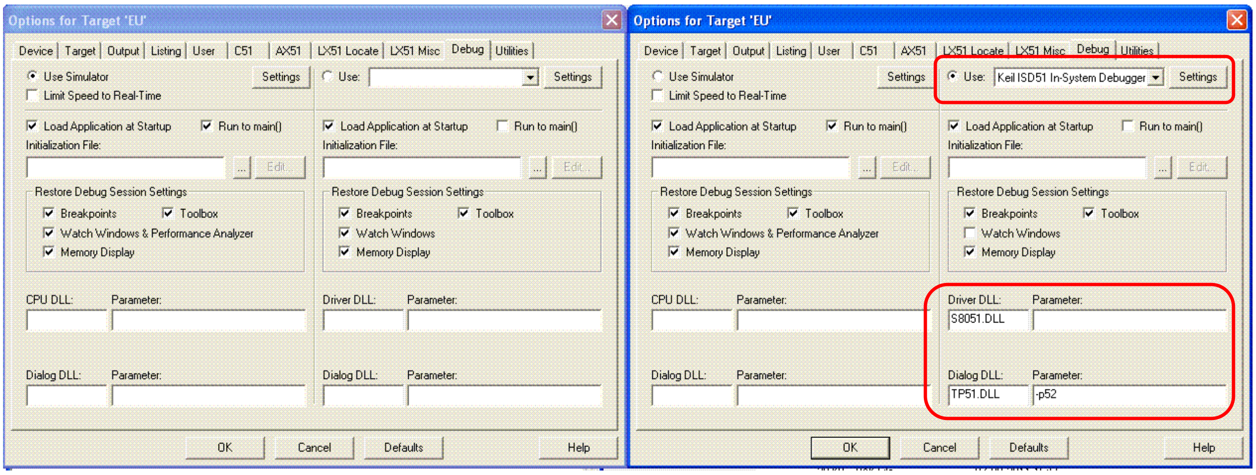
□ Enable ANSI integer promotion rules

Include Paths: ..\..\Z-WAVE\INCLUDE;..\..\Z-WAVE\IO_DEFINES;.\BUILD_PRJ;.\BUILD_PRJ\LEDDIMMER_Z

Misc Controls: DB OE NOCO SB LC CD

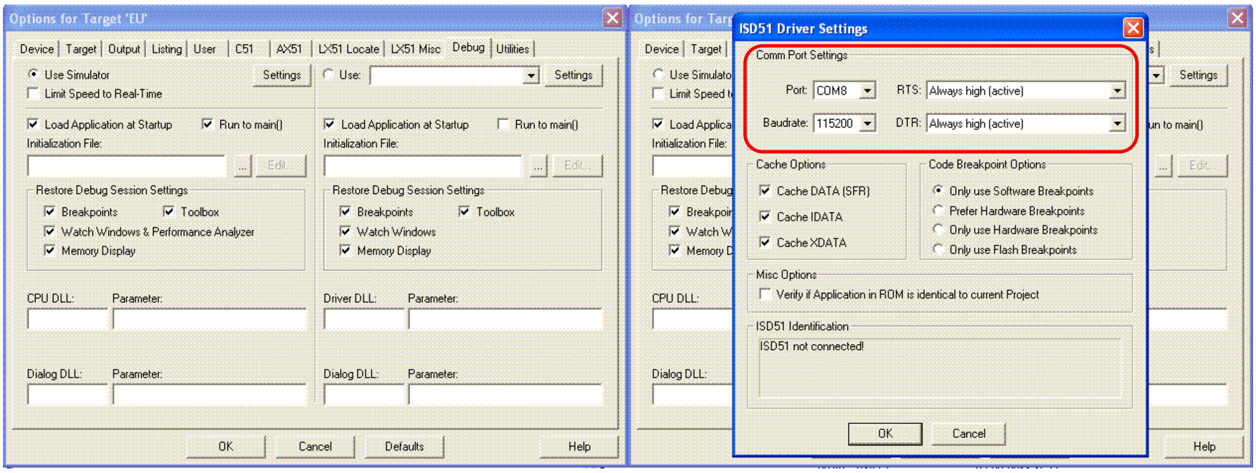
Compiler control string: LARGE OBJECTADVANCED WARNINGLEVEL (1) OPTIMIZE (11,SIZE) BROWSE NOINTPROMOTE

Options for Target 'x' -> Debug:


Options for Target 'EU'
Device | Target | Output | Listing | User | C51 | AX51 | LX51 Locate | LX51 Misc | Debug | Utilities
Settings
Restore Debug Session Settings
Edit...
OK
Cancel
Defaults
Help

Original: ● Use Simulator □ Limit Speed to Real-Time; ○ Use: ; ☑ Load Application at Startup ☑ Run to main() | ☑ Load Application at Startup □ Run to main(); Initialization File: ; ☑ Breakpoints ☑ Toolbox ☑ Watch Windows & Performance Analyzer ☑ Memory Display | ☑ Breakpoints ☑ Toolbox ☑ Watch Windows ☑ Memory Display; CPU DLL: Parameter: ; Dialog DLL: Parameter: ; Driver DLL: Parameter: ; Dialog DLL: Parameter:

New: ○ Use Simulator □ Limit Speed to Real-Time; ● Use: Keil ISD51 In-System Debugger; ☑ Load Application at Startup ☑ Run to main() | ☑ Load Application at Startup □ Run to main(); Initialization File: ; ☑ Breakpoints ☑ Toolbox ☑ Watch Windows & Performance Analyzer ☑ Memory Display | ☑ Breakpoints ☑ Toolbox □ Watch Windows ☑ Memory Display; CPU DLL: Parameter: ; Dialog DLL: Parameter: ; Driver DLL: S8051.DLL Parameter: ; Dialog DLL: TP51.DLL Parameter: -p52



Port: COM8 RTS: Always high (active)

Baudrate: 115200 DTR: Always high (active)

☑ Cache DATA (SFR)

☑ Cache IDATA

☑ Cache XDATA

● Only use Software Breakpoints

○ Prefer Hardware Breakpoints

○ Only use Hardware Breakpoints

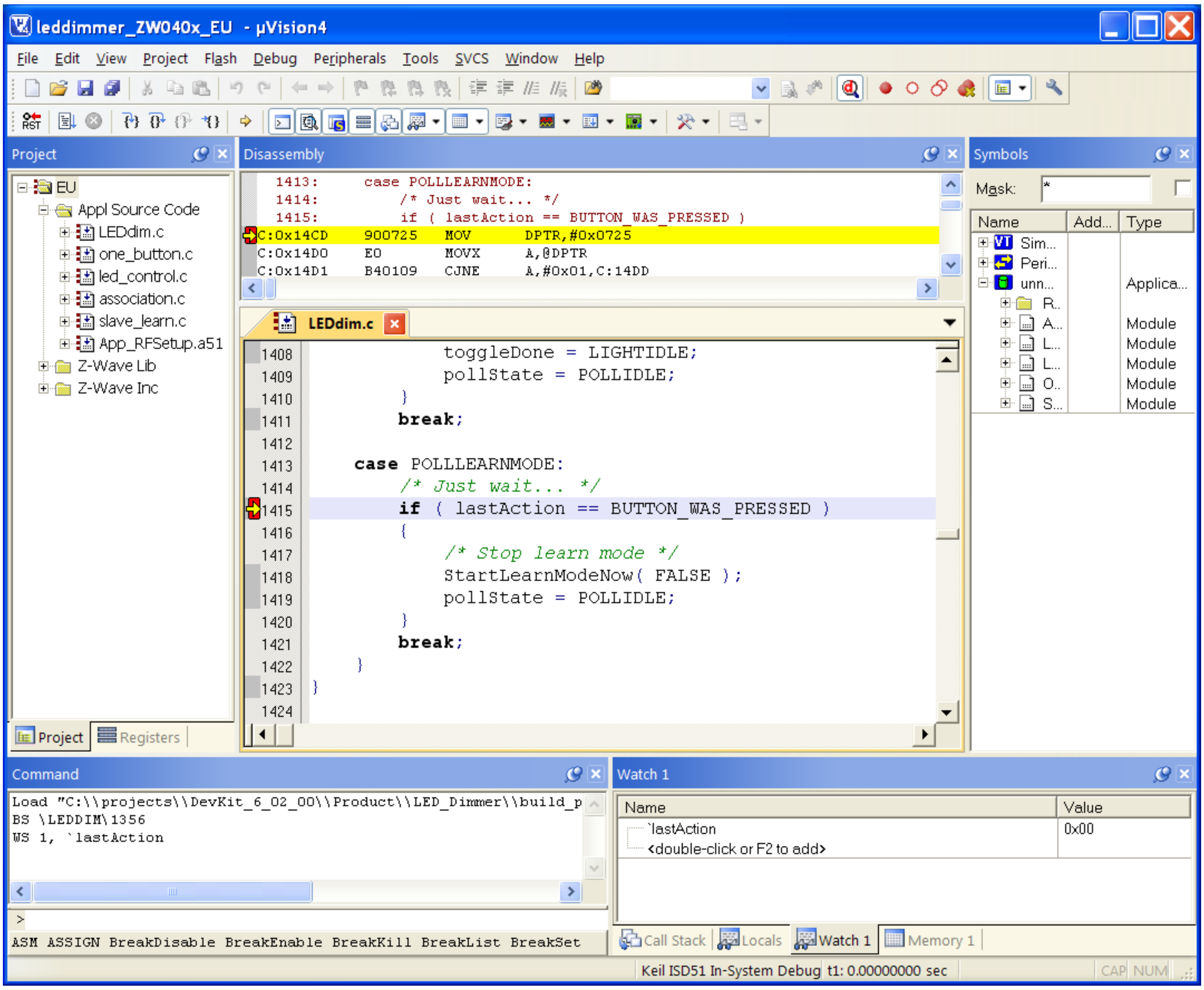
○ Only use Flash Breakpoints

□ Verify if Application in ROM is identical to current Project

ISD51 not connected!

Notice: uVision configured to use software breakpoints to obtain a simple interface to 400 Series chip.

The screen dump below shows the debugger in action. The IDE has hit the configured breakpoint and it is possible to inspect the parameters. For details about the debugger facilities refer to the Keil uVision4 IDE documentation



```
   1413:      case POLLLEARNMODE:
   1414:          /* Just wait... */
   1415:          if ( lastAction == BUTTON_WAS_PRESSED )
C:0x14CD      900725    MOV       DPTR,#0x0725
C:0x14D0      E0        MOVX      A,@DPTR
C:0x14D1      B40109    CJNE      A,#0x01,C:14DD
```

```
              toggleDone = LIGHTIDLE;
              pollState = POLLIDLE;
          }
          break;

      case POLLLEARNMODE:
          /* Just wait... */
          if ( lastAction == BUTTON_WAS_PRESSED )
          {
              /* Stop learn mode */
              StartLearnModeNow( FALSE );
              pollState = POLLIDLE;
          }
          break;
      }
}
```

```
Load "C:\\projects\\DevKit_6_02_00\\Product\\LED_Dimmer\\build_p
BS \LEDDIM\1356
WS 1, `lastAction
>
ASM ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet
```

Name	Value
`lastAction	0x00
<double-click or F2 to add>	

NOTICE: It is recommended to only use one breakpoint at a time to enable it as a hardware breakpoint. Configuring multiple breakpoints result all interpreted as software breakpoints slowing execution down considerably.

REFERENCES

[1] Sigma Designs, INS12035, Instruction, Z-Wave 400 Series Developer’s Kit v6.01.03 Contents.

[2] Sigma Designs, APL10979, Instruction, Porting Z-Wave Appl. SW from ZW0301 to 400 Series.

INDEX