NXP Semiconductors
Application Note

Document Number: AN5282
Rev. 3, 09/2016

CodeWarrior to S32 Design Studio (S32DS) Migration Guide

Migrating legacy programs to the new S32DS IDE environment

by: David Chung

Contents

1 Introduction........................... 1

2 How to Migrate an ARM project from CodeWarrior to S32DS.......................... 3

3 CodeWarrior Eclipse and S32DS Comparison for Power Architecture........................6

4 Project Migration from CodeWarrior Eclipse for Power Architecture................ 26

5 Project Migration from CodeWarrior Classic for Power Architecture.............................51

6 Assembly Translation..53

7 Debug Configurations.................. 55

8 Building the Converted Project......... 58

9 Conclusion..58

10 Revision History...................... 58

1 Introduction

This application note aims to teach the reader how to migrate projects from CodeWarrior v10.6 (CW Eclipse or CWE) and CodeWarrior for MPC55xx, MPC56xx 2.10 (Classic CW or CCW) embedded software development tools to S32 Design Studio (S32DS). There are code examples included in this application note which support KEAZ128 for the ARM® portion and MPC5604B and MPC5644C for the Power Architecture® section.

1.1 What is S32DS?

S32DS is NXP's new development tool for Ultra-Reliable Automotive and Industrial MCUs. It is a no-cost, Eclipse-based integrated development environment (IDE) that integrates various open-source software components, such as the GNU Compiler Collection (GCC) and GNU Debugger (GDB), as well as NXP proprietary device initialization tools. The latter provides a layer of abstraction so that a user needs only call device registers by name, rather than having to specify their memory locations as well. NXP's goal with this tool is to provide customers with an easy-to-use IDE that gives the look and feel of CWE at no cost and provides customers with faster development cycle time than previously available.

1.2 Why migrate?

There are no plans to discontinue CodeWarrior in the near-future, but software enablement for future automotive and ultra-reliable industrial releases from NXP will be supported through S32DS. CodeWarrior support for existing products will remain available. Products released in between tool transition, including the Kinetis EA (KEA), MPC57xx series, and MPC56xx, will be supported by both CodeWarrior and S32DS. However, migrating code to S32DS where available will allow for maximum software reusability for future NXP products. The table below summarizes tool support for selected product lines.

Table 1. Product Support from S32DS, CWE, and CCW

Device	S32DS[1]	CWE	CCW
KEAZ128	Yes	Yes	No
KEAZN64	Yes	Yes	No
KEAZN8	Yes	Yes	No
S32K	Yes	No	No
MPC560xB/C/D	Yes	Yes	Yes
MPC560xP	Yes	Yes	Yes
MPC564xB	Yes	Yes	Yes
MPC564xC	Yes	Yes	Yes
MPC560xS	Yes	Yes	Yes
MPC574xG	Yes	No	No
MPC574xB/C/D	Yes	No	No
MPC5777C	Yes	No	No
MPC5777M	Yes	No	No
MPC574xP	Yes	No	No
MPC574xR	Yes	No	No
MPC574xK	Yes	No	No
S12G128	No	Yes	No
S12XE	No	Yes	No
S12ZVM	No	Yes	No
S12ZVC	No	Yes	No
S12VL	No	Yes	No
S12ZVR	No	Yes	No

1. Future S32DS releases will support more products

S32DS features a CodeWarrior project importer for KEA MCUs that allows a user to migrate a CodeWarrior KEA MCU project to S32DS with a click of a button, because CodeWarrior projects for KEA MCUs and S32DS both use the GCC Compiler. Power Architecture Processor projects are not so lucky: CodeWarrior project for Power Architecture products use a propriety NXP compiler. CWE and S32DS may share the same Eclipse tools framework, but differences such as compiler and project structure schemes prohibit a straight one-to-one migration of CodeWarrior Power Architecture projects to S32DS. An owner of a CodeWarrior Power Architecture project that wishes to migrate to S32DS will have to start a new project. This application note seeks to help the user streamline the migration process, covering the items that must be migrated over, what needs to stay, and what has to be modified in order for the new S32DS project to work the same as its CodeWarrior counterpart did.

2 How to Migrate an ARM project from CodeWarrior to S32DS

There are separate versions of S32DS for ARM and Power Architecture. The ARM tool is called S32 Design Studio for ARM.

KEA is the only ARM device that is supported by both CodeWarrior and S32DS. Importing a KEA project from CWE to S32DS requires the CodeWarrior project importer. Go to *File>Import* as shown in the following figure.

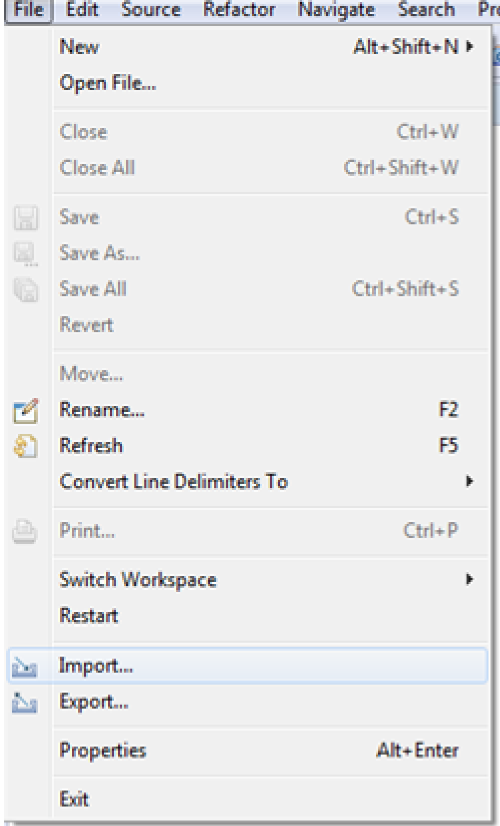

Figure 1. S32DS import option

The CodeWarrior project importer option is located within the S32DS Design Studio folder. Expand it, as shown in the figure below.

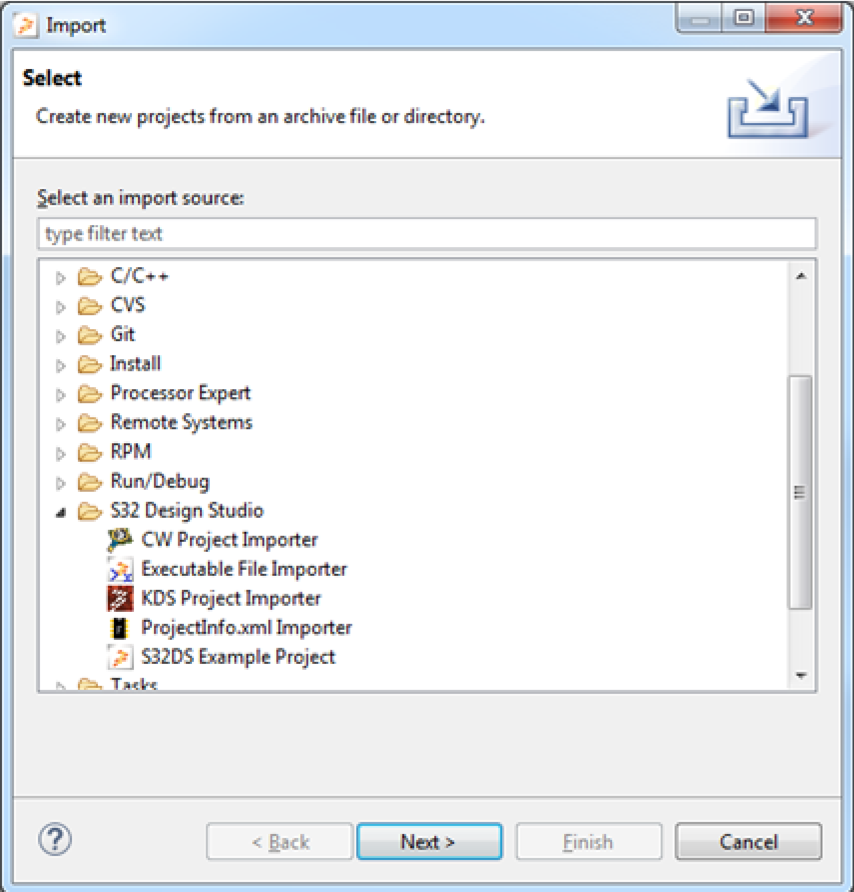

Figure 2. CW Project Importer option

Select CW Project Importer and click Next. Configure the importer options to your liking and browse for your CWE project. You can also give your imported project a new name. If you choose not to, the wizard will give your imported project the same name as the CWE project. In this application note's example, Old Project_KEA is selected as shown in the figure below.

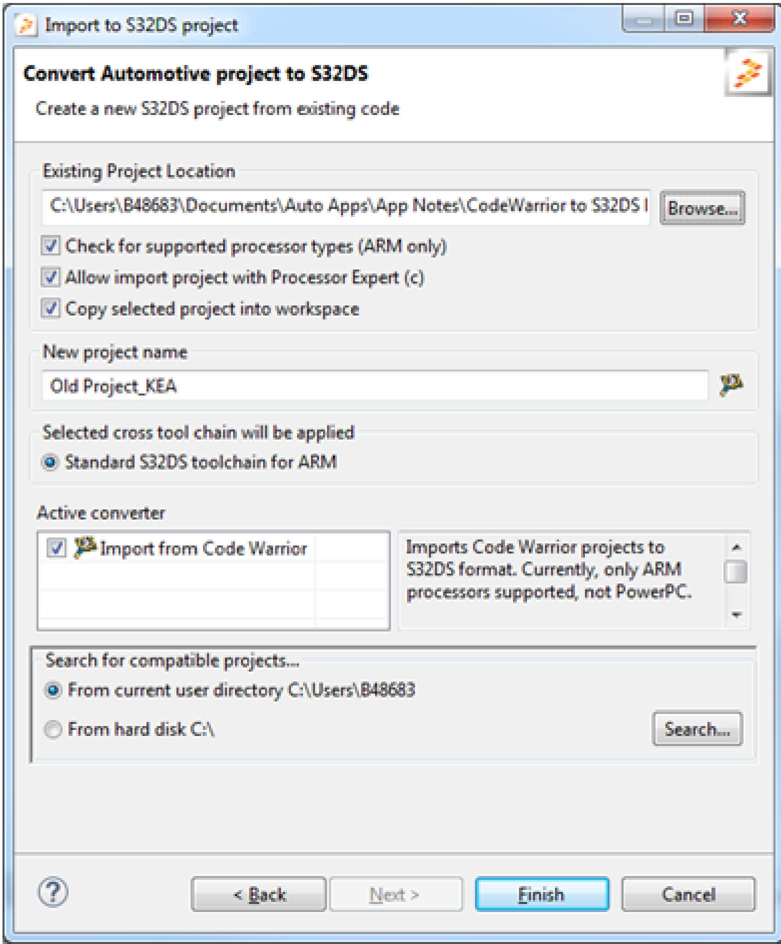

Figure 3. CW Project Importer

Then click on Finish. An S32DS project will appear in your S32DS Project Explorer, as shown in the figure below.

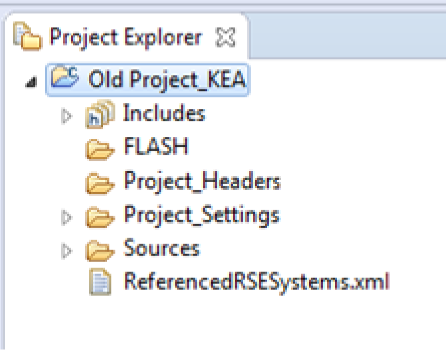

Figure 4. KEA project migrated to S32DS

The CodeWarrior project importer does not modify the original CWE project. It creates a copy of the CWE project, configures it for S32DS, and stores it in your S32DS workspace directory. Since CWE ARM projects use the GCC compiler just like S32DS, migration is straightforward enough, compared to Power Architecture, for the process to be automated. The project importer adjusts the sysroot, which specifies the path to compiler libraries, to link to the S32DS libraries. The importer then modifies the toolchain selection in the debug configurations of the CWE project. If you simply import a CWE project using the Existing Projects into Workspace option, these steps would not be performed and the project would be unable to compile in S32DS.

An imported CWE project will retain the folder structure of the original CodeWarrior project. Therefore, a blank CWE project imported into S32DS will still look different from a blank project generated by S32DS. Regardless, a CWE project imported using the CodeWarrior project importer will work fine on S32DS.

3 CodeWarrior Eclipse and S32DS Comparison for Power Architecture

The version of S32DS for Power Architecture is called S32 Design Studio for Power Architecture. Since Power Architecture products use a legacy NXP compiler in CWE while S32DS uses GCC for all devices, migration of Power Architecture products is much more complex than for ARM.

CodeWarrior Eclipse and CodeWarrior Classic, despite sharing the same branding, are ultimately very different IDEs. This application note will start with CodeWarrior Eclipse, as it shares a framework with S32DS.

CWE and S32DS are both based on the Eclipse open-source IDE. The respective user interfaces of the development tools look virtually identical as shown in the figure below and Figure 6.

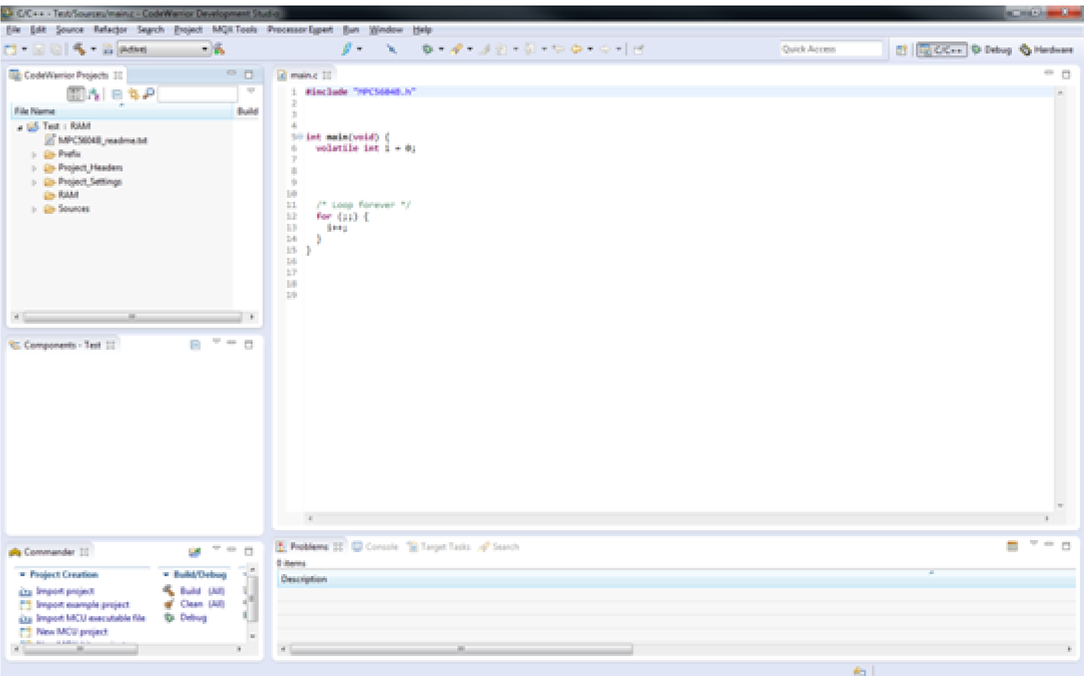

Figure 5. CWE UI

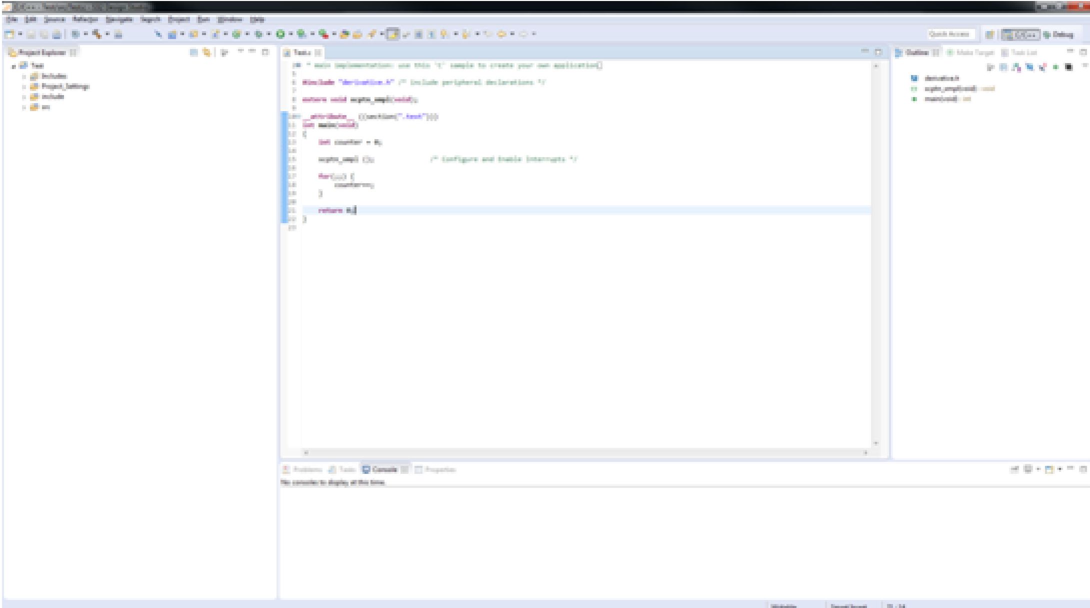

Figure 6. S32DS UI

3.1 Workspace Structure of CWE and S32DS

The NXP Power Architecture portfolio comprises many products that differ from each other in many respects, including core types and core counts. Projects within CWE will differ based on whether the Power Architecture project is for a single core device or a multi-core one; the same is true of S32DS. For the sake of organization, this application note will explain single and multi-core workspace structures in separate sections. The current section will discuss single core projects.

3.2 Workspace Structure for Single-Core Projects

Since both IDEs are Eclipse-based, project structures look similar, but are ultimately different. Figure 7 shows new projects as they appear in CWE and S32DS, respectively. In both screenshots, the project has been compiled once in debug/FLASH configuration so that each already has an automatically generated folder containing object files and binaries.

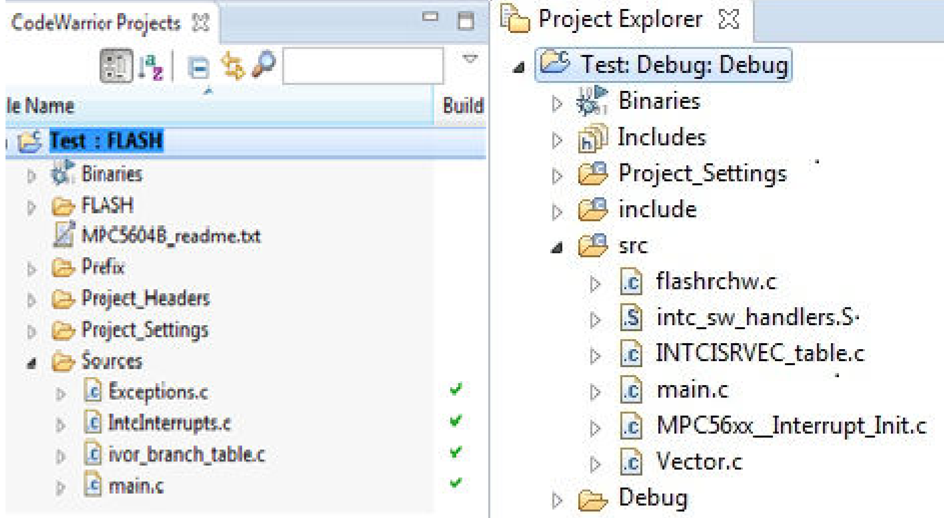

Figure 7. a) New Project on CW b) New project on S32DS

CWE and S32DS project folders use different names, but, with the exception of Includes (with the capital ‘I’), MPC56xxx_readme.txt, and Prefix, every folder in one IDE has a functional equivalent in the other. This section will cover each folder one-by-one. The table below summarizes the folder equivalents between the two tools and the following sections explain each folder in more detail.

Table 2. S32DS and CodeWarrior folder equivalents

CodeWarrior	S32DS
Binaries	Binaries
FLASH	Debug
Project_Headers	include
Project_Settings	Project_Settings
Sources	src
[No Equivalent]	Includes
Prefix	[No Equivalent]
MPC56xxx_readme.txt	[No Equivalent]

3.2.1 FLASH/Debug

To start with, FLASH in CWE is the same as Debug in S32DS. These two folders do not exist when a new project is first created. They are automatically generated upon compilation and contain the object files and binaries created during compilation, such as the project binary .elf file, which stands for Executable and Linkable Format. The name of this folder is not always Debug/FLASH; it inherits the name of the active build configuration at the time of compilation. Build configurations will be further explained in a subsequent section.

3.2.2 Binaries

The two Binaries folders are identical and are also created by the tool upon compilation. These are virtual folders in the sense that no Binaries folder actually exists in the file system. CWE and S32DS generate the folder in the workspace to show the user the .elf file to which a project corresponds.

3.2.3 Project_Headers/include

Project_Headers in CodeWarrior is equivalent to the *include* (all lower case) folder in S32DS. This folder contains the for header files. The files that populate this folder by default vary depending on the embedded target of the associated project, but for single-core Power Architecture projects, the folder contains *Exceptions.h, IntcInterrupts.h, MPC56xxx_HWInit.h, typedefs.h*, and the device header. S32DS's include folder contains *derivative.h, typedefs.h*, and the device header. Figure 8 below shows the two folders side-by-side.

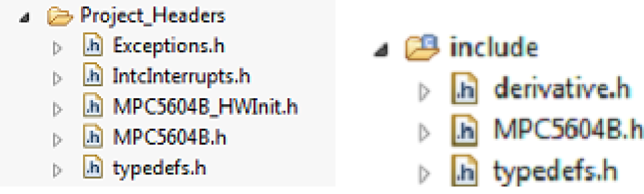

Figure 8. a) Project_Headers b) include

The files *Exceptions.h, IntcInterrupts.h*, and MPC56xxx_*HWInit.h* contain function prototypes related to interrupt handler initialization and have source file counterparts in the Sources folder, explained later. The device header file, which specifies a target's embedded register addresses, inherits the name of the target device. So an MPC5604B project would have a device header called *MPC5604B.h*. Lastly, *typedefs.h* contain certain type definitions that are commonly used in Power Architecture projects, such as *vuint8_t*, which is an unsigned, 8-bit integer.

S32DS initializes the flash and interrupts differently and so does not contain such files as *Exceptions.h*. The device header is identical to the CWE version and *typedefs.h* nearly so. S32DS's version of *typedefs.h* contains an extra section of definition for when the macro __STDC__ is defined. No types are lost to this change so it makes no real difference to a project. *derivative.h*, whose purpose is to save users time, is the only new file. Users might have multiple headers that every source file of a project will need to reference, for example the device header, to which *derivative.h* generates with a reference using the "#include" preprocessor command. So instead of having to include every header file in every source file, the user can do it once in *derivative.h* and then reference only *derivative.h* in each source file. The figure below juxtaposes a newly generated *derivative.h* and one being used as is intended.

```
2⊕ * Note: This file is recr
5
6  /* Include the derivative-
7  #include <MPC5604B.h>
8
```

```
2⊕ * Note: This file is recreated
5
6  /* Include the derivative-specif
7  #include <MPC5604B.h>
8  #include <stdio.h>
9  #include <string.h>
10 #include <math.h>
11 #include "RTC.h"
12 #include "SPI.h"
13
```

Figure 9. a) New derivative.h b) Typical derivative.h

There is also the *compiler_api.h* file that only exists in the S32DS. This file contains the definitions for calling assembly functions and is intended for the exception handling and accessing the e200 special-purpose registers. The user does not have to worry about this file for the vast majority of the time. Table 3 sums up the purpose of the header files.

Table 3. Project_Headers/include Files, Equivalents, and Function

CWE	S32DS	Function
[No Equivalent]	derivative.h	Save users time. Include multiple source files in *derivative.h* once, then just include *derivative.h* in each source file instead of having to include multiple headers everywhere.
MPCxxxx.h	MPCxxxx.h	Device header. Contains peripheral register definitions and memory locations
typedefs.h	typedefs.h	Declarations of various types for use in code, such as different-byte-sized integers, signed/unsigned, etc.
Exceptions.h	[No Equivalent]	Function prototypes for *Exceptions.c*
IntcInterrupts.h	[No Equivalent]	Function prototypes for *IntcInterrupts.c*
MPCxxxx_HWInit.h	[No Equivalent]	Function prototypes for *MPCxxxx_HWInit.c*
[No Equivalent]	compiler_api.h	Assembly language macros for exception handling

3.2.3.1 Project_Settings

The *Project_Settings* folders on both IDEs are functionally identical. The folder is target-specific and contains linker, debug, and device startup information. Organizationally, both versions' subfolders are *Debugger*, *Linker_Files*, and *Startup_Code*, but the contents within the subfolders are different, as in Figure 10.

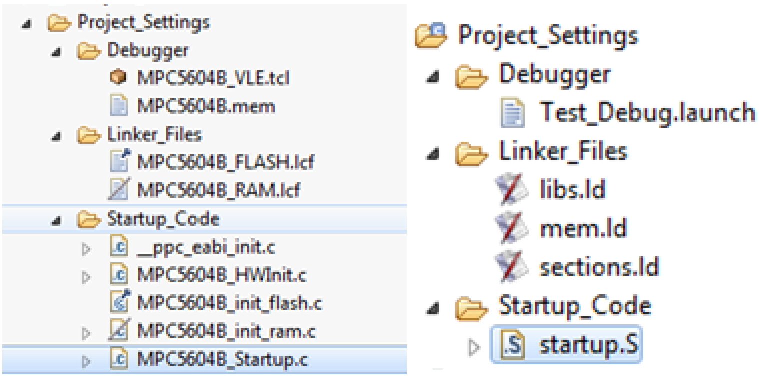

Figure 10. Project_Settings in a) CWE and b) S32DS

Debugger in CWE contains two MCU initialization scripts for Power Architecture devices and a memory configuration file; S32DS instead has a single launch script. The initialization scripts and memory config, instead of being copied into the project, are hardcoded in S32DS, which the launch script calls.

Linker_Files are organized differently as well. CWE has an *MPCxxxx_FLASH.lcf* and an *MPCxxxx_RAM.lcf*. Only one is used at any time, depending on the build configuration selected. S32DS holds a *libs.ld*, *mem.ld*, and *sections.ld*.

Startup_Code in CWE contains five files to S32DS's one file. CWE includes *__ppc_eabi_init.c*, *MPCxxxx_HWInit.c*, *MPCxxxx_init_flash.c*, *MPCxxxx_init_ram.c*, and *MPCxxxx_Startup.c*, whereas S32DS simply contains *startup.S*. Despite all the differences, the two IDEs just perform the same task differently. S32DS just initializes everything from a single assembly file. CWE's entry point is in *MPCxxxx_Startup.c*. The *__startup()* function branches to *__start()*, which is a CodeWarrior library function. All the other files within *Startup_Code* are callbacks for *__start()*. Because *__start()* is a library function, it is a generic implementation. It calls the functions in the source files for anything device-specific. For example, *__ppc_eabi_init.c* determines the build configuration (e.g. flash or RAM) and calls the appropriate hardware initialization

functions. These include the functions in *MPCxxxx_init_flash/ram.c*, which in turn call *MPCxxxx_HWInit.c*. *MPCxxxx_HWInit.c* disables the watchdog, initializes the SRAM, and initializes the external bus. *MPCxxxx_init_flash.c* also has definitions for the reset vector and boot information by the name *RCHW*. Such information can be found in S32DS in the file *flashrchw.c*. Table 4 summarizes these files.

Table 4. Project_Settings Files, Equivalents, and Functions

CWE	S32DS	Function
MPCxxxx_VLE.tcl and MPCxxxx.mem	[Project Name]_Debug/Release.launch	Initializes the debugger.
MPCxxxx_FLASH.lcf and MPCxxxx_RAM.lcf	libs.ld, mem.ld, and sections.ld	Linker files. CWE separates by build configuration and S32DS by component
__ppc_eabi_init.c, MPCxxxx_HWInit.c, MPCxxxx_init_flash.c, MPCxxxx_init_ram.c, MPCxxxx_Startup.c	startup.S	S32DS uses a single assembly file to initializes the MCU. CWE calls a generic library function. The multiple files in the list are device-specific callbacks

3.2.4 Sources/src

Source and *src* are the respective source file directories for CodeWarrior and S32DS (Figure 11).

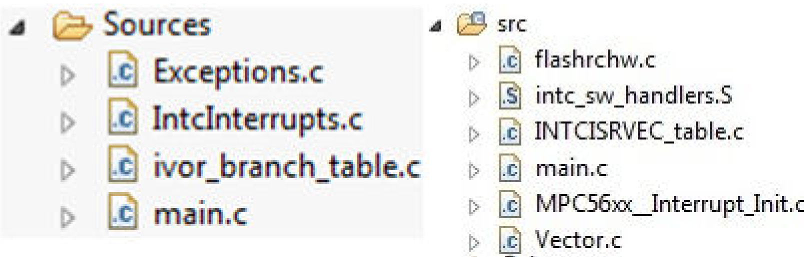

Figure 11. a) Sources b) src

The new build of S32DS changes the naming convention for new projects.

IntcInterrupts.c in CWE does everything that S32DS's *MPC56xx_Interrupt_Init.c, intc_sw_handlers.S*, and *INTCISRVEC_table.c* collectively do. *MPC56xx_Interrupt_Init.c* contains functions to initialize the interrupt controller peripheral (INTC), which sets the interrupt vector table base address and software/hardware interrupt mode. *IntcInterrupts.c* has the same function, just by a different name.

In S32DS, *intc_sw_handlers.S* is an assembly file that contains instructions for saving the state of the program at the time of an interrupt and for restoring the state once the CPU returns from interrupt. For example, general purpose registers and the return address must be pushed onto the stack before entering the interrupt service routine (ISR) and then retrieved once the ISR is done. Only nested interrupt implementation is supported. In CWE, *IntcInterrupts.c* contains the same assembly function for context preservation, but it is only used in nested interrupt mode. If CWE is left in the default configuration, which does not allow for nested interrupts, a CWE library function is used instead, and the assembly function is not compiled.

INTCISRVECT_table.c contains an array of function pointers called *IntcIsrVectorTable*. S32DS ISRs must be implemented in flash. You must reference your ISR in that table at the correct vector position of your interrupt. Therefore if you use an RTC interrupt on the MPC5604B, which is vector 38 (see MPC5604B reference manual), and you write an *RTC_ISR* (you

can call it whatever you want), then “&RTC_ISR”, must be written at *IntcIsrVectorTable[38]*. CWE ISRs, by contrast, are implemented in RAM. *IntcInterrupts.c* contains a function called *INTC_InstallINTCInterruptHandler*, which you must call during initialization and pass it your ISR’s address and priority level.

Vector.c and *ivor_branch_table.c* serve the same purpose. There are multiple IVOR vectors. Each IVOR responds to a different type of exception, such as when an alignment interrupt occurs or the core executes an instruction that does not exist. In the case of peripheral interrupts, IVOR4 is called. The interrupt handler then branches to the location specified in the IVOR4 section of the ivor branch table in *ivor_branch_table.c* and *Vector.c* for CWE and S32DS, respectively.

flashrchw.c has to do with program startup from flash. Its CWE counterpart is in *MPCxxxx_init_flash.*c from the *Project_Settings* folder. It generates the rest vector and a special reset word used by the core internal flash boot, which identifies which core should run. Table 5 describes each file’s purpose and equivalents between CWE and S32DS.

Table 5. Sources/src Files, Equivalents, and Functions

CWE	S32DS	Function
Project_Settings> Startup_Code> MPCxxxx_init_flash.c	flashrchw.c	See table 2-3
IntcInterrupts.c	intc_sw_handlers.S, INTCISRVEC_table.c, MPC56xx_Interrupt_Init.c	Initializes interrupt controller and functions for registering ISRs
ivor_branch_table.c	Vector.c	Branch table for core exceptions. When external interrupts from peripherals fire, IVOR4 is called. Interrupts then branch to the function specified in the IVOR4 section. INTC_INTC_InterruptHandler/ IVOR4_Handler
main.c	main.c	The file where *main()* function is located

3.2.5 Includes (S32DS Only)

Finally, S32DS has the *Includes* directory, which has no equivalent in CodeWarrior. Like *Binaries*, it is not a true folder but is a list of include paths for the project. Figure 12 shows the *Includes* directory of a blank project for the KEAZ128. The *Includes* directory contains the various paths to the Embedded Warrior Library (ewl) and the *include* folder. Other compiler libraries offered by S32DS are newlib and newlib_nano.

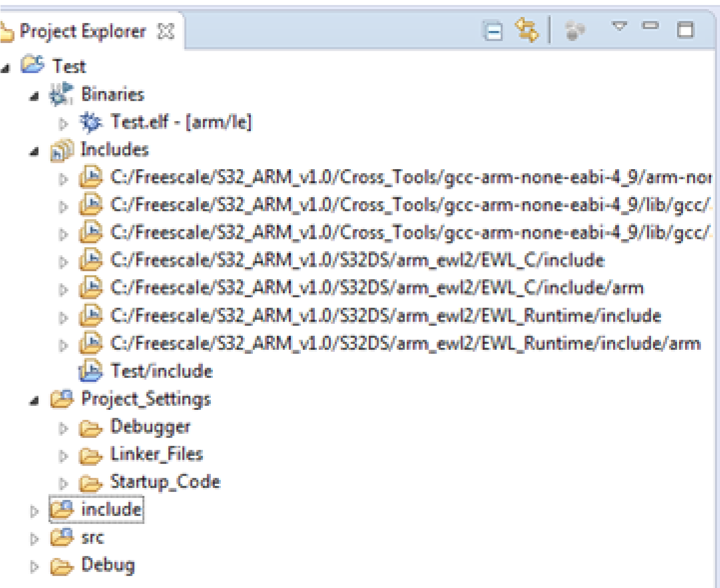

Figure 12. Includes in S32DS

3.3 Prefix (CWE Only)

The *Prefix* folder contains the files *MPC56xxx_FLASH_VLE.prefix* and *MPCMPC56xxx_RAM_VLE.prefix*, where the "prefix" means that the contents are included in every C file in the project. Only one of the prefix files is included in the build at a time, depending on which build configuration is selected. "VLE" stands for "variable-length encoding". Each file contains three definitions which are necessary for the CWE preprocessor in determining which parts of the startup sequence must be compiled. S32DS does not contain prefix files because it uses the GCC Compiler.

3.4 MPC5604B_readme.txt

This purpose of this text file is to give some guidance and tips to the user. There is no code in the file, so its presence in a migrated project or lack thereof would not affect the performance of the migrated project

3.5 Workspace Structure for Multi-Core Projects

CWE and S32DS treat multi-core projects differently. CWE creates one project that contains as many additional folders as there are cores in the target device, named *PRCx_Sources*, where 'x' is the index of the core. S32DS generates a separate project for each core that follows the naming convention *[project name]_[core type]*, such as *Test_Z4*. If the project has multiple cores of the same type, S32DS adds an additional suffix so a project would generate as for example *Test_Z4_1* and *Test_Z4_2*. The screenshot below shows the workspace of a multi-core project on CWE and S32DS. Each individual S32DS project's structure in the multi-core working set is the same as a standalone Power Architecture project. CWE's structure does change, which the following subsections will describe. Any folders that are not mentioned in the following subsections can be assumed to be unchanged from the single-core description. Figure 13 displays how multi-core projects appear in the respective CWE and S32DS project explorer windows.

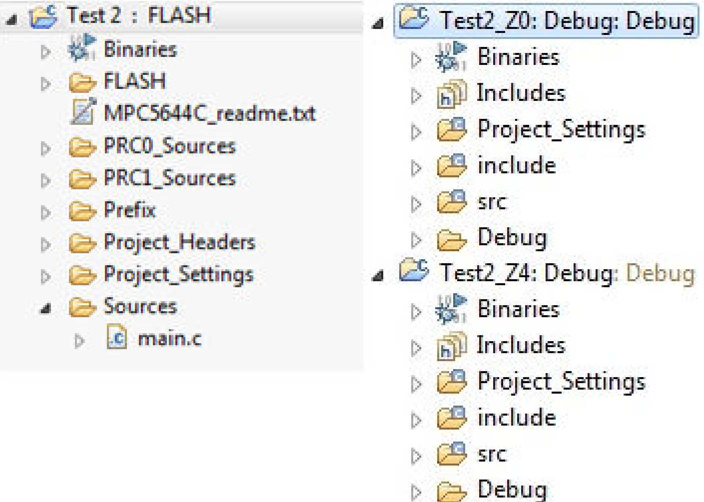

Figure 13. a) Multi-core project in CWE b) Multi-core project in S32DS

The standalone projects for each core link together in S32DS by referencing each other under *Project References* in the project settings, as shown in Figure 14.

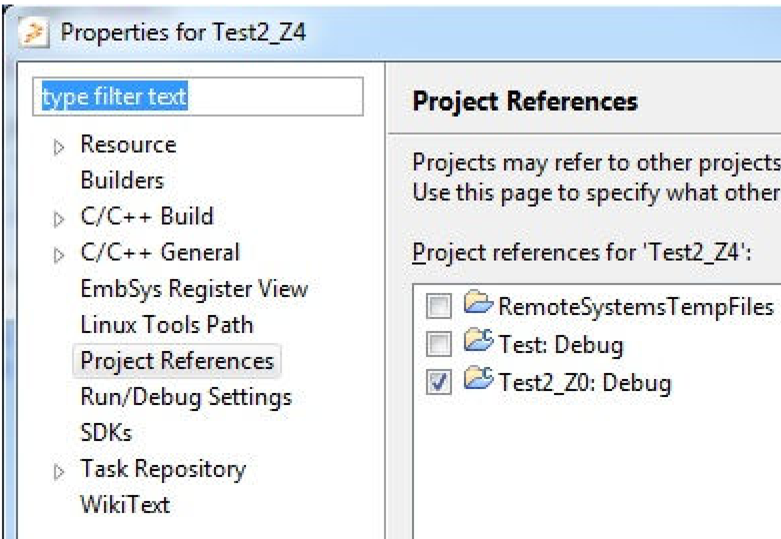

Figure 14. S32DS Multi-core project references

3.6 Project_Headers

The *Project_Headers* folder in a multi-core project contains the header files *Exceptions.h* and *IntcInterrupts.h* for each core that the project uses. The first core's (Core 0) header files still go by the names *Exceptions.h* and *IntcInterrupts.h*. All subsequent core header files follow the naming convention *Exceptions_px.h* and *IntcInterrupts_px.h*, where 'x' goes from 1 to the index of the last core in use. The dual-core MPC5644C's *Project_Headers* folder is shown in Figure 15.

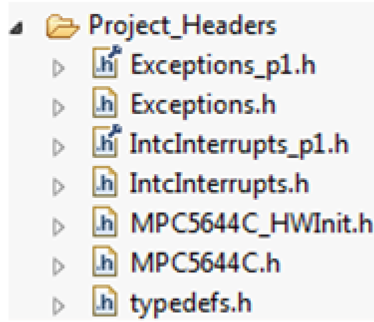

Figure 15. Dual Core header files in CWE

3.7 Project_Settings\Debugger

There is a very minor change between single-core and multi-core projects in CWE. The latter contains separate launch script for FLASH and RAM configurations. This does not affect migration.

3.8 Sources

The CWE *Sources* folder only has *main.c*, which pertains to the first core; the second core's main file, *main_p1.c*, resides within *PRC1_Sources*. There are no changes to the S32DS *src* folders. Each core gets a standalone project so the projects looks like a single-core project.

3.9 PRCx_Sources

For the first core, this folder contains *Exceptions.c*, *IntcInterrupts_p0.c*, and *ivor_branch_table_p0.c*. Every additional *PRCx_Sources* folder has its own version of these three files, by the names *Excpetions_px.c*, *IntcInterrupts_px.c*, *ivor_branch_table_px.c*, as well as its own main, *main_px.c*, and a startup file, *__start_px.c*.

3.10 Project Options

Right-clicking on a project will show various options. The CWE and S32DS project options are largely the same, but the respective lists are arranged differently. For example, CWE has an option called *Show in Windows Explorer*, which opens an explorer at the project's location within the file system; S32DS also has that capability but one must select a project in the project explorer, press ALT+SHIFT+W, and select *System Explorer*. Refer to Figure 16 for screenshots of the two menus.

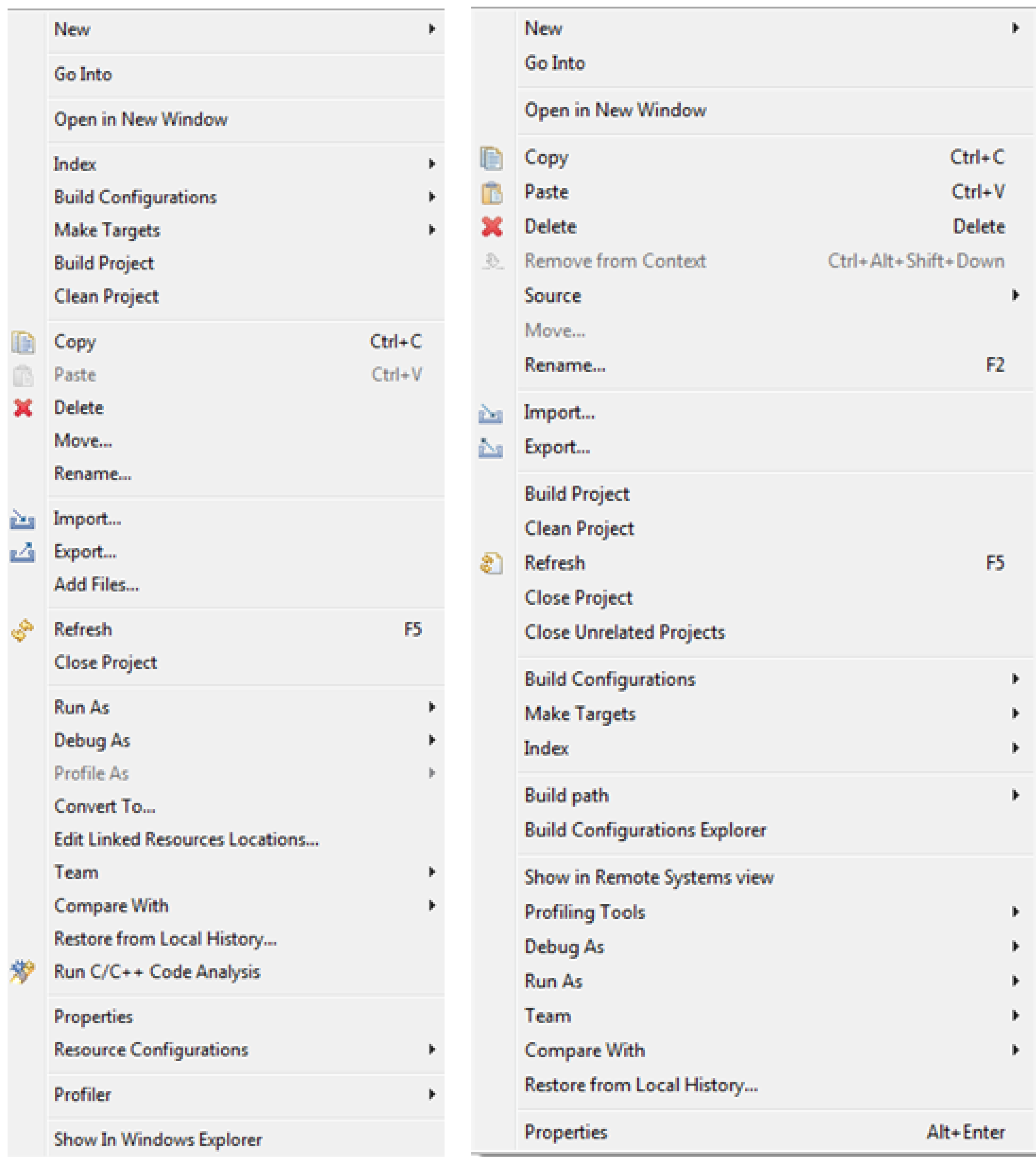

Figure 16. a) Project options on CWE. b) Project options on S32DS.

3.11 Project Properties

Of the options available, arguably the most important to a user is *Properties*. In both tools, selecting *Properties* opens a window, as shown in Figure 17 and Figure 18.

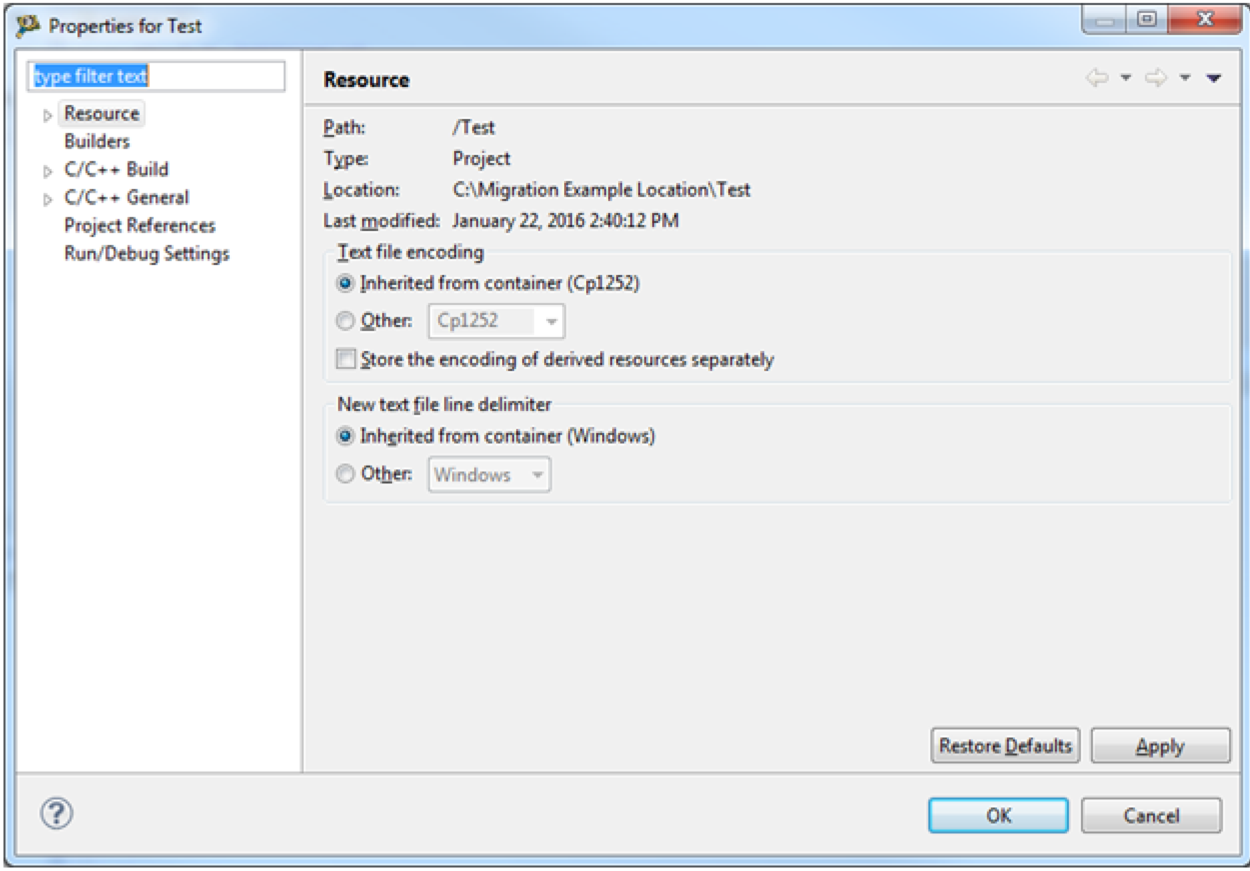

Figure 17. Project Properties on CWE

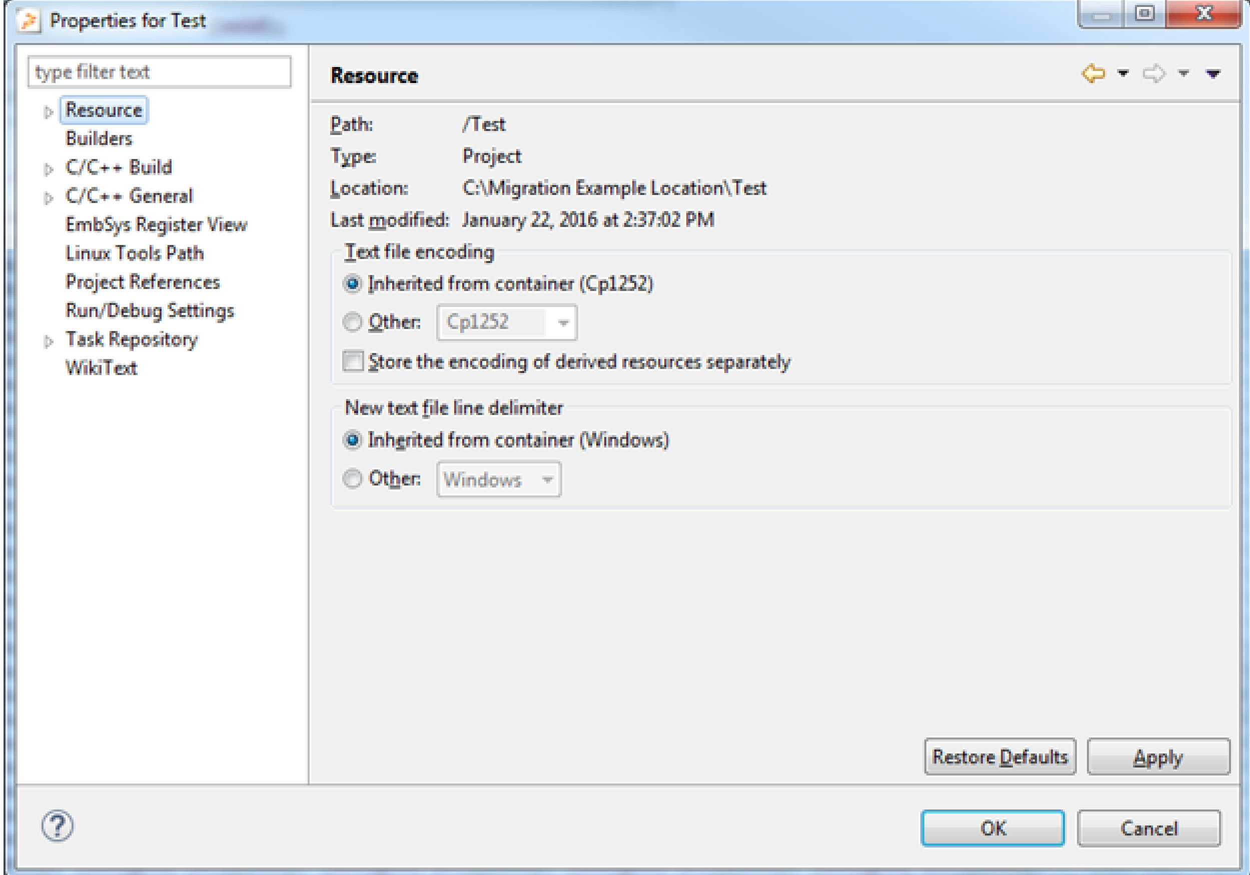

Figure 18. Project Properties on S32DS

S32DS for Power v1.1 and newer supports translation from the BookE (Power Architecture) instruction set to the Variable Length Encoding (VLE) instruction set. The extra steps needed to enable assembly translation are described in Assembly Translation. Library support from CodeWarrior has also been extended: S32DS offers the Embedded Warrior Library (EWL) and Newlib as well as their more compact cousins, ewl_nano and newlib_nano.

Another useful feature within *Properties* is the optimization level. You can select the your project's degree of optimization by going to *Properties>C/C++ Build>Settings>Standard S32DS C Compiler>Optimization* and *Properties>C/C++ Build>Settings>Standard S32DS Assembler>Optimization*. The optimization options available to the user are, in order of increasing optimization, *None*, *Optimize*, *Optimize More*, *Optimize Most*, and *Optimize for size*. Figure 19 shows the optimization window for the C compiler.

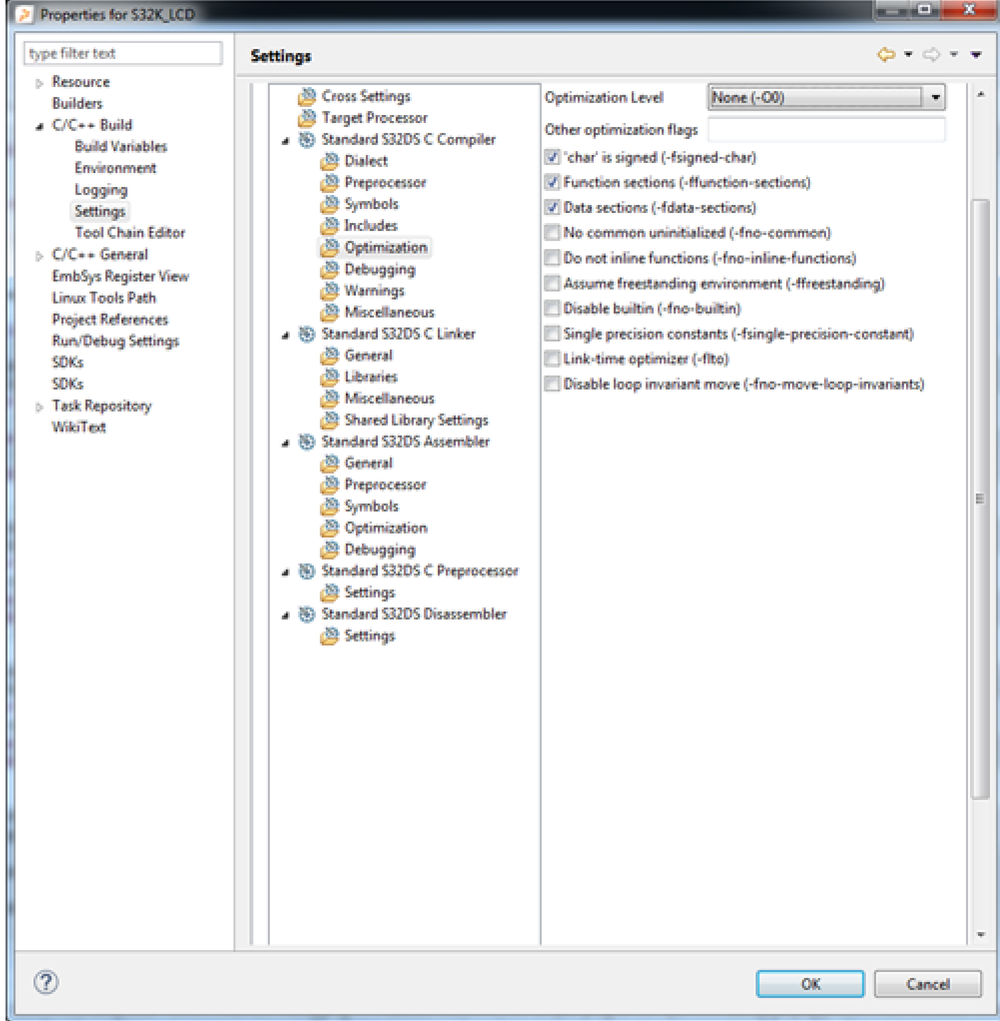

Figure 19. S32DS Optimization Window

Increasing optimization decreases code size, but usually comes at the expense of performance. High levels of optimization also makes a program difficult to debug as some instructions are stripped from the build by the optimizer and others are linked. The best level of optimization ultimately depends on the project.

3.12 Compiler and Debugger

S32DS for Power Architecture uses the GCC v4.92-based Power Architecture Compiler. CWE supports both GCC and a legacy NXP compiler, but Power Architecture CWE projects use the latter. Build a project by clicking on the hammer ( ) icon on the toolbar. Like CodeWarrior, S32DS performs a dynamic build which generates binaries and stores them in a folder called either *Debug*, *Release*, or *Debug_RAM*, depending on which build configuration is selected via the drop down arrow in the above icon.

The debugger consists of three parts. There is the debug interface, which the user interacts with; then there is the core debug engine, which translates the user inputs to debug the code, such as writing breakpoints and program execution; and also a probe interface, which facilitates communications between the computer and the board. S32DS mostly uses a GNU-based debugger, while CWE uses custom legacy-NXP software. Below is a graphical description. Figure 20 shows the debugger layout of CWE and S32DS, where the top label of each block is the component and the bottom one is the developer of the component used by the associated IDE. The important takeaway from this information is that both IDEs use P&E software

for the probe interface. This means that you only need to set your S32DS debug interface configurations to be the same as it was in CWE. So if you used OpenSDA in CWE, keep it OpenSDA in S32DS. The differences between the GNU and legacy NXP debuggers only make a difference behind the scenes.

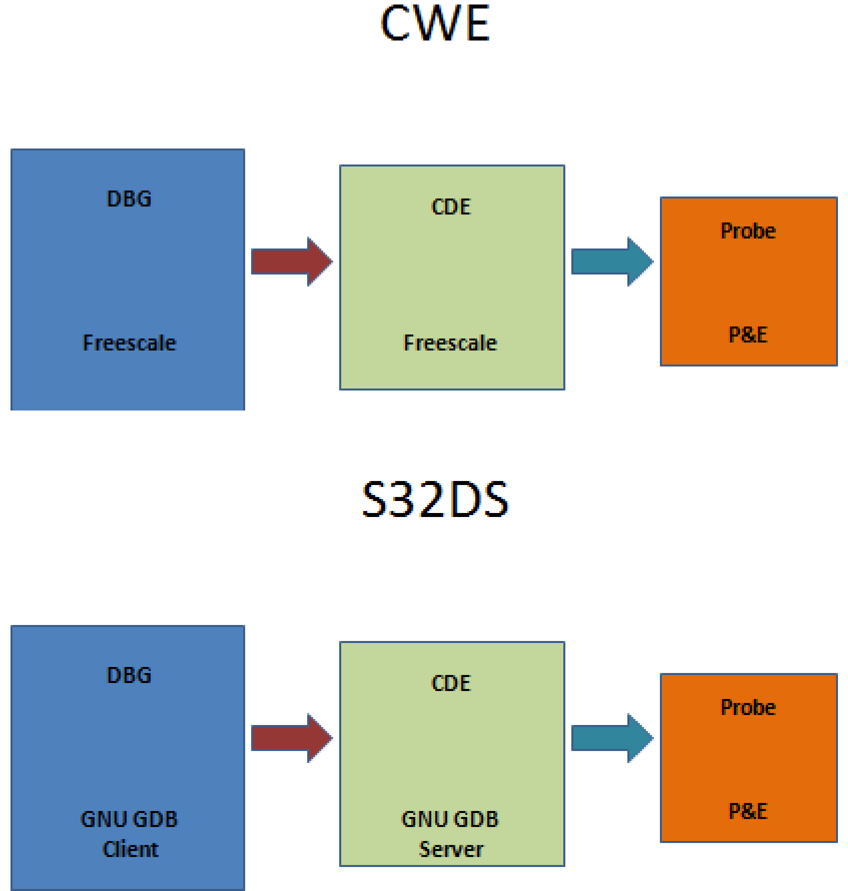

Figure 20. CWE and S32DS Debugger Setup

3.13 New Projects

Project creation in both IDEs follows nearly identical procedures:

1. Select the type of project and give it a name
2. Choose the target MCU (Figure 21)
3. Select connection type. Once the user clicks *Finish* in the project wizard, both IDEs generates the files described in FLASH/Debug.

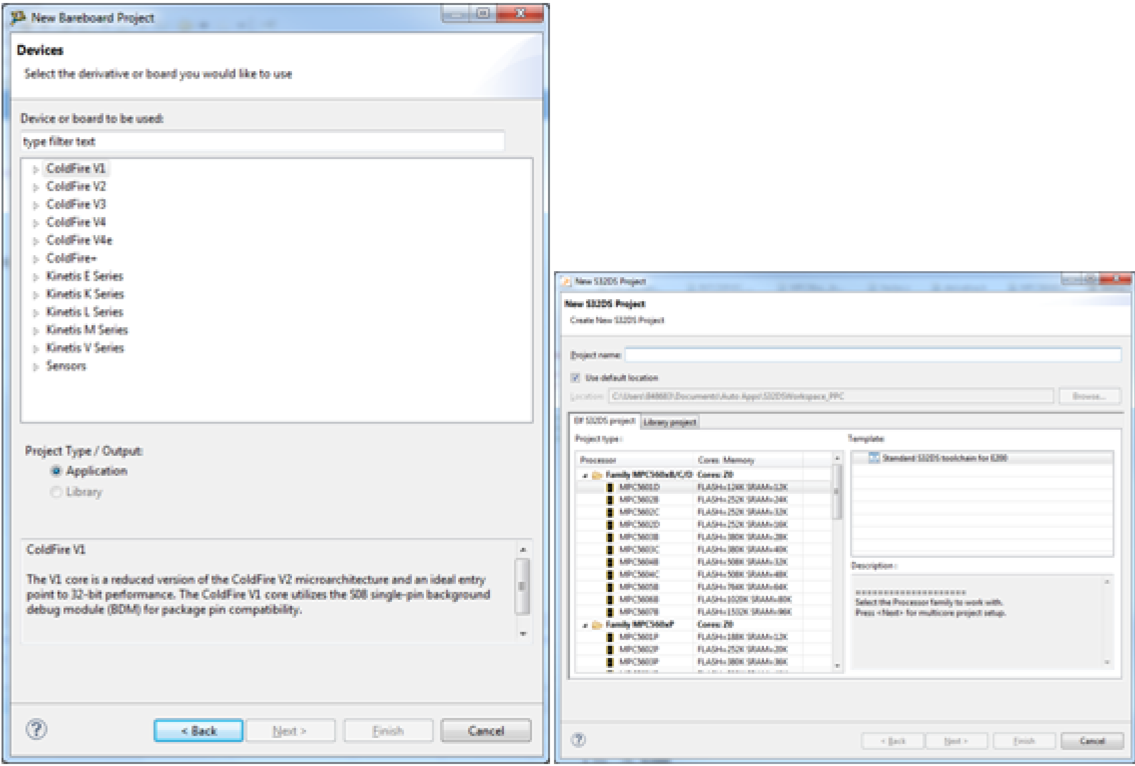

Figure 21. a) CW Target Selection Table. b) S32DS Equivalent.

In both instances, the tool populates the workspace with a project configured for the target. When a new MPC5604B project is created in CWE, for example, main initializes a counter to 0, which increments forever, as seen in Figure 22. S32DS's version also generates a counter as well as a function call to xcptn_xmpl(), which initializes the interrupt controller. CWE initializes the interrupt controller during startup.

main.c

```
1  #include "MPC5604B.h"
2
3
4
5  int main(void) {
6    volatile int i = 0;
7
8
9
10
11   /* Loop forever */
12   for (;;) {
13     i++;
14   }
15 }
```

Test.c

```
2  * main implementation: use this 'C' sample to create your own application
5
6  #include "derivative.h" /* include peripheral declarations */
7
8  extern void xcptn_xmpl(void);
9
10 __attribute__ ((section(".text")))
11 int main(void)
12 {
13     int counter = 0;
14
15     xcptn_xmpl ();              /* Configure and Enable Interrupts */
16
17     for(;;) {
18         counter++;
19     }
20
21     return 0;
22 }
23
```

Figure 22. Blank MPC5604B project in a) CWE and b) S32DS

From here writing a project in S32DS is virtually identical to CWE, except the folders for headers and sources files are called *include* and *src*, respectively, rather than *Project_Headers* and *Sources*.

3.14 Workspace structure of CCW and S32DS

CodeWarrior Classic is the other IDE that takes the CodeWarrior name. Built on a custom NXP IDE framework, CCW looks radically different from CWE. Where CWE had *FLASH* and *RAM* build configurations, CCW has, upon project generation, *internal_FLASH* and *RAM*. These names can be changed in the project settings, discussed later. Figure 23 is a screenshot of a blank CCW project for the MPC5604B.

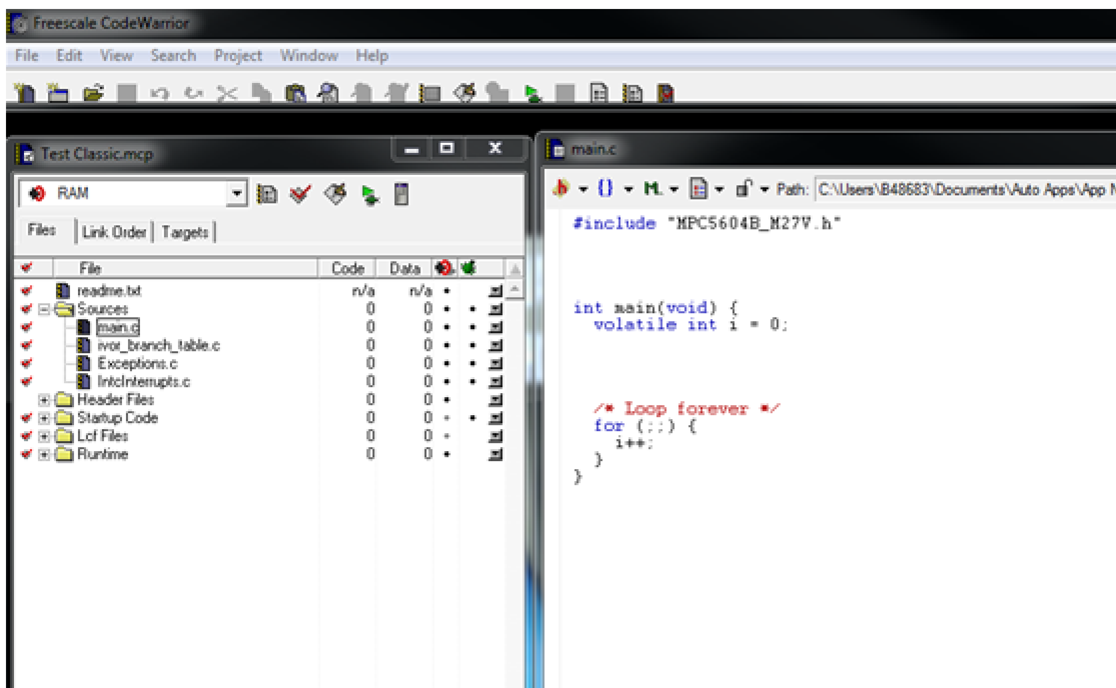

Figure 23. CodeWarrior Classic Blank MPC5604B Project

3.15 Classic CodeWarrior Differences From S32DS

CCW also supports both single-core and multi-core products. Their projects differ in structure accordingly. The following sections will describe the differences.

3.16 Classic CodeWarrior Differences From S32DS, Single-Core

A CCW project contains much the same files as a CWE, but the CCW organizational scheme differs greatly from both CWE and S32DS. Another difference is that the folders that most of the folders that appear in the CCW workspace are virtual folders, similar to the *Include* (with the capital 'I') folder of S32DS. These folders do not actually exist in the file system and and exist in the CCW workspace for organizational purposes. Table 6 summarizes CCW differences from S32DS in the workspace realm.

Table 6. CCW Workspace v S32DS Workspace

CCW	S32DS	Remark
[No Equivalent]	Binaries	Explained earlier.
[No Workspace Equivalent]	Debug	Binaries generated from compilation in CCW are stored in a *bin* folder, which does not appear in the workspace.
Header Files	include	Same function, different name.
[No Equivalent]	Project_Settings>Debugger	Debug launcher in CCW integrated into IDE.
Lcf Files	Project_Settings>Linker_Files	CCW also uses *.lcf* files but the naming convention follows the scheme *MPCxxxx_[derivative].lcf* and *MPCxxxx_[derivative]_DEBUG.lcf*, such as *MPC5604B_M27V.lcf*. The file with the *DEBUG* suffix is for RAM build configuration.
Startup Code	Project_Settings>Startup_Code	The difference between CCW and S32DS is the same as that between CWE and S32DS except that CCW is missing *MPCxxxx_Startup.c*, because entry point is in the library.
Sources	src	Same as CWE.
[No Equivalent]	Includes	Explained Earlier.
Runtime	[No Equivalent]	Contains the libraries *Runtime.PPCEABI.VS.UC.a* and *MSL_C.PPCEABI.bare.VS.UC.a*. In S32DS you would have to link these libraries in the project settings using the libraries' paths in the S32DS installation directory.
readme.txt	[No Equivalent]	Text file with tips and instructions for the user to get started, just like *MPC56xxx_readme.txt* in CWE.

Most of the CCW workspace folders are virtual directories. Figure 24 shows how the files of a CCW project are actually organized in the file system.

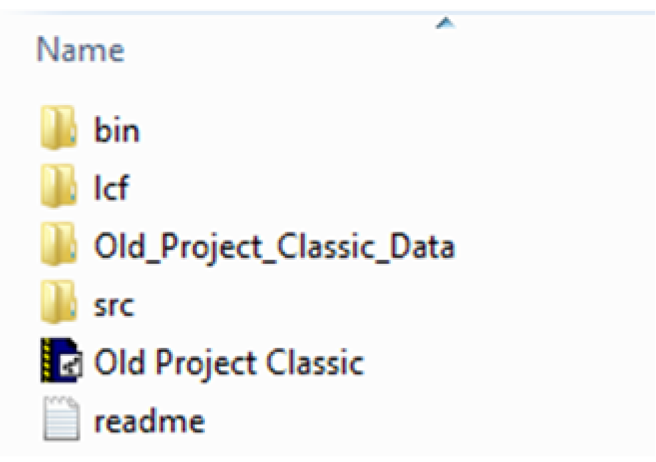

Figure 24. CCW project file directory

Table 7 explains the purpose of each of these file folders.

Table 7. CCW File Directory Summary

Folder	Function
bin	Stores the binary files that are generated during compilation.
lcf	Store the linker files.
*[Project Name]*_Data	Storage directory for temporary files. Automatically generated from compilation. Temporary files include data files regarding IDE preferences and object files.
src	The directory that contains the source **all**source files that were seen in the CCW workspace.

3.17 Classic CodeWarrior differences from S32DS, Multi-Core

A multi-core CCW project generates the same files as in CWE, with the same names. So migrating a multi-core CCW project to S32DS is, practically speaking, the same as going from CWE to S32DS. Figure 25 is a screenshot of a multi-core CCW project workspace.

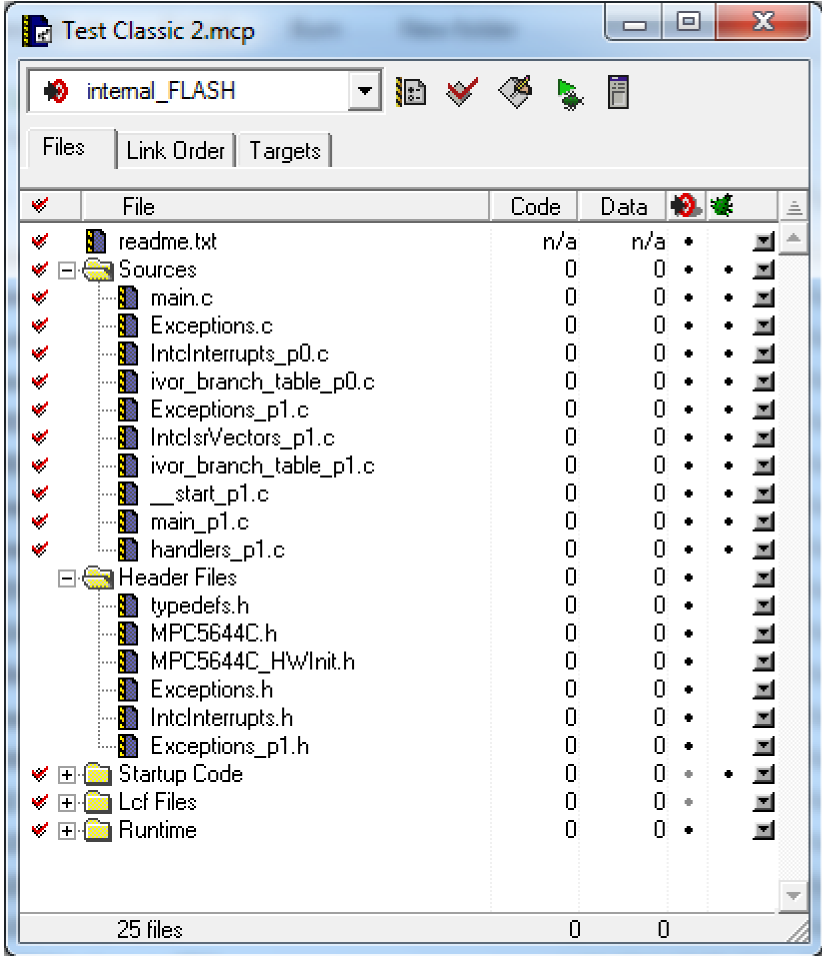

Figure 25. Multi-core project in CCW

3.18 Project Properties

Project properties in CCW are called *[Build Configuration] Settings*, for example, *internal_FLASH Settings* or *RAM Settings*.

You can access them by click the settings icon ( ). When you open settings, a dialog-box will appear. The options are different than CWE and S32DS, but for the purposes of project migration, make no practical difference. The CCW options window is shown in Figure 26.

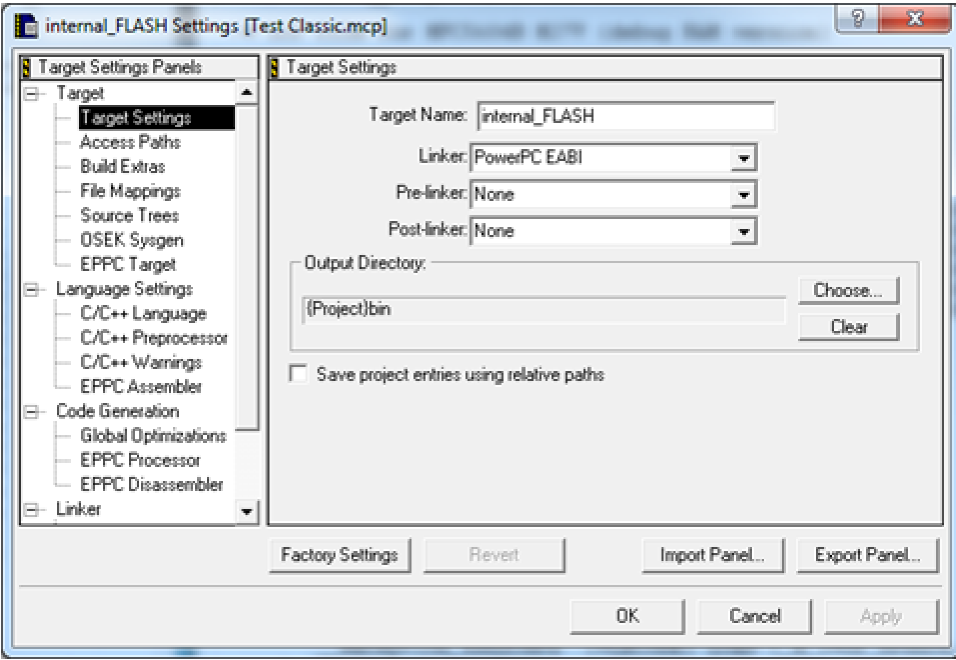

Figure 26. CCW Project Properties

3.19 Compiler and Debugger

CCW's *build* icon looks like a hand writing on paper ( ) and *debug* is the green arrow ( ). CCW uses a custom legacy NXP compiler and P&E-based debugger. As is the case for CWE, S32DS multi-core support is much better than in CCW. The figure below shows CCW's debugger setup.

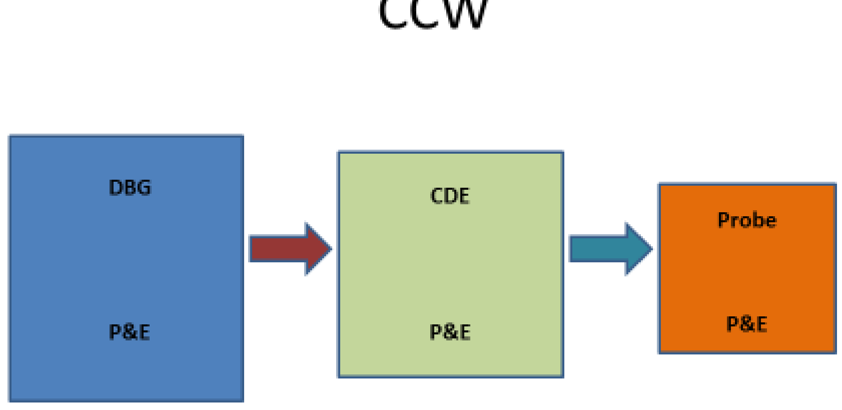

Figure 27. CCW Debugger Setup

4 Project Migration from CodeWarrior Eclipse for Power Architecture

To migrate a CWE project to S32DS, one must:

- Create a new project S32DS
- Copy the source files and headers over to the new project
- Incorporate them into the build
- Specify the include paths and symbols
- Compile the new project.

Create a new S32DS project by clicking on *File>New>New S32DS Project* (Figure 28).

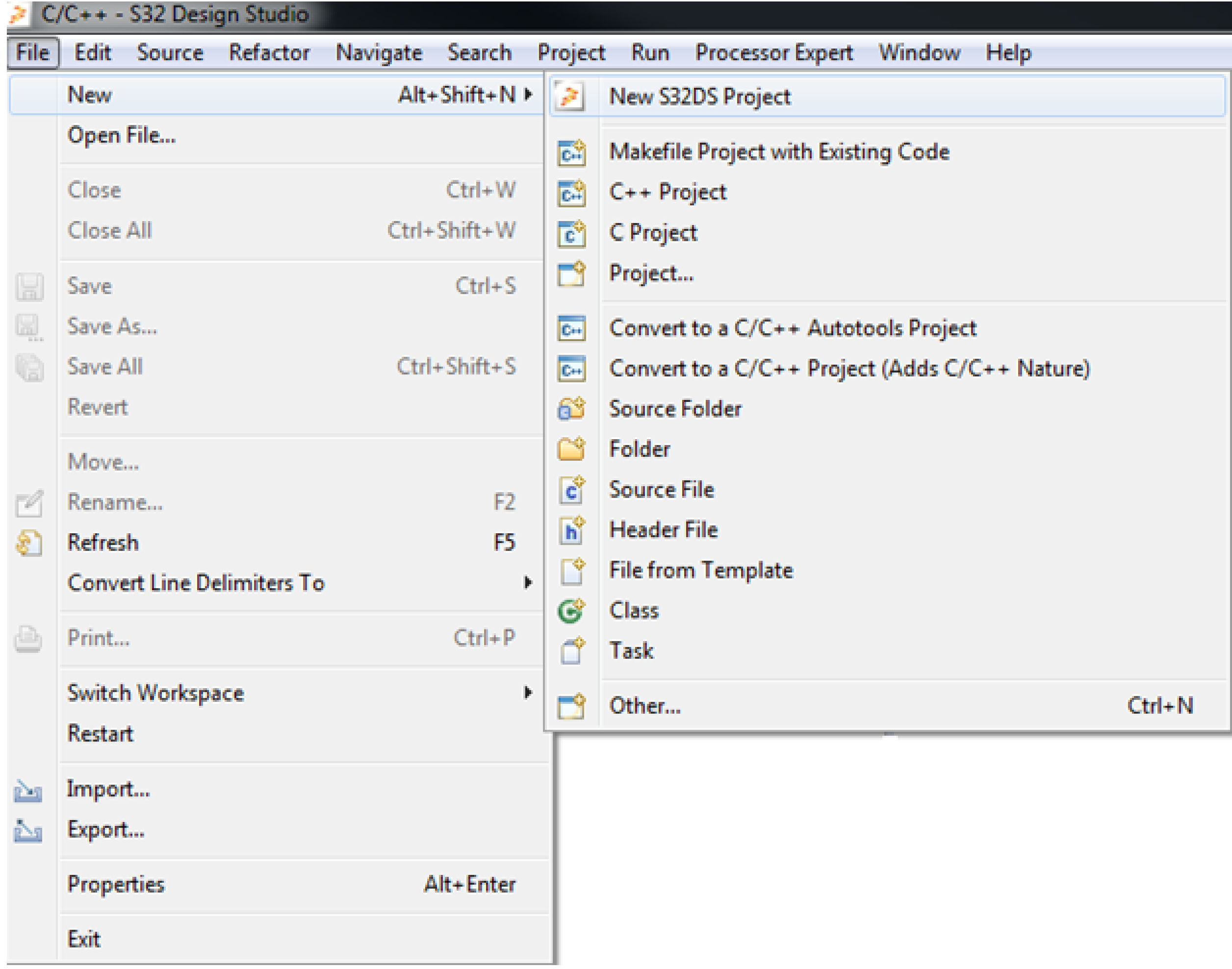

Figure 28. Creating a new S32DS project

Choose the MCU for the project you wish to port (Figure 29).

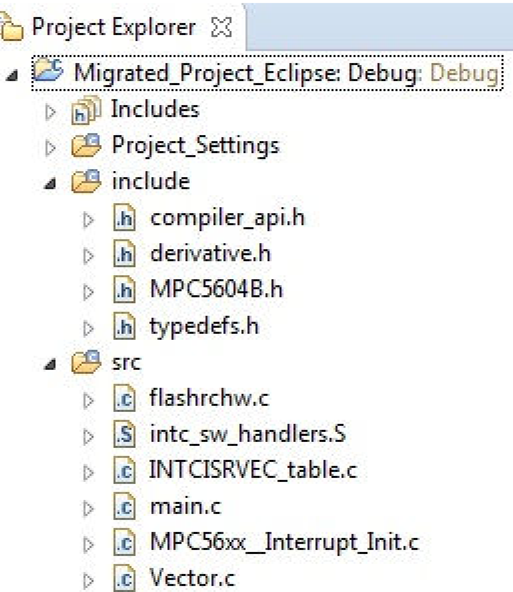

Figure 29. S32DS Project Creator

A single-core project like the MPC5604B will follow a largely identical organizational scheme to a CWE project. A multi-core one will generate one project for each core in use by the target device. From here the migration process for multi-core deviates slightly from single-core.

4.1 Headers and Sources

Move all user-written source files ,including *derivative.h* if it was modified, from CWE to S32DS. The appropriate location for the files in S32DS depends on whether the project is single-core or multi-core.

4.2 Headers and Sources, Single-Core

Open windows explorer and find your CWE project. As an example, this application note will move the following MPC5604B CWE project to S32DS (Figure 30).

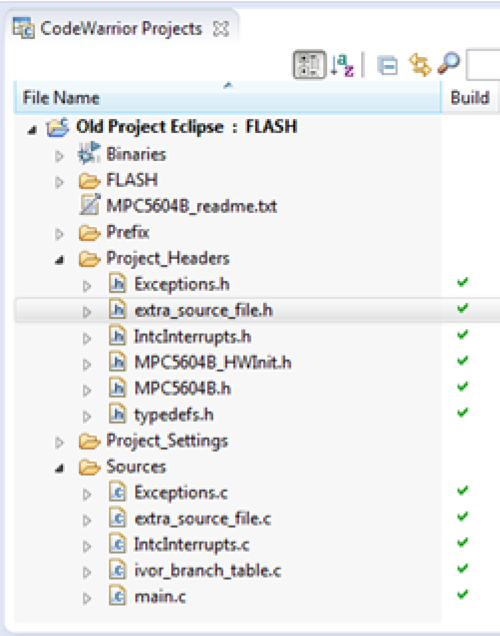

Figure 30. CodeWarrior Project to be migrated

Old Project Eclipse's is not a working project; it is just used as a demonstration for migrating to S32DS. The *main.c* file includes *extra_source_file.h*, registers an *RTC_ISR*, and makes a function call to *extra_function()*, which exists within *main.c* as well, shown in Figure 31.

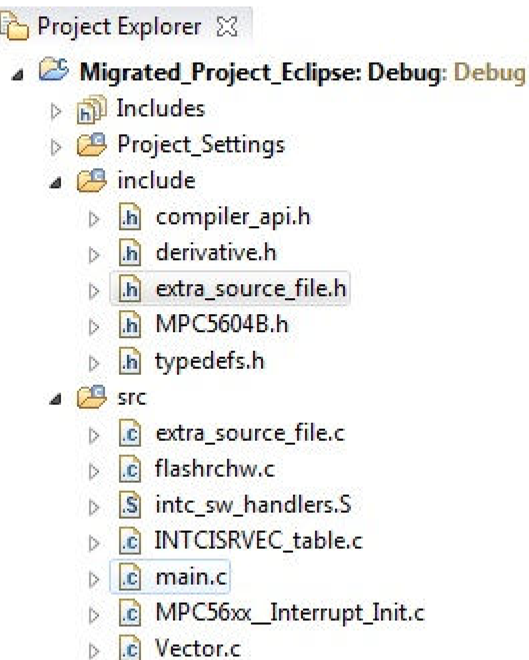

Figure 31. main.c of Old Project Eclipse

A new S32DS project will appear in the Project Explorer as in Figure 32.

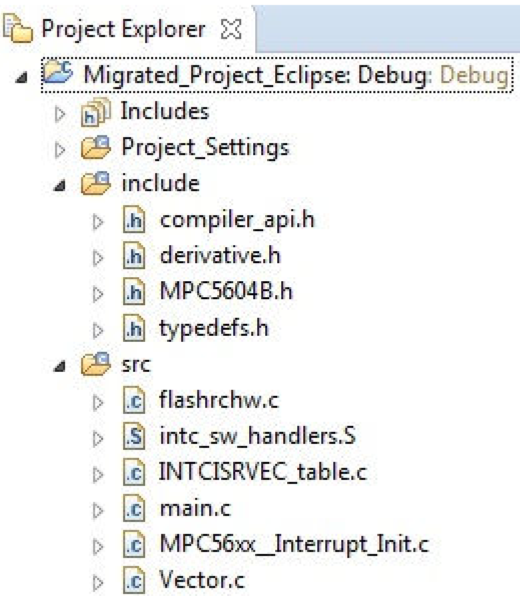

Figure 32. New S32DS project workspace

Copy all non-automatically generated header files within *Project_Headers* over to the *include* folder of S32DS. This can be done either by dragging and dropping the files into your project in the S32DS user interface or by copying them over to the directory and then refreshing the S32DS workspace (the refresh option is shown in Figure 33). In the case of *Old Project Eclipse*, *extra_source_file.h* would be moved over.

Next, do the same for the non-automatically generated source files *Sources* into *src*. Therefore, *extra_source_file.c* would be moved over to *src*.

```
main.c

 2⊕ * main implementation: use this 'C' sample to create your own application
 5
 6 #include "derivative.h" /* include peripheral declarations */
 7 #include "extra_source_file.h"
 8
 9 void extra_function(void);
10
11 void RTC_init(void);
12
13 void RTC_ISR(void);
14
15 extern void xcptn_xmpl(void);
16
17⊖void extra_function(){
18     int dummy=0;
19 }
20
21⊖void RTC_init(void){
22     int dummy=0;
23     /* You would enable RTC with interrupts here. */
24 }
25
26⊖void RTC_ISR(void){
27     int dummy=0;
28     /* Some function on RTC interrupt. */
29 }
30
31⊖int main(void)
32 {
33     volatile int i = 0;
34
35     RTC_init();
36
37     INTC_InstallINTCInterruptHandler(&RTC_ISR, 38, 15); //Install ISR into interrupt table
38
39     INTC.CPR.B.PRI = 0;           /* Single Core: Lower INTC's current priority */
40
41
42     /* Loop forever */
43     for(;;) {
44         i++;
45         extra_function();
46     }
47 }
```

Figure 33. Refresh for files copied into a directory

Your S32DS project should now look like the figure below in the Project Explorer.

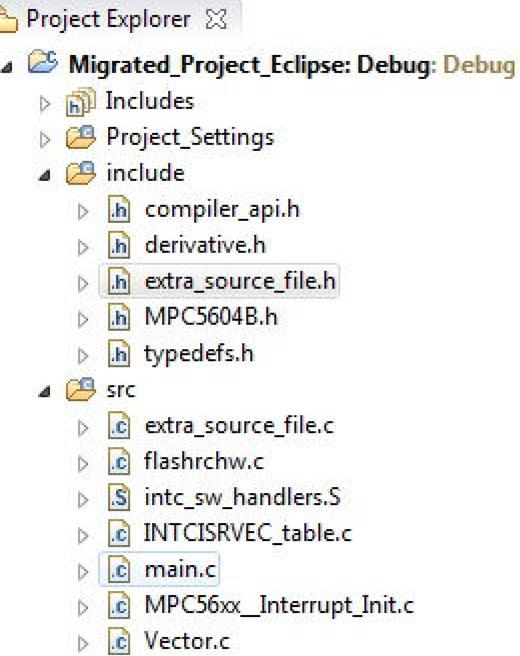

Figure 34. S32DS after source file migration

The file *main.c* cannot simply be copied over, since there are inherent differences between CWE and S32DS, such as how interrupts are handled. But first, copy all **user-written** includes, functions, and function prototypes as is. For *Old Project Eclipse*, that means copying over “#include “extra_source_file.h”, *RTC_ISR*, and *extra_function()*. *Migrated_Project_Eclipse* will look as follows. The figure below, is the main source file as it would appear at this point in the migration.

```
main.c
 2⊕ * main implementation: use this 'C' sample to create your own application
 5
 6 #include "derivative.h" /* include peripheral declarations */
 7 #include "extra_source_file.h"
 8
 9 void extra_function(void);
10
11 void RTC_init(void);
12
13 void RTC_ISR(void);
14
15 extern void xcptn_xmpl(void);
16
17⊖void extra_function(){
18     int dummy=0;
19 }
20
21⊖void RTC_init(void){
22     int dummy=0;
23     /* You would enable RTC with interrupts here. */
24 }
25
26⊖void RTC_ISR(void){
27     int dummy=0;
28     /* Some function on RTC interrupt. */
29 }
30
31⊖int main(void)
32 {
33     volatile int counter = 0;
34
35     xcptn_xmpl ();              /* Configure and Enable Interrupts */
36
37     /* Loop forever */
38     for(;;) {
39         counter++;
40     }
41 }
42
```

Figure 35. Functions in main.c migrated, except for main() itself

Special care must be taken for *main()*. In CWE, ISRs must be registered into the interrupt vector table via an automatically generated function from the file *IntcInterrupts.c* called *INTC_InstallINTCInterruptHandler*, which adds the interrupt to the table and sets its priority level. *Old Project Eclipse* shows such an example with a real-time counter (RTC) interrupt. The *main.c* file of *Old Project Eclipse*. m*ain()* initializes the RTC then installs the ISR, which is named *RTC_ISR*. S32DS implements ISRs in flash, meaning you must compile the project with the ISR already in the vector table.

Replace the *main()* of S32DS with that of the CWE project. The image below shows *Migrated_Project_Ecliplse's main()* overwritten with *Old Project Eclipse*'s. Make sure the line "__attribute__ ((section(".text")))" stays. This ensures that main.c goes to the *.text* section during compilation. S32DS should look as in Figure 36 after this step.

```
main.c

 2⊕ * main implementation: use this 'C' sample to create your own application
 5
 6  #include "derivative.h" /* include peripheral declarations */
 7  #include "extra_source_file.h"
 8
 9  void extra_function(void);
10
11  void RTC_init(void);
12
13  void RTC_ISR(void);
14
15  extern void xcptn_xmpl(void);
16
17⊖ void extra_function(){
18      int dummy=0;
19  }
20
21⊖ void RTC_init(void){
22      int dummy=0;
23      /* You would enable RTC with interrupts here. */
24  }
25
26⊖ void RTC_ISR(void){
27      int dummy=0;
28      /* Some function on RTC interrupt. */
29  }
30
31⊖ int main(void)
32  {
33      volatile int i = 0;
34
35      RTC_init();
36
37      INTC_InstallINTCInterruptHandler(&RTC_ISR, 38, 15); //Install ISR into interrupt table
38
39      INTC.CPR.B.PRI = 0;            /* Single Core: Lower INTC's current priority */
40
41
42      /* Loop forever */
43      for(;;) {
44          i++;
45          extra_function();
46      }
47  }
```

Figure 36. main() copied but not yet integrated

The act of pasting over the original contents of *main()* will have overwritten the function call to *xcptn_xmpl()*, unless you specifically took care to save the instruction beforehand. This function is needed because it initializes the S32DS INTC. *xcptn_xmpl()* must be written back in. The best location for the function depends on the contents of each project, but it must be placed before any user-written initialization sequence configures an interrupt—you cannot write to an INTC that is not configured. Then comment out the function calls for *INTC_InstallINTCInterruptHandler*, as it does not exist in S32DS. Figure 37 shows what *Migrated Project Eclipse* looks like after this step.

```c
2  * main implementation: use this 'C' sample to create your own application
5
6  #include "derivative.h" /* include peripheral declarations */
7  #include "extra_source_file.h"
8
9  void extra_function(void);
10
11 void RTC_init(void);
12
13 void RTC_ISR(void);
14
15 extern void xcptn_xmpl(void);
16
17 void extra_function(){
18     int dummy=0;
19 }
20
21 void RTC_init(void){
22     int dummy=0;
23     /* You would enable the RTC with interrupts here. */
24 }
25
26 void RTC_ISR(void){
27     int dummy=0;
28     /* Some function on RTC interrupt. */
29 }
30
31 __attribute__ ((section(".text")))
32 int main(void)
33 {
34         volatile int i = 0;
35
36         xcptn_xmpl(); //Enable interrupts
37
38         RTC_init();
39
40         //INTC_InstallINTCInterruptHandler(&RTC_ISR, 38, 15); //Install ISR into interrupt table
41
42         INTC.CPR.B.PRI = 0;          /* Single Core: Lower INTC's current priority */
43
44         /* Loop forever */
45         for (;;) {
46           i++;
47           extra_function();
48         }
```

Figure 37. main.c, xcptn_xmpl() is back and INTC_InstallINTCInterruptHandler gone

Now that *INTC_InstallINTCInterruptHandler* has been removed, you need to register your interrupts the S32DS way. Go to the file *INTCISRVEC_table.c*. In it is an array of function pointers called *IntcIsrVectorTable*. S32DS by default populates the array with "&dummy", which is defined within *INTCISRVEC_table.c* as an infinite loop, shown in the figure below.

```c
20 #include "typedefs.h"
21 /*========================================================================*/
22 /*      PROTOTYPES                              */
23 /*========================================================================*/
24 void dummy                         (void);
25
26 /*========================================================================*/
27 /*  GLOBAL VARIABLES                                */
28 /*========================================================================*/
29
30 const uint32_t __attribute__ ((section (".intc_vector_table"))) IntcIsrVectorTable[] = {
31
32 (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, /* ISRs 00 - 04 */
33 (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, /* ISRs 05 - 09 */
34 (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, /* ISRs 10 - 14 */
35 (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, /* ISRs 15 - 19 */
36 (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, /* ISRs 20 - 24 */
37 (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, /* ISRs 25 - 29 */
38 (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, /* ISRs 30 - 34 */
39 (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, /* ISRs 35 - 39 */
40 (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, (unsigned int)&dummy, /* ISRs 40 - 44 */
```

Figure 38. Default INTC Vector table.

This function, *dummy()*, serves as a default handler: any interrupt that does not have a defined, user-written ISR will branch to *dummy()* and loop there forever (Figure 39).

```
238 __attribute__ ((section(".text")))
239 /*========================================================================*/
240 /*          FUNCTIONS                                                     */
241 /*========================================================================*/
242
243 /************************************************************************/
244 /* FUNCTION       : dummy                                               */
245 /* PURPSE         : Dummy function which is entered if any non-used vector */
246 /*                  is called.                                          */
247 /************************************************************************/
248 void dummy (void) {
249
250     while (1){};
251
252 }
253
```

Figure 39. dummy(), the default handler, an infinite loop

The prototype for *dummy()* is located at the top of the file. For each interrupt, include the header files that contain your ISRs, *extern* the ISR prototype, and, at the array position of the interrupt's vector offset value, write "(unsigned int)&" followed by the name of the relevant ISR. In the case of *Old Project Eclipse*, the RTC interrupt is implemented. The RTC offset is 38 and the RTC ISR is named *RTC_ISR*, which is located in the main source file. Therefore, "extern void RTC_ISR(void);" is written below the *dummy()* prototype and "&RTC_ISR" replaces "&dummy" at *IntcIsrVectorTable[38]*, as shown below. Since *RTC_ISR* is located in the main source file, there is no need to include another file. Had *RTC_ISR* existed in another source file, say an *RTC.c*, then its companion header, *RTC.h*, would have needed to have been included. So you would have seen a '#include "RTC.h"' below line 20 in the figure below.

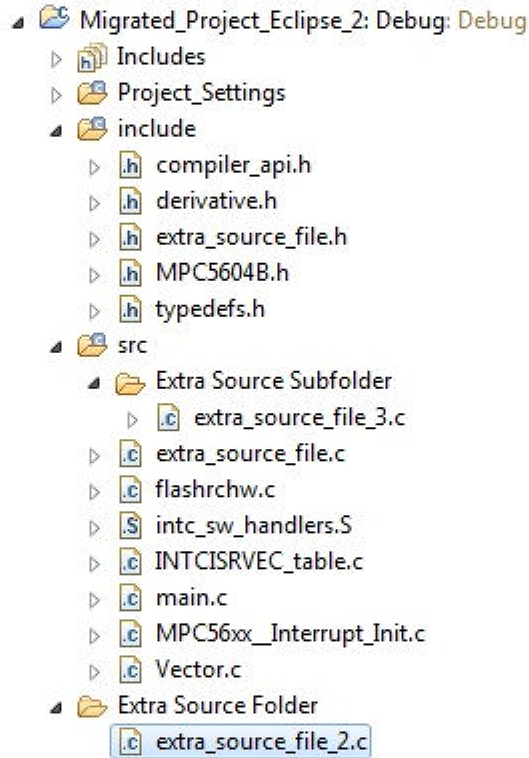

Figure 40. INTCISRVEC_table.c with RTC_ISR

INTC_InstallINTCINterruptHandler() in CWE installs an ISR into the interrupt vector table and records its interrupt priority. The steps just described in the previous paragraphs installed the ISR into S32DS. Now we need to manually record the priority. Return to *main()* and for each interrupt add the priority with the line "INTC.PSR[*Vector Offset*].R = *Priority number*". The PSR stands for Priority Select Register. They come in an array in Power Architecture design; the MPC5604B comes with 512. Each PSR register corresponds to the interrupt whose offset value is the same as the PSR array index. So *Old Project Eclipse*'s RTC interrupt, whose vector offset value is 38, corresponds to *PSR[38]*. Therefore in *main()*, the line

“INTC.PSR[38].R=15;” must be added to set a priority level for the RTC interrupt (Figure 41). This example uses 15, which is the highest priority possible, but any value that is greater than the value of *INTC.CPR.B.PRI* will work. *CPR* stands for Current Priority Register and the CPU will not respond to any interrupts whose priority is lower than the value of *CPR*.

```
30
31 __attribute__ ((section(".text")))
32 int main(void)
33 {
34       volatile int i = 0;
35
36       xcptn_xmpl(); //Enable interrupts
37
38       RTC_init();
39
40       //INTC_InstallINTCInterruptHandler(&RTC_ISR, 38, 15); //Install ISR into interrupt table
41
42       INTC.CPR.B.PRI = 0;          /* Single Core: Lower INTC's current priority */
43       INTC.PSR[38].R = 15; //Set priority level of RTC interrupt
44
45       /* Loop forever */
46       for (;;) {
47         i++;
48         extra_function();
49       }
50 }
```

Figure 41. main() with RTC interrupt priority set

If your CWE project did not modify the folder structure, and you only added source files and headers into *Sources* and *Project_Headers*, respectively, this would be all that you would need to do. Just compile the project at this point and you will have successfully migrated your CWE project to S32DS.

However, if your CWE project contains extra folders with source files for, say, organizational purposes, you would have to copy and include them in the build. Take this second example project (Figure 42). It is exactly the same as *Old Project Eclipse*, but with an extra folder of source files at the top level and a subfolder within S*ources*.

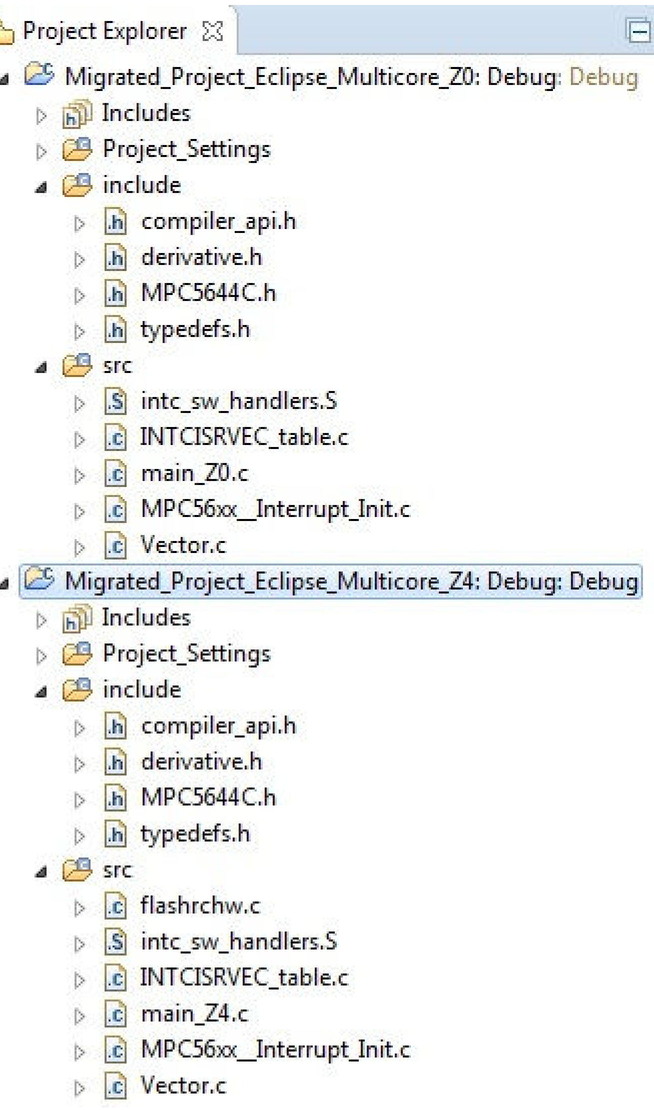

Figure 42. CW project with extra folders

Old Project Eclipse 2 has *Extra Source Folder* at the top level with a source file and *Extra Source Subfolder* under *Sources*, also with its own source file. These folders must be copied to S32DS in the same location relative to the project directory. Your S32DS project should look like Figure 43 once they are copied over.

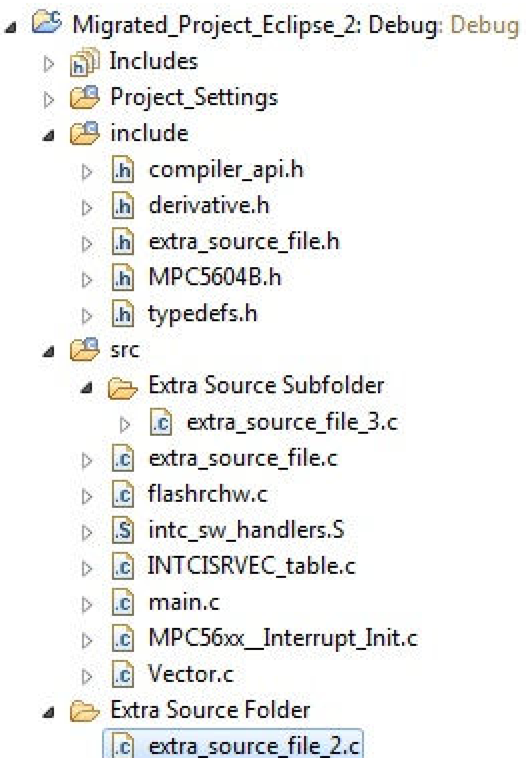

Figure 43. S32DS Project with extra folders copied over

Your files are now copied to S32DS, but the lack of a blue 'C' in their folder icon means they are not treated as source folders and therefore are not part of the build. For now, the folders (and the files in them) are just sitting in the workspace; S32DS does not know what to do with them. This project is revisited in a later section to go over how to integrate Extra Source Folder into the project.

4.3 Headers and Sources, Multi-Core

Source code migration on a multi-core project is quite similar to the single-core procedure. As discussed earlier, a multi-core CWE project separates core-specific source files into separate folders. This example shows a dual-core MPC5644C CWE project, called *Old Project Eclipse Multicore*, with one extra source file for each core, named *extra_source_file_p0* and *extra_source_file_p1*, respectively (Figure 44).

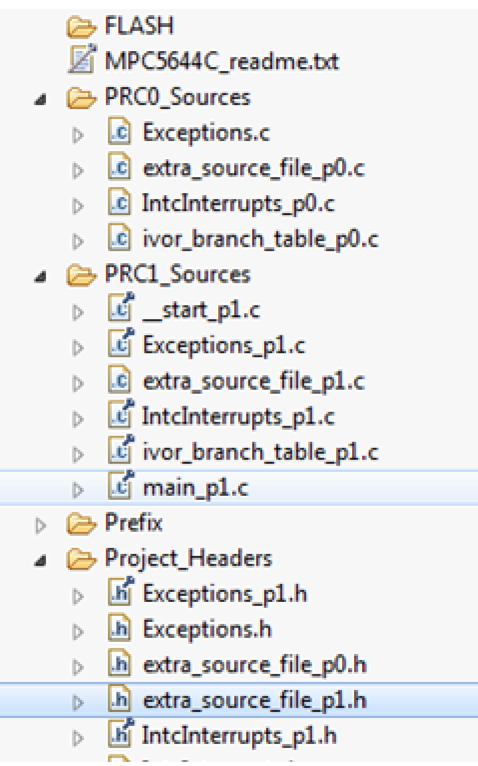

Figure 44. Generic MPC5644C CWE project

S32DS splits *Migrated Project Eclipse Multicore* into project into two standalone projects, named *Migrated_Project_Eclipse_Multicore_Z4_1* and *Migrated_Project_Eclipse_Multicore_Z4_1*, as shown in Figure 45.

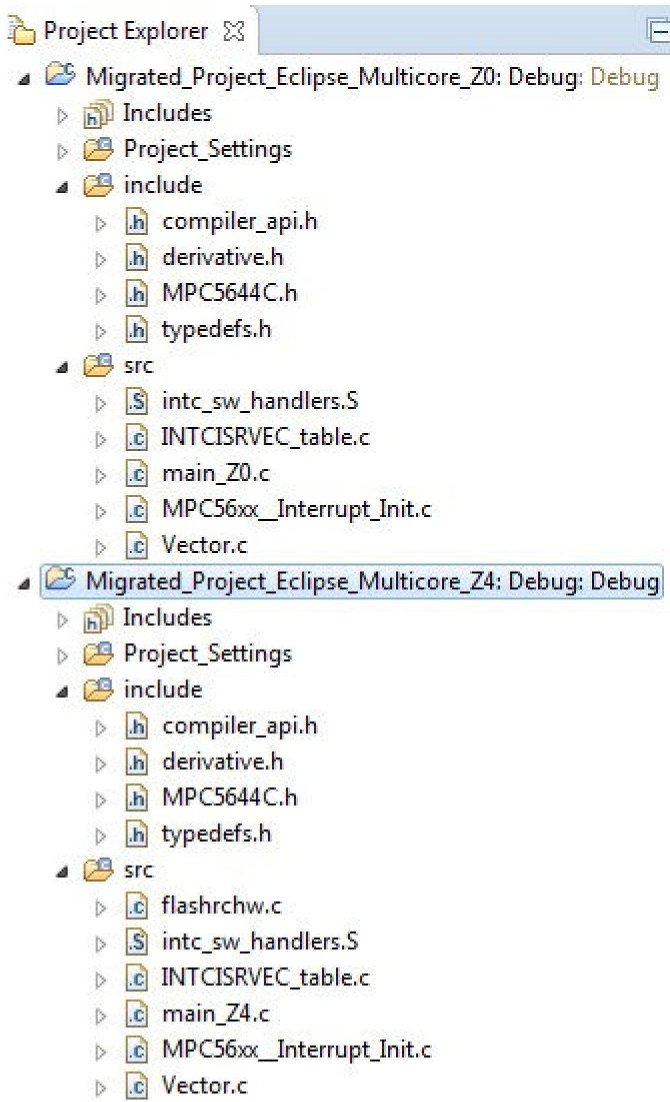

Figure 45. Generic MPC5644C S32DS project

From here follow the same migration steps as for a single-core project, but make sure to transfer the right files to the right projects. Note that the CWE index starts at 0 (e.g. p0) and S32DS starts at 1. Therefore in this example, *extra_source_file_p0* would belong in *Migrated_Project_Eclipse_Multicore_Z4_1* and *extra_source_file_p1* would go to *Migrated_Project_Eclipse_Multicore_Z0_2* (Figure 46).

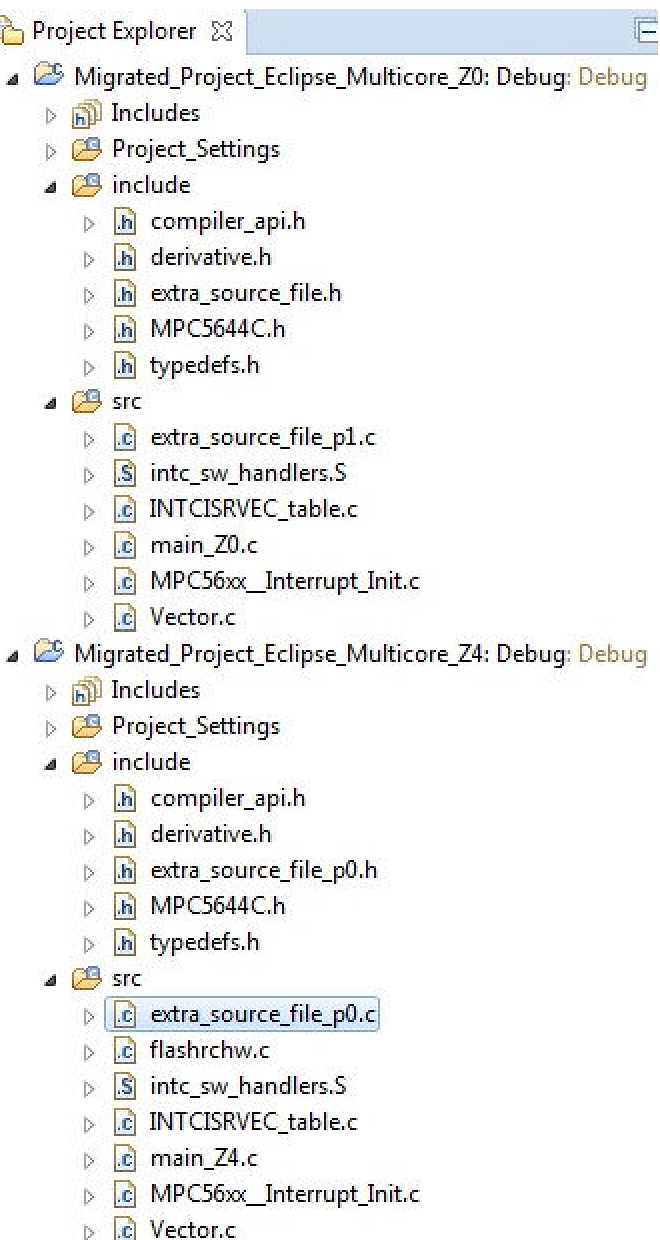

Figure 46. S32DS MPC5644C project with files properly parsed

With the exception of the extra file parsing, the other difference you must be careful for is how the first core starts the subsequent ones. In the example, *Old Project Eclipse Multicore* and *Migrated_Project_Eclipse_Multicore* are configured for the MPC5644C, which is a dual-core device. Shown in Figure 47 are the respective functions CodeWarrior and S32DS use to start subsequent cores.

```
int main(void) {
  volatile int i = 0;

#if ROM_VERSION == 1
/* Start the second core */
SSCM.DPMBOOT.R = (unsigned long)__start_p1;
SSCM.DPMKEY.R = 0x00005AF0;
SSCM.DPMKEY.R = 0x0000A50F;
#endif

  /* Loop forever */
  for (;;) {
    i++;
  }
}
```

```
6  #include "derivative.h" /* include peripheral declarations */
7
8  extern void xcptn_xmpl(void);
9
10 void hw_init(void)
11 {
12 #if defined(TURN_ON_CPU1)
13     /* Start the second core */
14     SSCM.DPMBOOT.R = (unsigned long)0xc0000;
15     SSCM.DPMKEY.R  = (unsigned long)0x00005AF0;
16     SSCM.DPMKEY.R  = (unsigned long)0x0000A50F;
17 #endif
18 }
19
20 __attribute__ ((section(".text")))
21 int main(void)
22 {
23     int counter = 0;
24
25     xcptn_xmpl ();              /* Configure and Enable Interrupts */
26
27     for(;;) {
28         counter++;
29     }
30
31     return 0;
32 }
33
```

Figure 47. How to initialize the second core in a) CWE and b) S32DS

They are instructions, one to boot the second core and two to unlock the System Status and Configuration Module, but CWE implements it in *main()* while S32DS stores the instructions in its own standalone function called *hw_init()*, which is called by the startup script in *startup.S*. Do not carry over the CWE version; use *hw_init()* instead when migrating a multi-core CWE project. It is important to remember that any file that is shared by the core-specific source files will need to be copied over to **every** S32DS core project. The fact that the S32DS projects reference each other does not mean the project for Core 0 will be able to reference functions from a source file in the Core 1 project.

All other migration steps for single-core projects apply to multi-core ones too.

4.4 Include Paths

S32DS does not support recursive include directories. Therefore every subfolder that contains header files must be added to the list of include paths. Go to project properties. In the properties window, go to *C/C++ Build>Settings>Settings>Standard S32DS C Compiler>Includes>Include paths*. The directories specified in the *Include Paths* window are the locations that the linker searches when the *#include* macro is implemented. The *ewl* include paths have to do with the NXP Embedded Warrior compiler library, as shown in Figure 48.

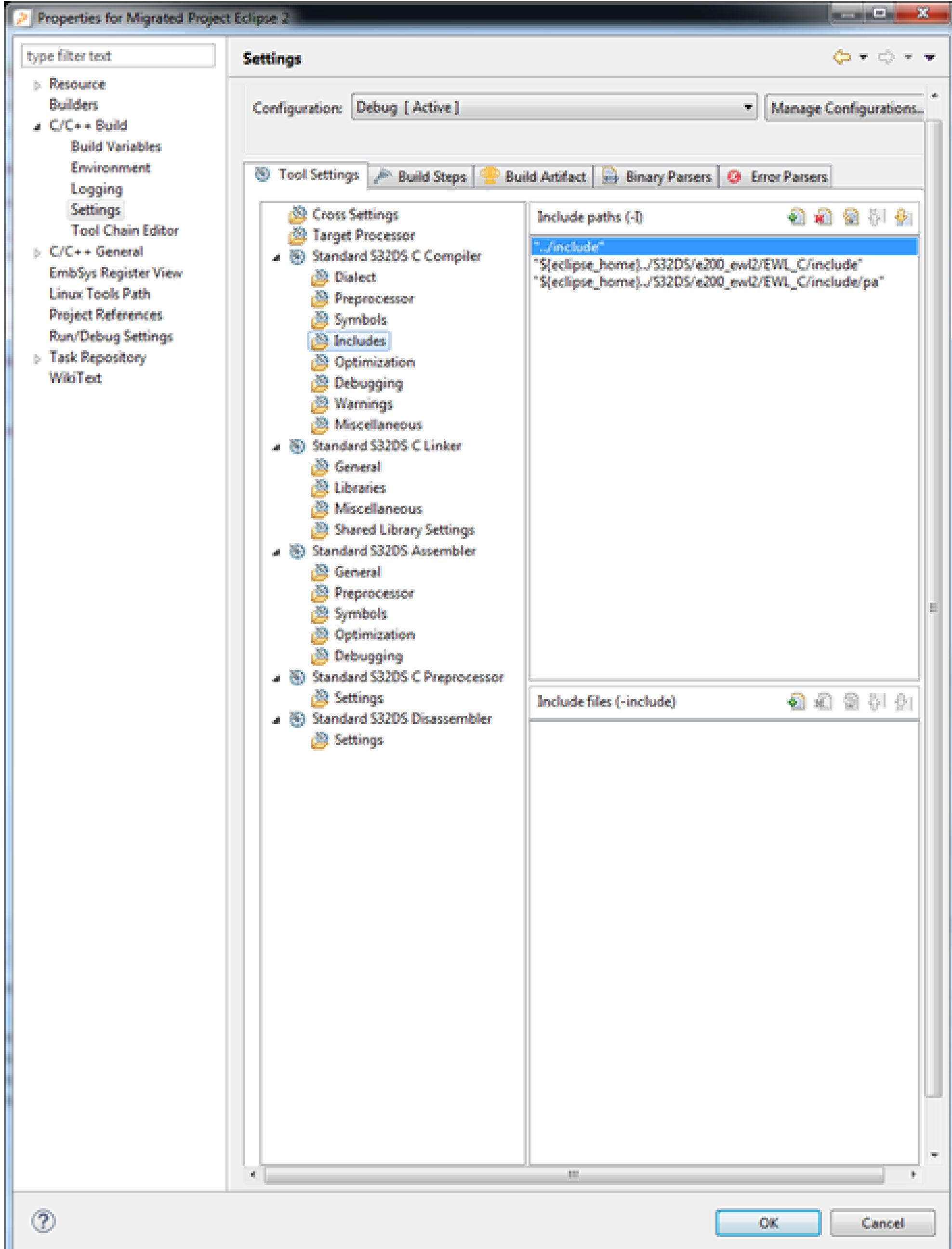

Figure 48. Include paths in S32DS

This option is where you specify your include paths. In this example, *Extra Source Folder* and *Extra Source Subfolder* need to be included. Click on the “add” button ( ) and add the folders one by one. The following dialog-box will appear (Figure 49).

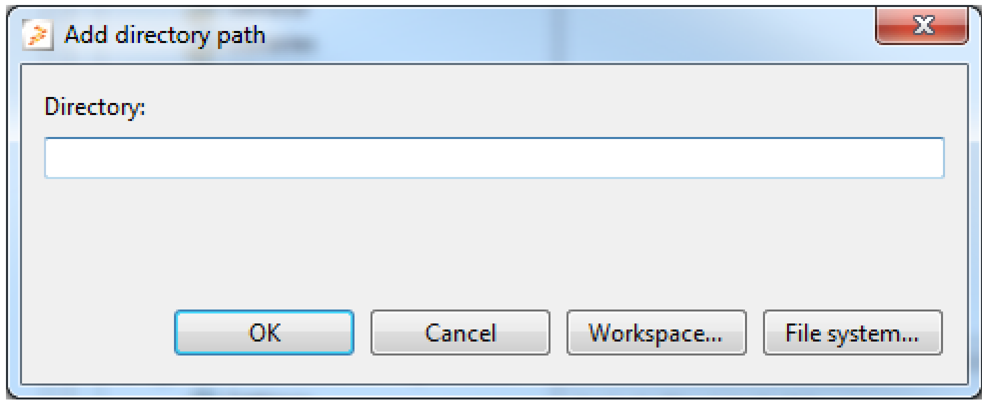

Figure 49. Adding include paths

Click on *Workspace* and another window will appear (Figure 50). Select *Extra Source Folder* and hit *OK* in the window displayed below.

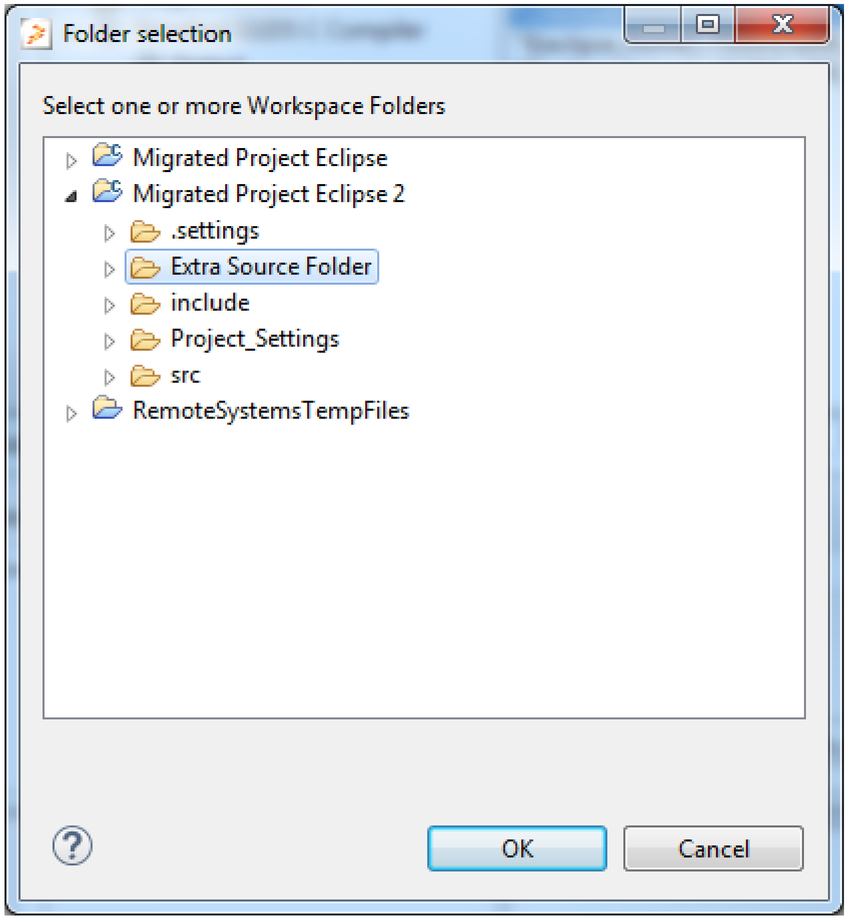

Figure 50. Folder Selector

Repeat this step for all other folders that were copied over, in the case of this example, *Extra Source Subfolder*. Folders you select with the workspace option will use relative paths. This is recommended over the *File System* option, since *Workspace* makes your paths non-machine-specific. Thus, your project will be more portable between computers. After adding *Extra Source Folder* and *Extra Source Subfolder* to *Migrated_Project_Eclipse_2's* set of include paths, the Properties window should look as shown in Figure 51 :

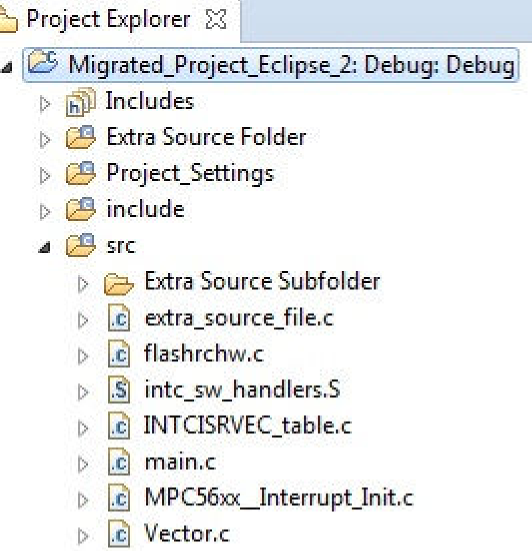

Figure 51. Include paths with folders added

S32DS will now know where to look when you call functions from source files that reside in those folders. However, there remains one last problem: S32DS does not know it needs to build the folders even though it has been made aware of them.

4.5 Build Configurations

To tell S32DS to actually use the folders that have been included, right-click and select *Build Configurations Explorer*. The menu option is shown in Figure 52.

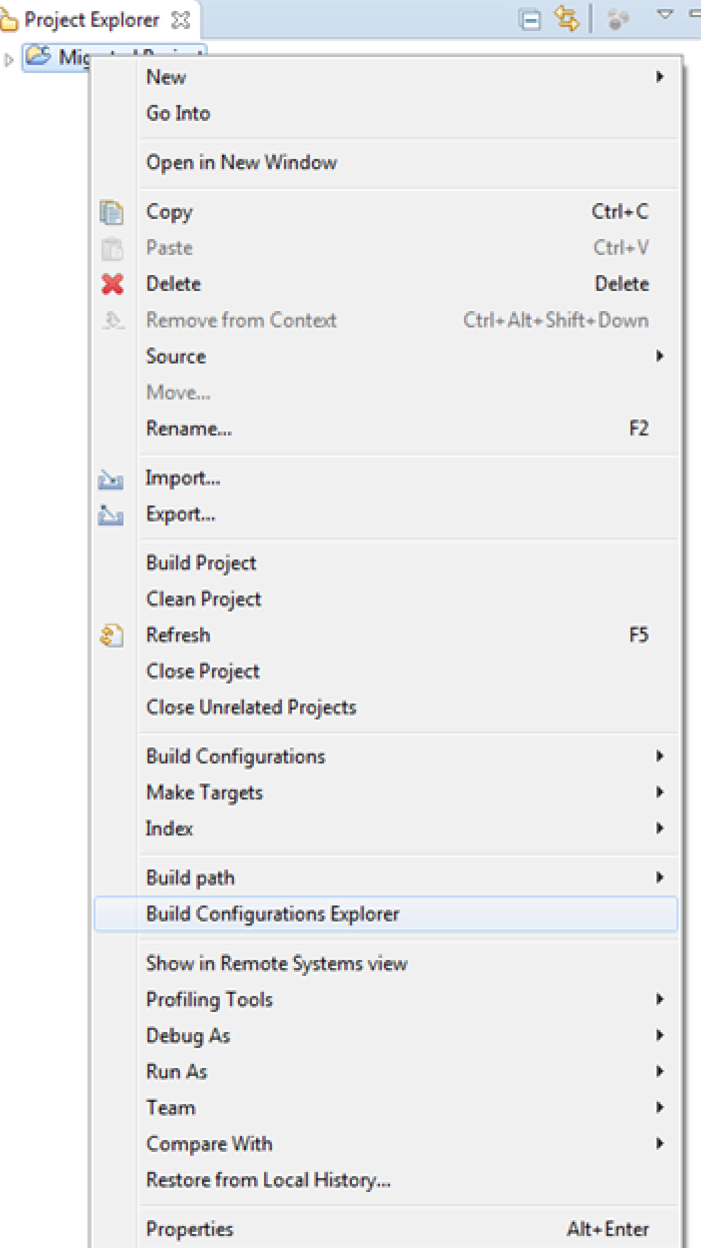

Figure 52. Build Configurations

The window as shown in the figure below will be displayed, which shows what is included in the build when you compile the project.

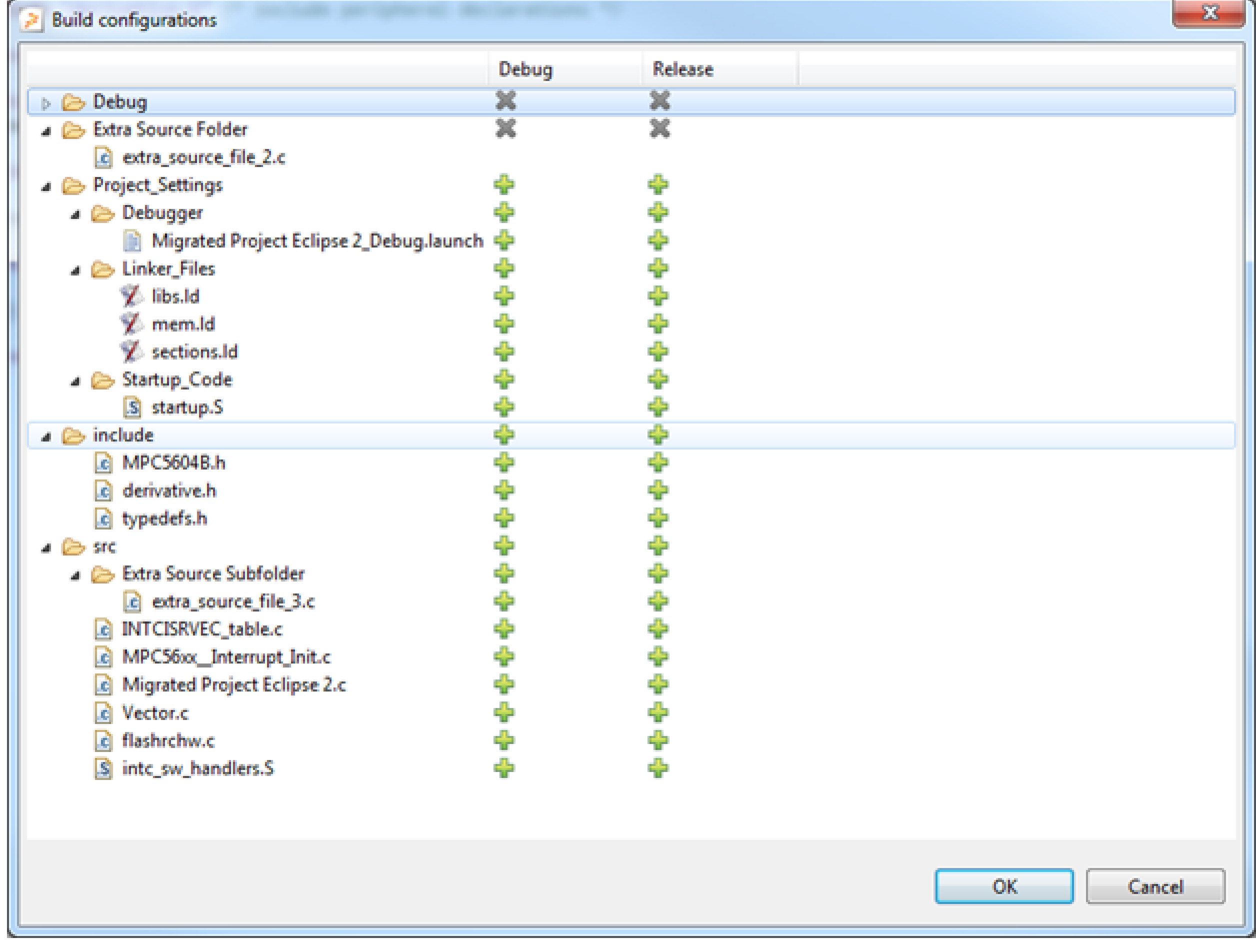

Figure 53. Build Configuration Window

The ( ) icon means the corresponding folder/file is included in the build for the corresponding build mode (Debug, Release, or Debug_RAM). In this example, *Extra Source Folder* has a ( ) in all three build modes so it is currently never included during compilation. Clicking on ( ) the will turn it into a ( ) to include the folder for all three build modes. Then hit *OK*. Figure 54 shows the Build Configurations explorer after the change.

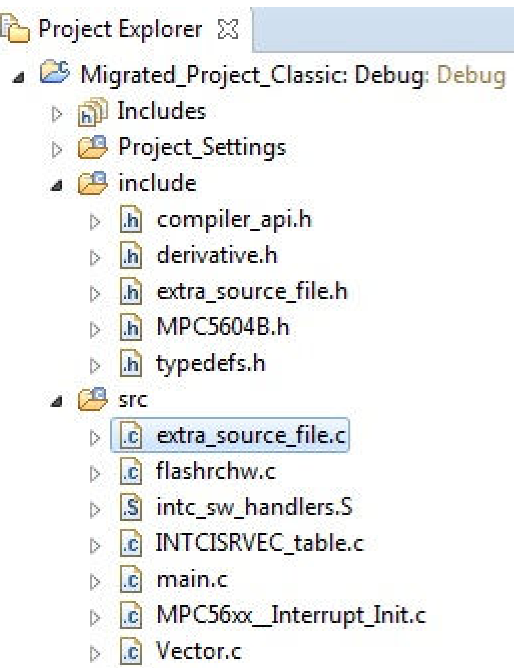

Figure 54. Build config window with extra folders included

Extra Source Folder now has a blue 'C' in its folder icon, which means S32DS now sees it as a source folder and will incorporate it into the build. Subfolders do not get the blue 'C' icon so even though *Extra Source Subfolder* has been included, as a subfolder under *src*, no blue 'C' appears (Figure 55).

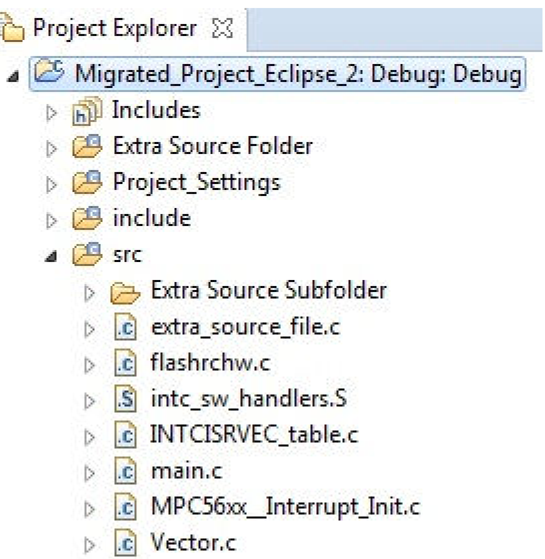

Figure 55. S32DS Workspace with extra folder fully integrated

4.6 Libraries

If your CodeWarrior project has an application library, such as the NXP Math and Motor Control Library, you can link it to S32DS. Open project properties and go to *C/C++ Build>Settings>Settings>Standard S32DS C Linker>Miscellaneous>Other Objects*. *Standard S32DS C Linker>Libraries* is for linker libraries, not application

libraries.Make sure, also, that the library you wish to include is GCC-compliant. Add the library's path in the same way the folders were included. If the .a file is not in your workspace, find it through the file system. Figure 56 shows a project that includes the NXP Automotive Math and Motor Control Library.

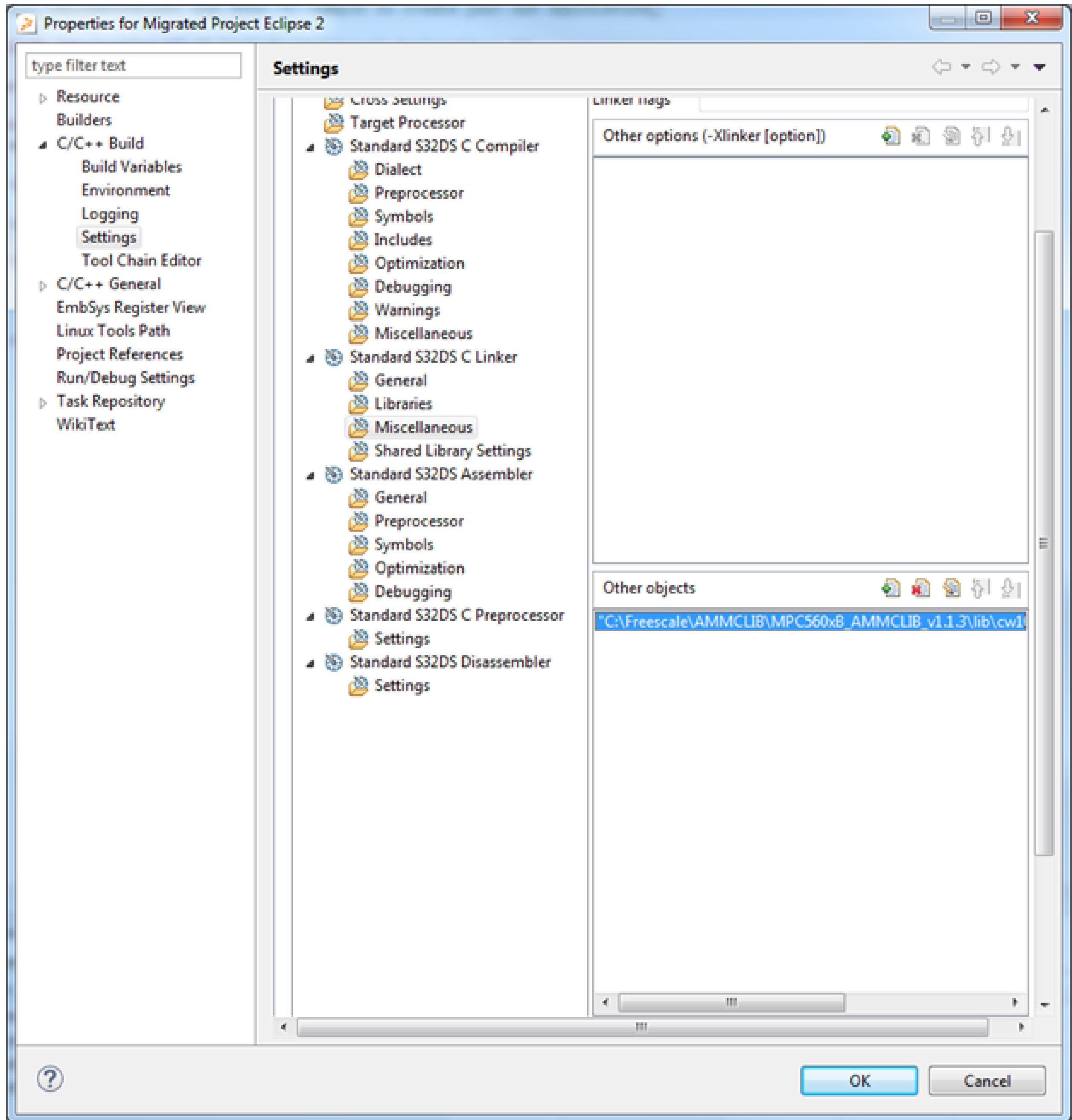

Figure 56. Where to link your application library

NOTE

If you export your project to another computer, that computer will need to have the library in the same location

4.7 Symbols

Be sure to add any predefined symbols you have in your CodeWarrior project to S32DS. Go to project properties and navigate to *C/C++ Build>Settings>Settings>Standard S32DS C Compiler>Symbols>Defined Symbols*. Click on the button and add the symbols. The two that are populated by default specify the device name and where to load the code from (flash or RAM). Figure 57 shows the symbols window of the S32DS project settings.

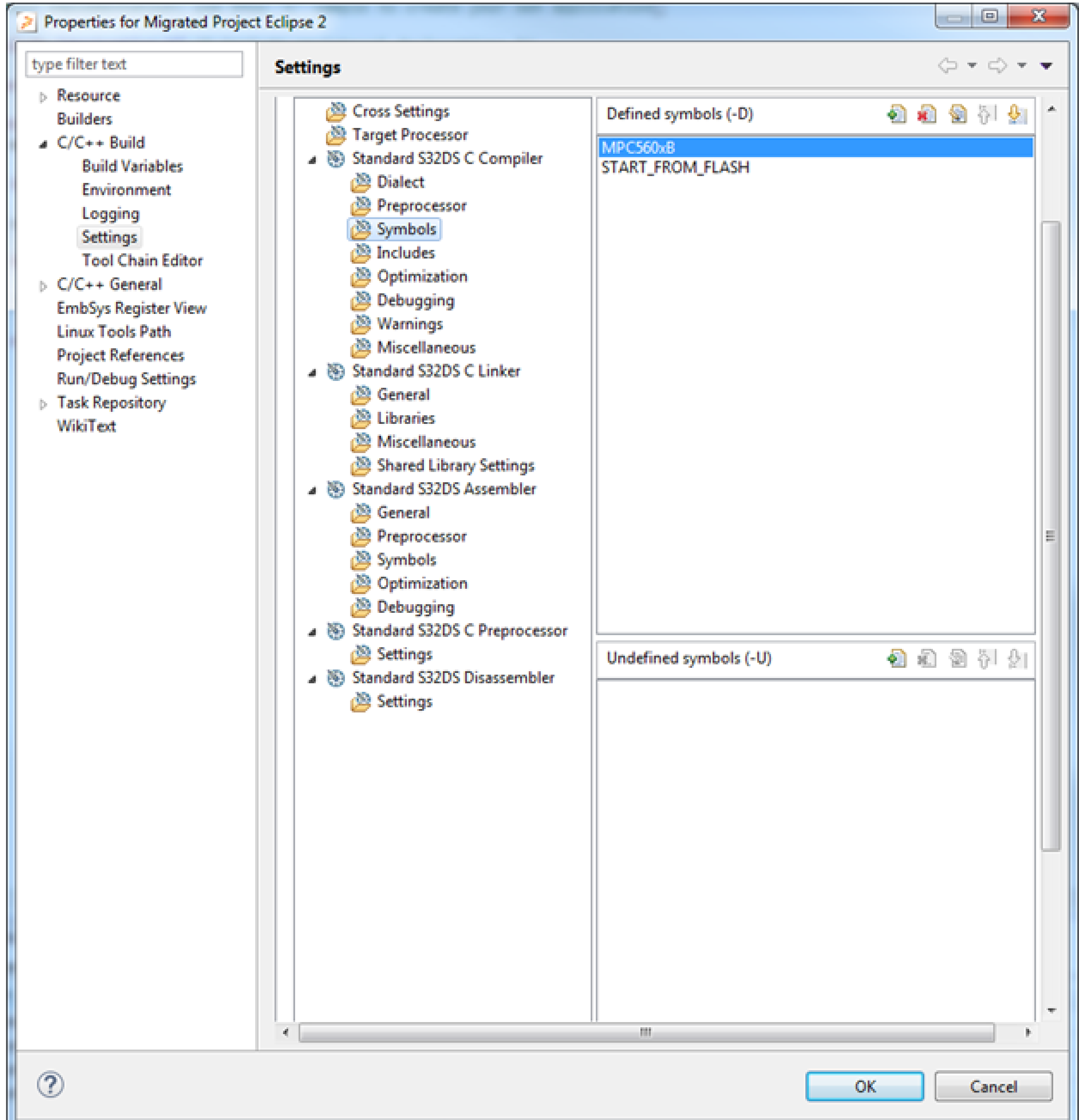

Figure 57. Symbol table in S32DS

5 Project Migration from CodeWarrior Classic for Power Architecture

Migration from CodeWarrior Classic is similar to the process for CWE, because even though it is built on a different framework, the source files are the same. The following example migrates an MPC5604B CCW project called *Old Project Classic* to a blank S32DS MPC5604B project called *Migrated_Project_Classic*. The modifications done to *Old Project Classic* are the same as those done for *Old Project Eclipse*: there is an additional header and source file called *extra_source_file.h/.c*, as shown in Figure 58.

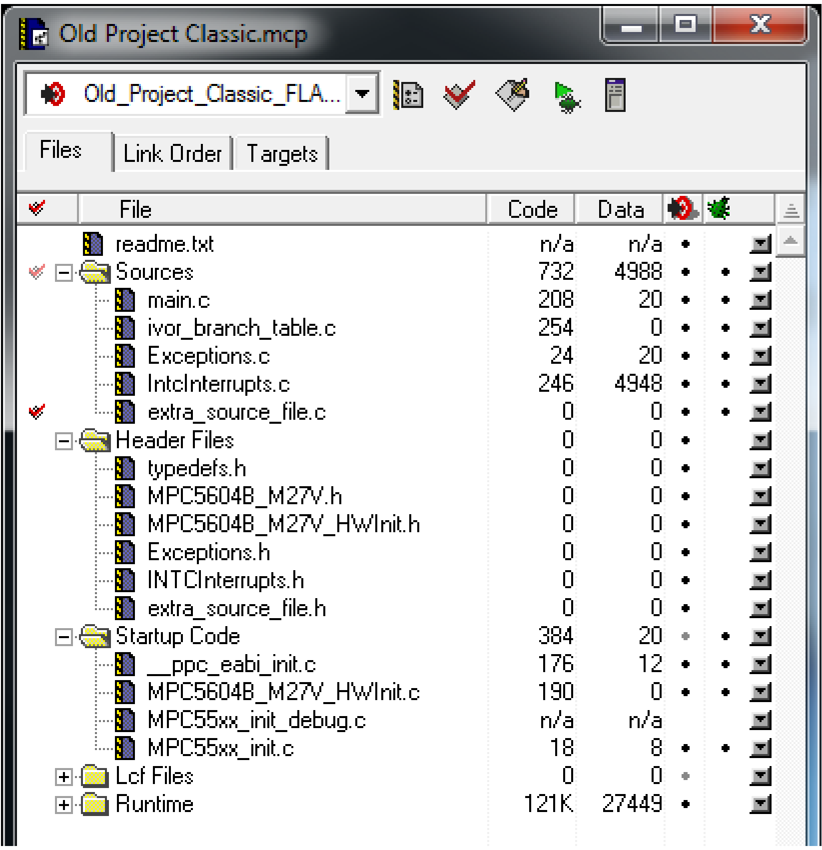

Figure 58. Old Project Classic

The difference you must be aware of is that CCW does not support copying and pasting source files from the workspace. You have to go to the file directory, copy the file, and then paste it into the S32DS project. The organization in a CCW workspace does not reflect how the files are actually arranged in the hard drive. All header and source files, by default, live in the *src* folder, which is located in the project root directory (Figure 59).

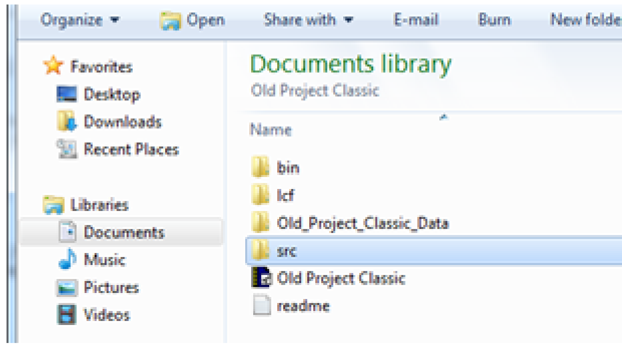

Figure 59. Where to find files in CCW project

Migrated_Project_Classic ultimately looks the same as *Migrated_Project_Eclipse*, shown in Figure 60.

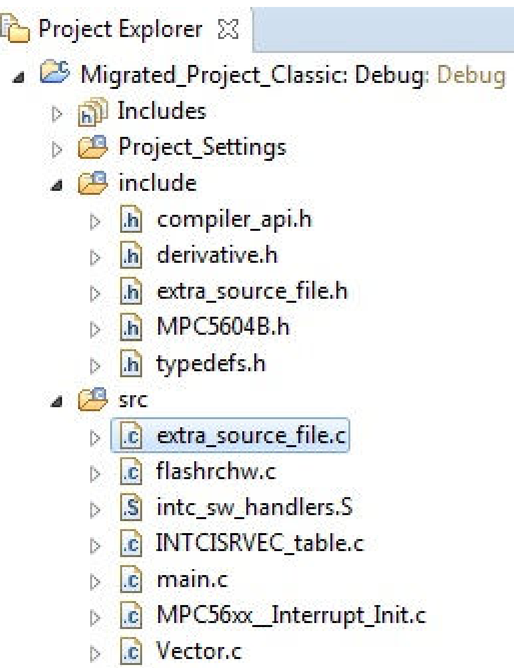

Figure 60. Migrated Project Classic

A multi-core project is much the same, except you have more files to move and need to make sure you switch to the S32DS implementation second-core-initialization.

6 Assembly Translation

S32DS for Power v1.1 and newer supports BookE to VLE translation. S32DS builds in the VLE instruction set and S32DS for Power initially could not recognize BookE instructions. Attempting to compile a project with BookE instructions then would cause build errors. However starting with S32DS for Power v1.1, S32DS can translate BookE to VLE during compilation, once certain steps are taken. The following example uses a MPC5604B project called *Test_with_ASM*. It is an empty project with the addition of an assembly file called *booke.S*, as shown in the following figure.

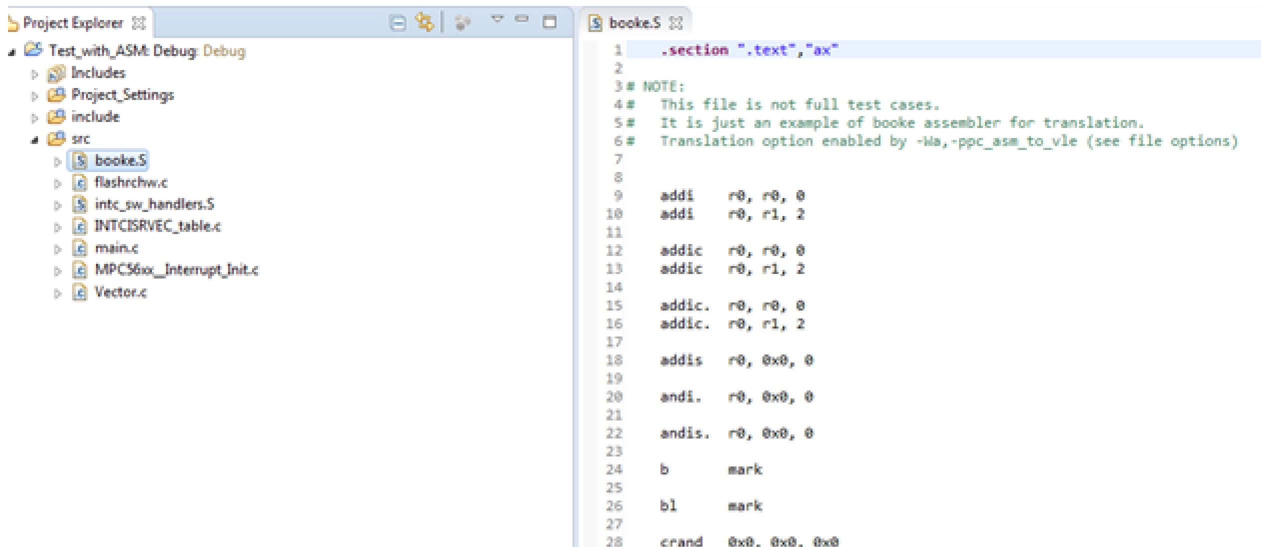

Figure 61. Project with BookE file

The file *booke.S* is a set of BookE instructions. Right-click on *booke.S* and select *Properties*. A dialog-box will appear showing file-specific properties, displayed below.

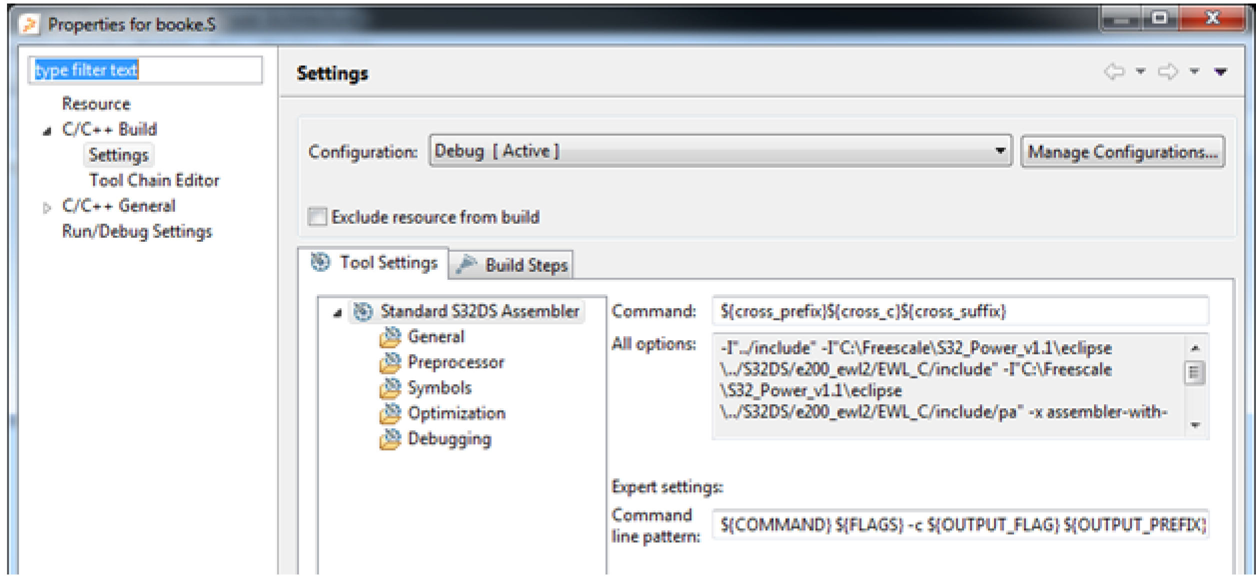

Figure 62. Project with BookE file

Go to *C/C++ Build>Settings>Tool Settings>Standard S32DS Assembler>Command line pattern*. A new assembly file will have the command line pattern:

```
"${COMMAND} ${FLAGS} -c ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}"
```

This *does not* support translation. To enable BookE-to-VLE assembly language translation, replace the command line pattern with:

```
"${COMMAND} ${FLAGS} -c -Wa,-ppc_asm_to_vle ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT} $
{INPUTS}"
```

The extra "-Wa,-pcc_asm_to_vle" command activates the assembly translation feature. The new command line pattern is shown in the figure below.

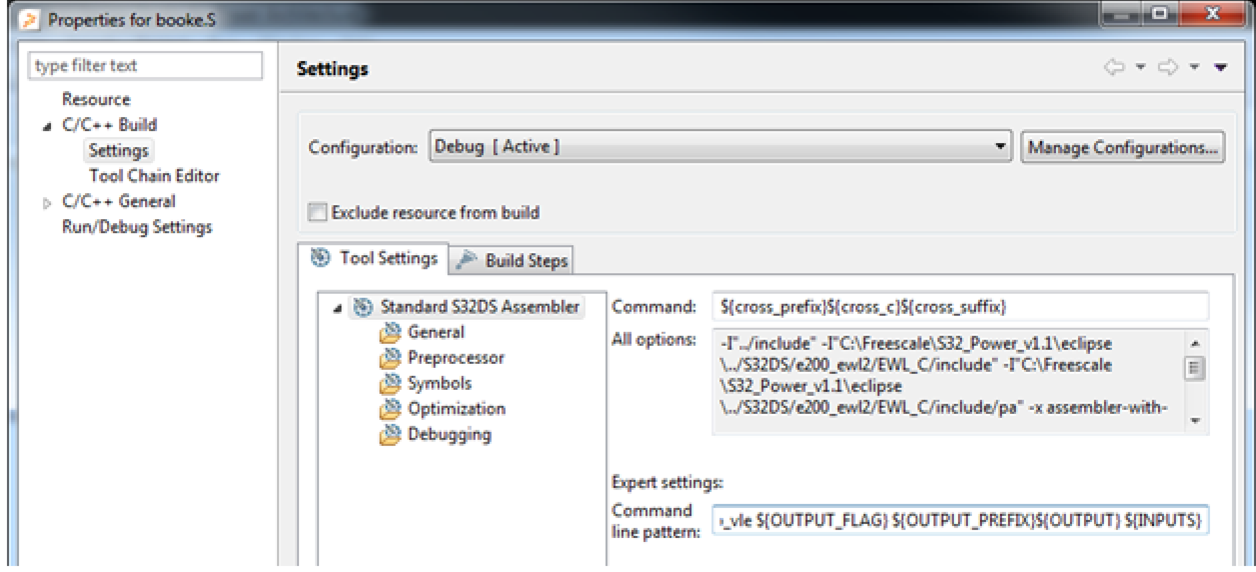

Figure 63. File properties with translation enabled

Hit *OK*. Now BookE-to-VLE translation is now be enabled for *booke.S*. Building the project *Test_with_ASM* will no longer generate compile errors even with the BookE instructions of *booke.S*. Perform these steps for your BookE assembly files to migrate your CodeWarrior assembly code to S32DS.

7 Debug Configurations

Code is only useful if you can run it. You would therefore likely want to carry over debug configurations from CodeWarrior to S32DS as well. S32DS cannot import debug configurations from CodeWarrior. When you create a new project in S32DS, debug configurations are automatically generated based on the connection options you select in the New Project wizard. Make sure the connection type you choose in the S32DS project is the same as the one you have in your CodeWarrior project. Debug configurations are not automatically generated if the S32DS project is copied from another S32DS project. In that case you will have to create your own debug configurations. For steps on how to do that, visit the KEA software integration guide that is part of the KEA Quick Start Package. To obtain the package visit FRDM-KEAZ128. But because the KEA is a single-core device, the KEA software integration guide only covers single-core project debug configurations. There are a few more considerations for multi-core project debug configurations.

7.1 Multi-Core Debug Configurations

Multi-core debug is different in S32DS than in CodeWarrior. S32DS produces a separate project, and therefore a separate debug configuration, for each core of a device. The MPC5644C is dual core, so *Migrated Project Eclipse Multicore* is split into two debug configurations, shown below.

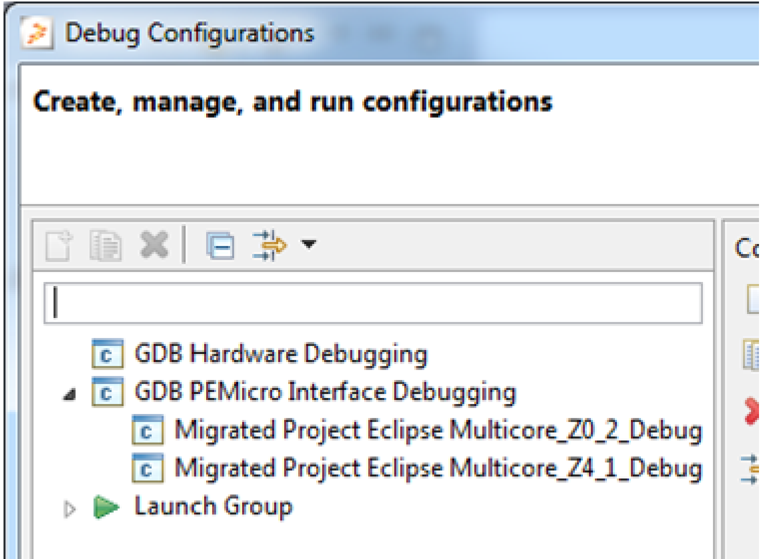

Figure 64. Dual core device, two projects, two debug configurations

Multi-core debug configurations linked to each other by the configuration of the main core, in this case *Migrated Project Eclipse Multicore_Z4_1_Debug*. Recall that the main core is the one whose project *main.c* contains *hw_init()*. The main core debug configuration specifies the locations of the secondary core ELF files to be launched along with said main core debug configuration. The following figure shows the settings of *Migrated Project Eclipse Multicore_Z4_1*. It specifies one additional ELF file to program and points to the ELF file of the second core project, *Migrated Project Eclipse Multicore_Z0_2.elf*.

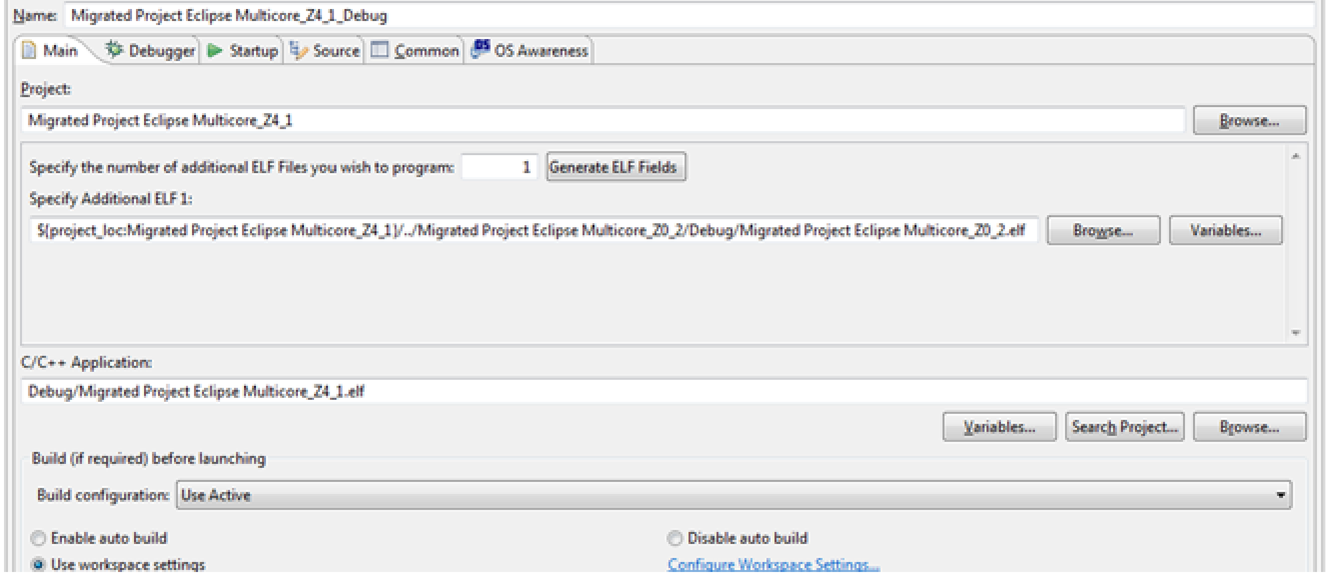

Figure 65. Main core debug config specifying additional ELF file for secondary core

Secondary core debug configurations specify no additional ELF files, shown below.

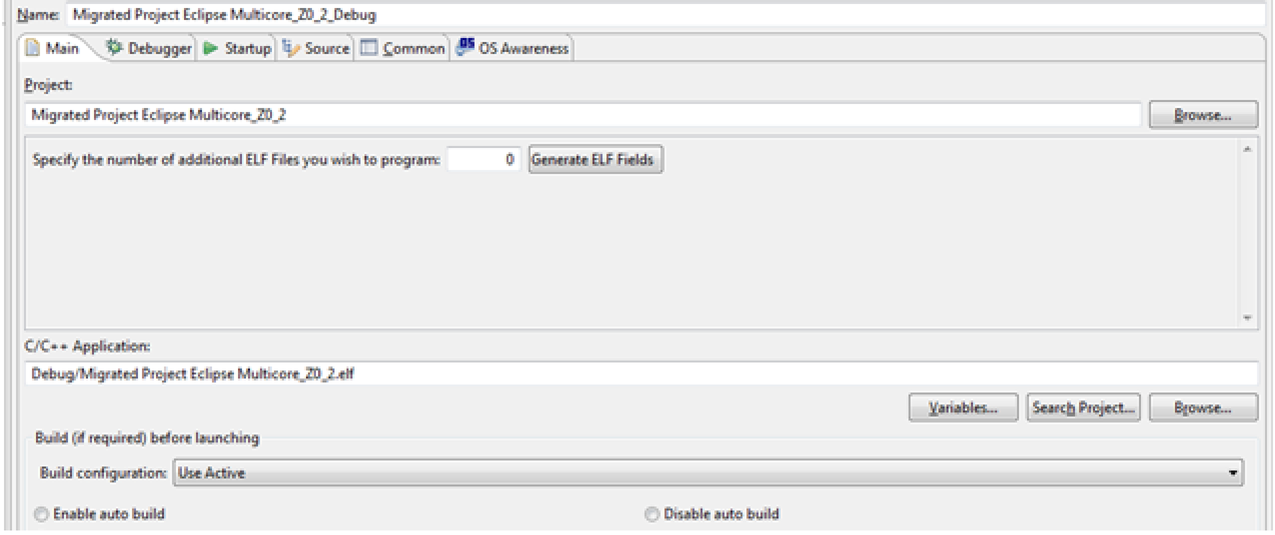

Figure 66. Main core debug config specifying additional ELF file for secondary core

You can launch both projects at once by selecting a launch group instead of a specific debug configuration. The launch group is also located under both the debug menu. In the figure below, the launch group in question is called *Migrated Project Eclipse Multicore_LaunchGroup*.

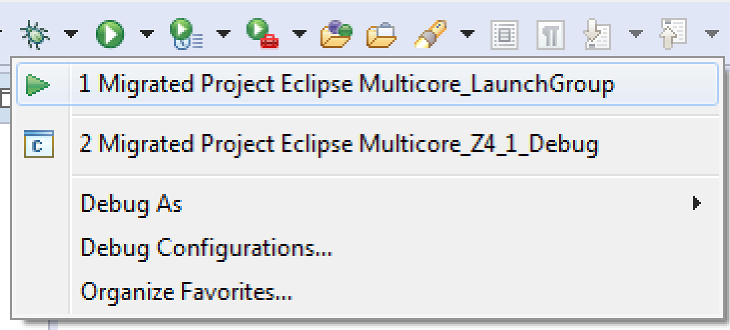

Figure 67. Multicore launch group

Each core possesses its own debug thread. Switch between the threads by clicking on the project you want within the *Debug* perspective, as displayed in the figure below. The debug controls (e.g. run, pause, step through) are associated with whichever thread is currently selected.

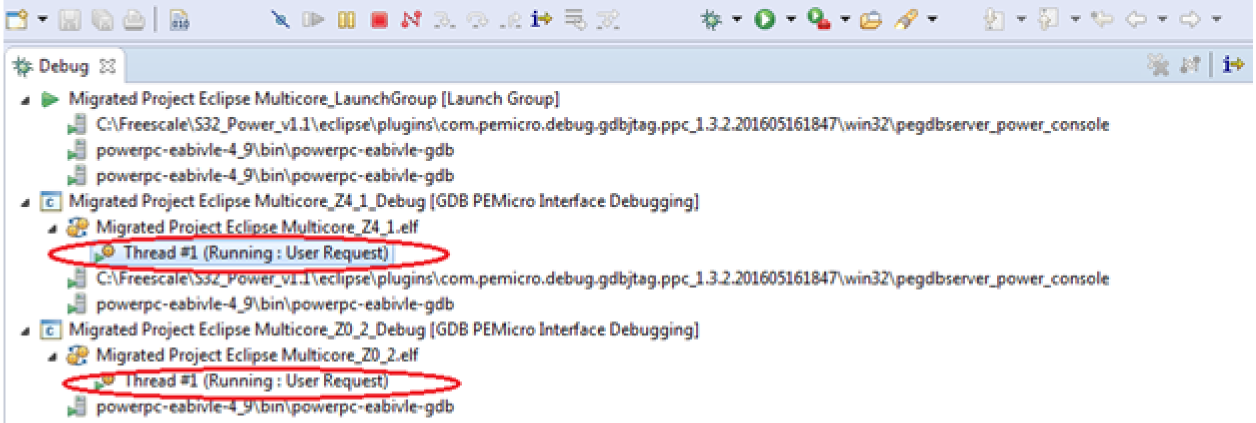

Figure 68. Multicore launch group

You can terminate a debug session one thread at a time, or stop them all at once by selecting the launch group. To do so, click on the launch group in the Debug perspective, in this case *Migrated Project Eclipse Multicore_LaunchGroup [Launch Group]* and hit the stop icon, shown below. This will terminate all threads at once.

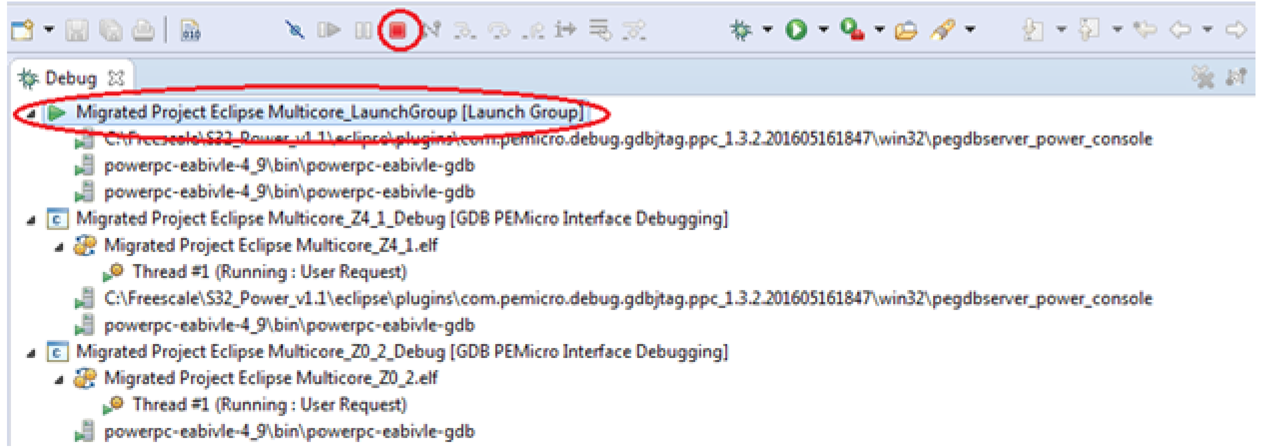

Figure 69. Multicore launch group

8 Building the Converted Project

To build the converted project:

1. Select the project in the Project Explorer.
2. Select Project > Build Project or click on the hammer icon ( ).

9 Conclusion

This application note has covered the steps that you need to take in order to successfully migrate your CodeWarrior project to S32DS. Future automotive/industrial devices from NXP will have tools enablement through S32DS. CodeWarrior will continue to exist to support legacy devices. Once the tool transition is complete, we encourage customers to develop new projects on S32DS. We plan to continually release updates to enhance and expand your S32DS experience. For more information on S32DS, visit S32DS.

10 Revision History

The following table summarizes the changes done to this document since the initial release.

Table 8. Revision history

Revision	Date	Substantive changes
0	04/2016	Initial release.
1	06/2016	Adjusted the project descriptions to accommodate the changes made in the S32 Design Studio for Power Architecture v1.2.
2	08/2016	Added topic "Assembly Translation" and updated topic "Project properties".
3	09/2016	Added topic "Multi-Core Debug Configurations" and updated topic "Headers and Sources, Multi-C" and "Debug Configurations".

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer’s technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: http://www.nxp.com/SalesTermsandConditions.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, Freescale, the Freescale logo, and CodeWarrior are trademarks of NXP B.V. ARM, the ARM powered logo, and Cortex are the trademarks of ARM Limited (orits subsidiaries) in the EU and/or elsewhere. All other product or service names are the property of their respective owners. All rights reserved.

The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© 2016 NXP B.V.

Document Number AN5282
Revision 3, 09/2016