S32 Design Studio for ARM, Version 2018.R1

Reference Manual

Contents

Chapter 1

Introduction

Topics:

This document explains how to use S32 Design Studio for ARM, Version 2018.R1 (hereinafter also referred to as S32DS ARM v2018.R1) to help you create applications and configure the S32DS ARM v2018.R1 IDE. This chapter presents an overview of the manual and introduces you the structure of the document.

In this chapter:

Quick links	Lists web-links to S32 Design Studio pages
S32 Design Studio for ARM, Version 2018.R1 content	Describes S32DS ARM v2018.R1 contents
Terminology	Lists terms used in the manual
About this manual	Describes the contents of this manual
Accompanying documentation	Describes supplementary S32DS ARM v2018.R1 documentation, third-party documentation, and references
Installing S32 Design Studio for ARM, Version 2018.R1	Describes the installation of S32DS ARM v2018.R1 software
Uninstalling S32 Design Studio for ARM, Version 2018.R1	Describes the uninstallation of S32DS ARM v2018.R1 software

Quick links

• S32 Design Studio page (overview, downloads)	www.nxp.com/S32DS
• S32 Design Studio community (for publicly shared cases)	community.nxp.com/community/s32/s32ds
• Technical support (for confidential issues)	www.nxp.com/support Hardware and Software link

S32DS ARM v2018.R1 content

The S32DS ARM v2018.R1 product package supports following features and technologies:

- Eclipse Neon 4.6.3 Framework (CDT version 9.2.1)
- GNU tools for ARM® Embedded Processors (launchpad) build tools 4.9 2015q3 (4.9.3 20150921)
- ARM64: Linaro GCC 4.9-2015.05
- GNU Tools for ARM® Embedded Processors build (6.3.1 20170824)
- Libraries: newlib, newlib-nano, and ewl2 (ewl and ewl-nano)
- P&E Multilink/Cyclone/OpenSDA (with PnE GDB Server)
- SEGGER J-Link (with SEGGER GDB Server)
- Integrated S32 SDK for S32K14x EAR release v.0.8.6
- S32DS Application Project and S32DS Library Project wizards helping you to create new applications and libraries for supported devices
- SDK management with support for:
  - Drivers for KEA family (Evaluation grade)
  - FreeMaster Serial Communication driver for KEA and S32K1xx families
  - Automotive Math and Motor Control Libraries for KEA and S32K devices
- Import of projects created in CodeWarrior for MCU v.10.6 and Kinetis Design Studio
- Built-in usage examples for the MQX RTOS on the MAC57D54H MCU
- IAR v7.x and v8.x compiler support by S32DS Application Project and S32DS Library Project wizards
- GHS v2017.1.4 compiler support by S32DS Application Project and S32DS Library Project wizards
- iSystem, Lauterbach, and IAR debuggers support by S32DS Application Project and S32DS Library Project wizards
- Kernel-Aware debugging for FreeRTOS, OSEK

Following tools are supported but are not included in the package and should be installed separately:

- IAR, GHS compilers
- iSystem, Lauterbach and IAR debuggers

Supported devices:

- SKEAZN8, SKEAZN16, SKEAZN32, SKEAZN64, SKEAZ128, SKEAZ64
- S32K142, S32K144, S32K146, S32K148
- S32V234
- MAC57D54H

About this manual

Each chapter of this manual describes a different area of software development. The following table lists the contents of this manual.

Table 1: Manual Contents

Chapter	Description
Introduction	This chapter presents an overview of the manual and introduces you to the information layout of the manual
Working with projects	Explains how to use the S32DS ARM v2018.R1 tools to create and work with projects
Build properties for S32DS Projects	Explains build properties for a S32DS ARM v2018.R1 project
Working with debugger	Explains how to use the S32DS ARM v2018.R1 development tools to debug a program executing on the microcontroller
Multicore debugging	Explains how to define multiple, arbitrary groupings of cores and perform multicore operations
Connections	Describes the features and settings of the connections that interface the debugger with bareboard target and allows it to debug program code on the target
Working with SDKs	Describes usage of the software development kits (SDKs) in the S32DS ARM v2018.R1

Note: The text of and illustrations in this document are primarily targeted for Microsoft Windows platform. With respect to platform implementation differences and terminology common on the specific platform, described procedures can be applied to other operating systems supported by S32DS ARM v2018.R1. This includes the **Start** button, which is available on Windows platform only.

Terminology

The following are some of the terms used in the document:

Table 2: Terminology

Term	Description
ARM	Acorn RISC Machine architecture - family of instruction set architectures for computer processors developed by British company ARM Holdings, based on a reduced instruction set computing (RISC) architecture
CDT	C/C++ Development Tooling - a fully functional C and C++ IDE based on the Eclipse platform
DWARF	Debugging With Attributed Record Formats - a widely used, standardized debugging data format
ELF	Executable and Linkable Format - a common standard file format for executables, object code, shared libraries, and core dumps
EmbSys Registers	EMBedded SYStems REGister View - an Eclipse Plugin designed for monitoring and modifying memory values of embedded devices

Term	Description
FPU	Floating-point unit - math coprocessor; a part of a computer system specially designed to carry out operations on floating point numbers
IDE	Integrated development environment
GCC	GNU Compiler Collection - a compiler system produced by the GNU Project supporting various programming languages
GDB	GNU Debugger
KDS	Kinetis Design Studio - a complimentary integrated development environment for Kinetis MCUs that enables robust editing, compiling and debugging of your designs
P&E	P&E Microcomputer Systems
RAM	Random-access memory
S32DS ARM v2018.R1	S32 Design Studio for ARM, Version 2018.R1
XML	Extensible Markup Language - a markup language that defines a set of rules for encoding documents in a format which is both human-readable and machine-readable

Accompanying documentation

S32DS ARM v2018.R1 includes an extensive documentation library of user guides, reference manuals etc. Take advantage of this library to learn how to efficiently develop software using the S32DS ARM v2018.R1 programming environment.

S32DS ARM v2018.R1 documentation presents information in the following formats:

- **PDF** Portable Document Format of the manuals, such as the Common Features Guide, Reference Manual or Release Notes
- **HTML** Hypertext Markup Language version of the manuals

PDF-documentation

The **S32 Design Studio Documentation Suite** includes the S32DS ARM v2018.R1 PDF-documentation.

You can access the manuals, FAQ, etc. by:

- opening the **start_here.html** in **<S32 Design Studio install dir>/S32DS/help** directory, where **S32 Design Studio Install dir** is the directory that S32 Design Studio was installed into
- selecting **Help** > **Documentation** from the S32DS ARM v2018.R1 menu bar
- selecting the **Documentation** shortcut.

Online help

To view the online help:

1. Select **Help** > **Help Contents** from the S32DS ARM v2018.R1 menu bar.
2. Select required manual from the **Contents** list.

Release notes

Before using the S32DS ARM v2018.R1, read the developer notes. These notes contain important information about last-minute changes, late-breaking information about new features, bug fixes, incompatible elements, known problems, or other topics that may not be included in this manual.

Read the Release notes in this directory:

<S32 Design Studio install dir>/S32DS/Release_Notes.

The release notes for specific components of the S32DS ARM v2018.R1 are located in the **Release_Notes** directory in the S32DS ARM v2018.R1 installation directory.

Installing S32DS ARM v2018.R1

- Windows platform:

  Double-click the package to start the installation of S32DS ARM v2018.R1. The user account designated for installing S32DS ARM v2018.R1 must be a member of local Administrators security group. If User Account Control (UAC) is enabled, Windows will ask you to elevate the privileges when you run the installation package. When asked by UAC, grant the S32DS ARM v2018.R1 installer permissions to make changes on your computer.

  Note: Installing S32DS ARM v2018.R1 in the console or silent installation modes is not supported.
- Linux platform:

  1. Open terminal window.
  2. Navigate to the directory with the downloaded BIN file:

     ```
     cd ~/S32DS
     ```
  3. Add the execute permissions to the binary:

     ```
     chmod a+x ./<install_name>.bin
     ```
  4. Run the installer:

     ```
     ./<install_name>.bin
     ```
  5. Follow the on-screen instructions.

Refer to **Installation manual** > **How to install S32DS ARM v2018.R1** in **Getting Started** for details.

The updates for S32DS ARM v2018.R1 are published on NXP website and can be installed using **Help** > **Check for Updates** and **Help** > **Install New Software**. Refer to Installing updates, patches and service packs for details.

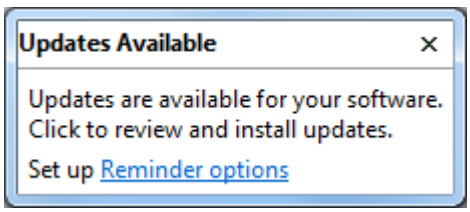
Installing updates, patches and service packs

New functionality including support for new devices can be added to S32DS ARM v2018.R1 with service packs, updates, and patches. Service packs add specific support for new devices. Updates and patches correct software defects and add general functionality affecting more than one device family.

There are several ways to obtain updates:

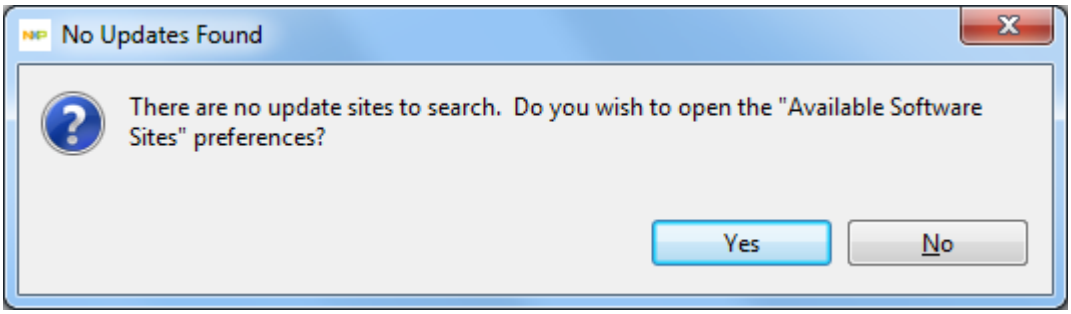
- IDE automatically looks for updates and notify you each time S32DS ARM v2018.R1 starts. Schedule and options can be configured in **Window** > **Preferences** > **Install/Update** > **Automatic Updates**.

  Notification appears in the bottom right corner:

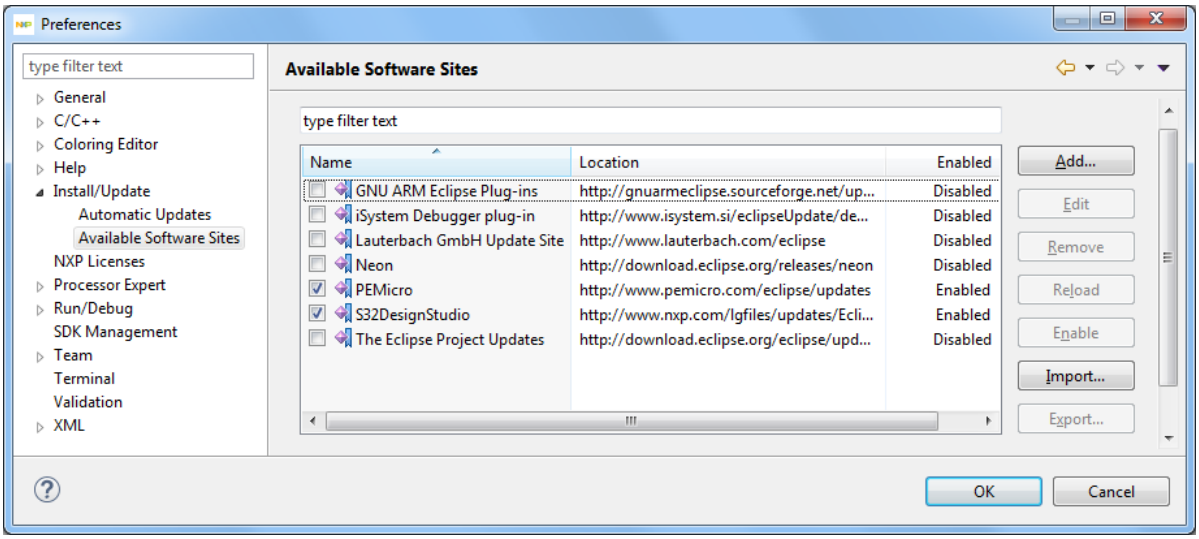
  

  1. Click on the notification, **Available Updates** window appears.
  2. Check the updates that you wish to install and click **Next**. **Update Details** page appears.
  3. Review and confirm the updates, click **Next**. Page with License text appears.
  4. Review Liceses and choose **I accept the terms of the license agreement**. Click **Finish**. Installation starts.

- If Automatic Updates option is disabled or it can not find required updates, follow these instructions:
  1. Choose **Help** > **Check for Updates**.
  2. IDE refers to the list of sites that are used when browsing available software and checking for updates. If your list of sites is empty, the following notification appears:

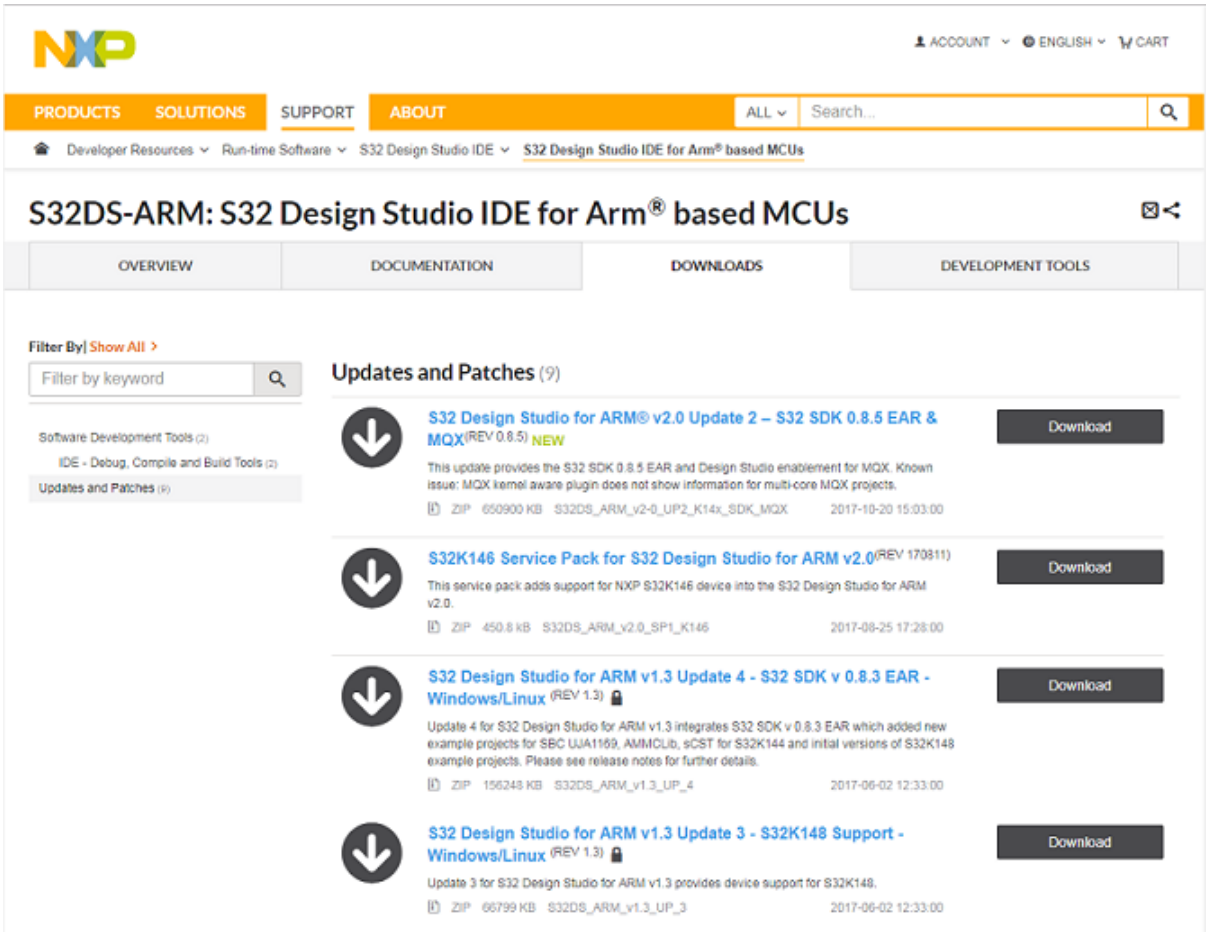


  3. Click **Yes**. **Available Software Sites** page of Preferences appears.
  4. Choose required update sites and click **OK**:

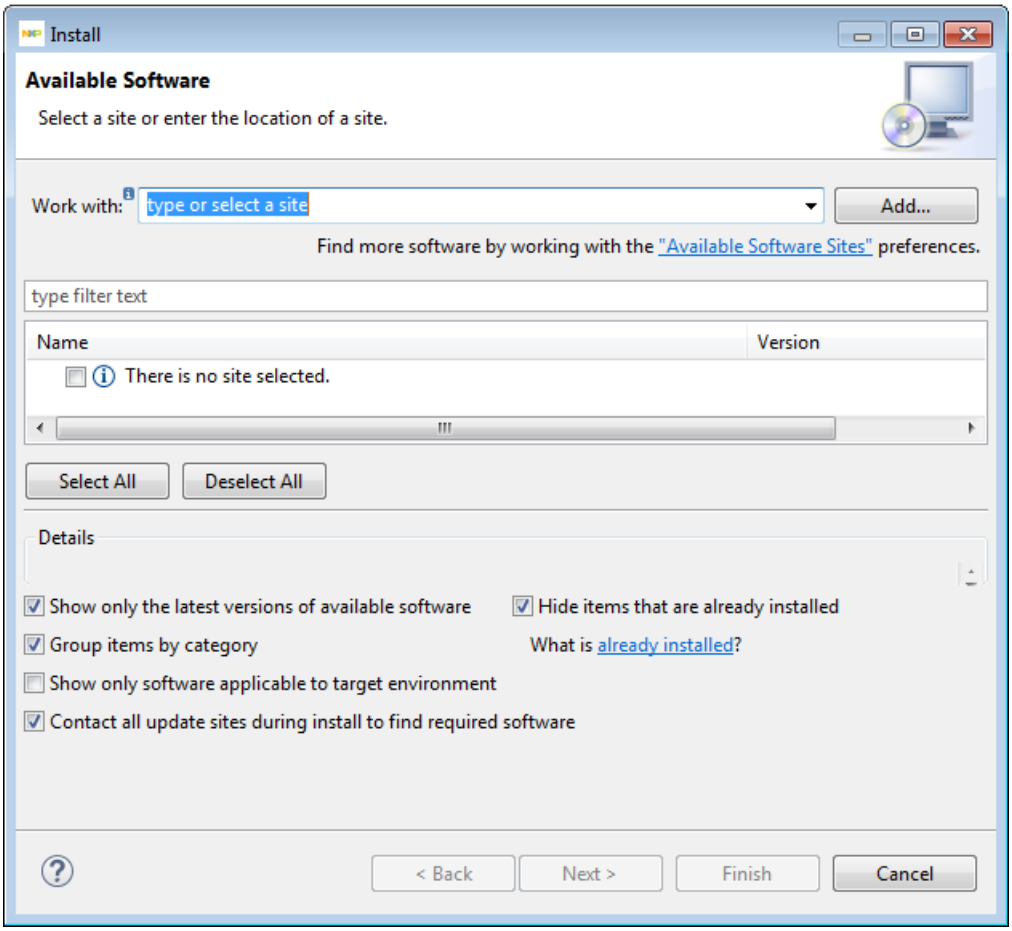


  5. Go back to **Help** > **Check for Updates**. **Available Updates** window appears and updates can be installed as described above.

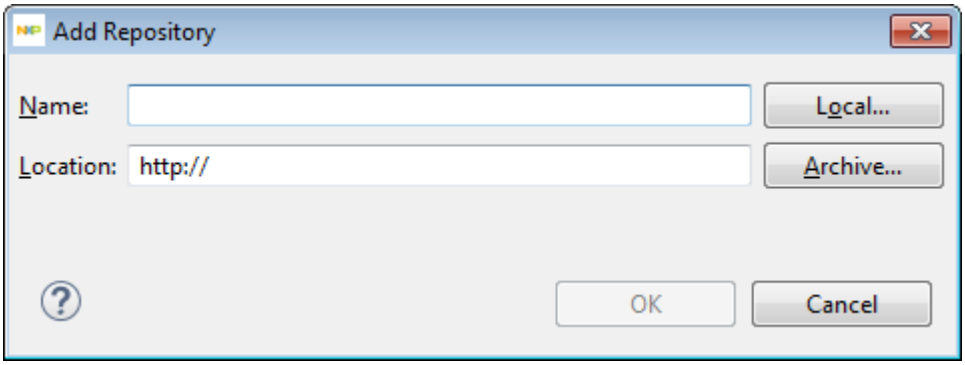
- Updates and patches for S32DS ARM v2018.R1 are also published on NXP website. If your computer does not have internet access, you can use another computer to download the archive that contains the service pack, update or patch from the NXP website and install offline.
  1. Go to nxp.com/S32DS and choose IDE option, then click **Downloads** tab and filter by **Updates and Patches**:



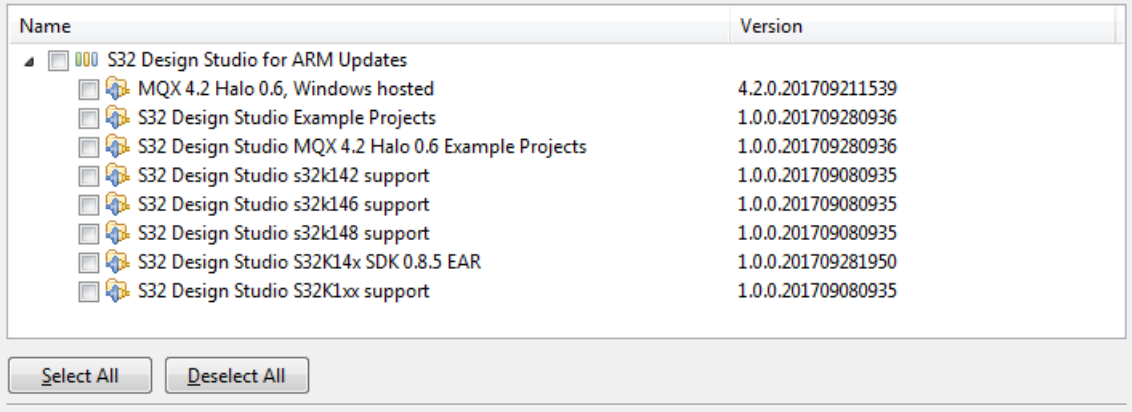
2. Choose required item and click **Download**. Downloading of .zip file starts.
3. Choose **Help** > **Install New Software...** from IDE menu bar. **Install** wizard appears.



4. Click **Add...**. **Add Repository** window appears



5. Click **Archive...** and navigate to directory with downloaded .zip file. Choose it and click **Open**, then **OK**.
6. List of features available in update appears, choose several options or click **Select All**, then click **Next>**. **Update Details** page appears.



7. Review and confirm the updates by clicking **Next>**. **Review Licenses** page appears.
8. Read license text and choose **I accept the terms of the license agreement**.
9. Click **Finish**. Updating software starts.
10. When notification about certificate appears, confirm that you trust the vendor.
11. Update installation continues, and the **Software Updates** message box appears notifying you of the required restart of S32 Design Studio. Click **Yes** to restart IDE.

After S32DS ARM v2018.R1 restarts, all new features are available.

Uninstalling S32DS ARM v2018.R1

To uninstall S32DS ARM v2018.R1 perform these steps:

1. Close the S32DS ARM v2018.R1 application.
2. Navigate to the uninstaller:
   - Windows platform: from the menu **Start** > **All Programs** > **S32 Design Studio for ARM Version 2018.R1** > **Uninstall**.

     Note: For Microsoft Windows 10 go to **Settings** > **System** > **Apps & features** > **S32 Design Studio for ARM Version 2018.R1** > **Uninstall**
   - Linux platform:

     1. Open terminal window.

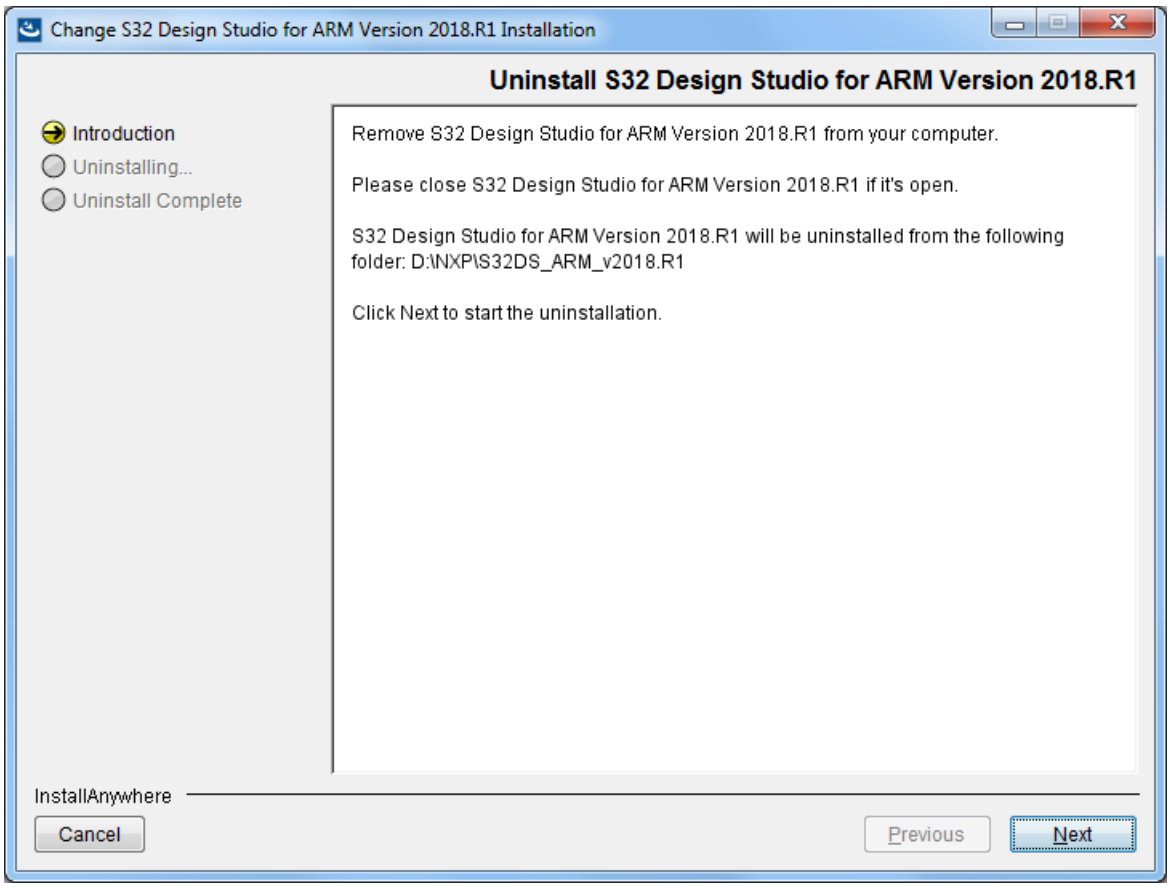
     2. Navigate to the Desktop directory with the S32DS ARM v2018.R1:

     ```
     cd ~/"Desktop/NXP S32 Design Studio/S32 Design Studio for ARM Version 2018.R1"
     ```

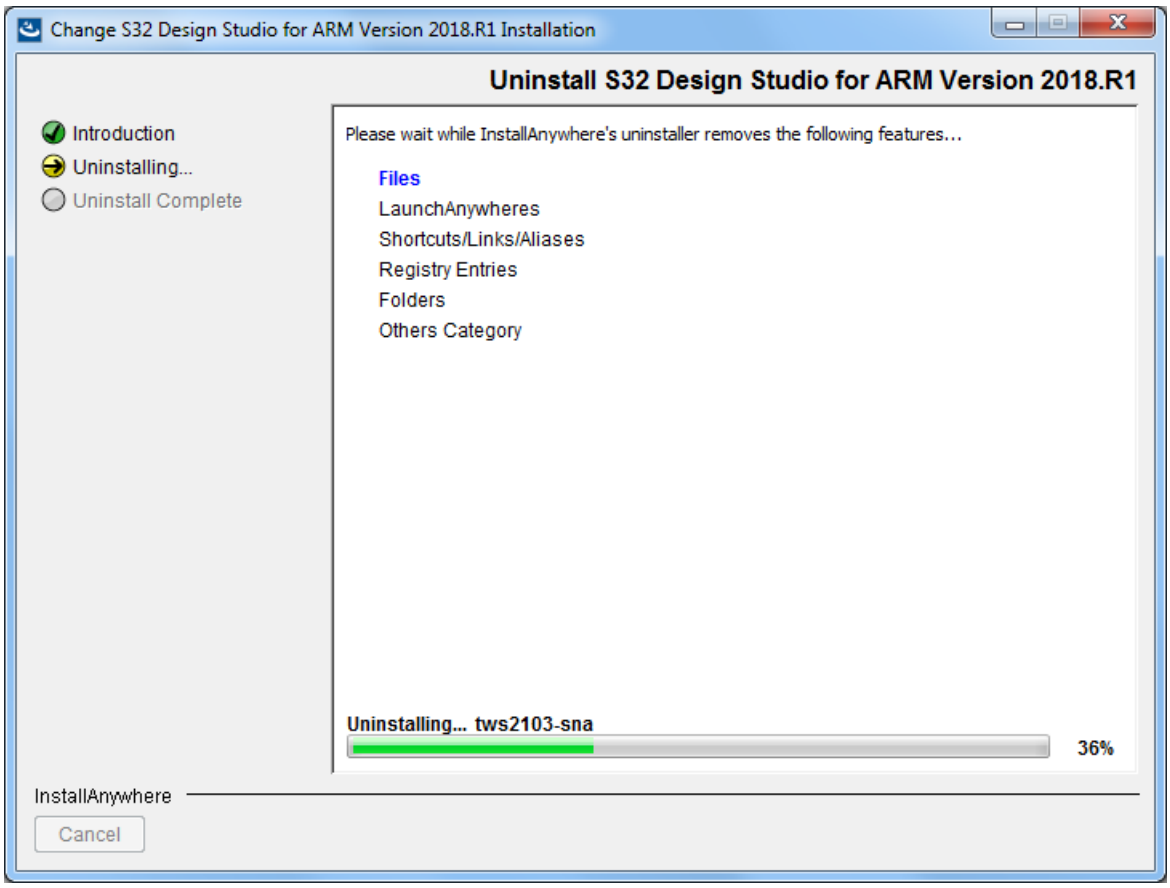
     3. Run the uninstaller:

     ```
     ./Uninstall
     ```

   The uninstall wizard appears.
3. Click **Next**:

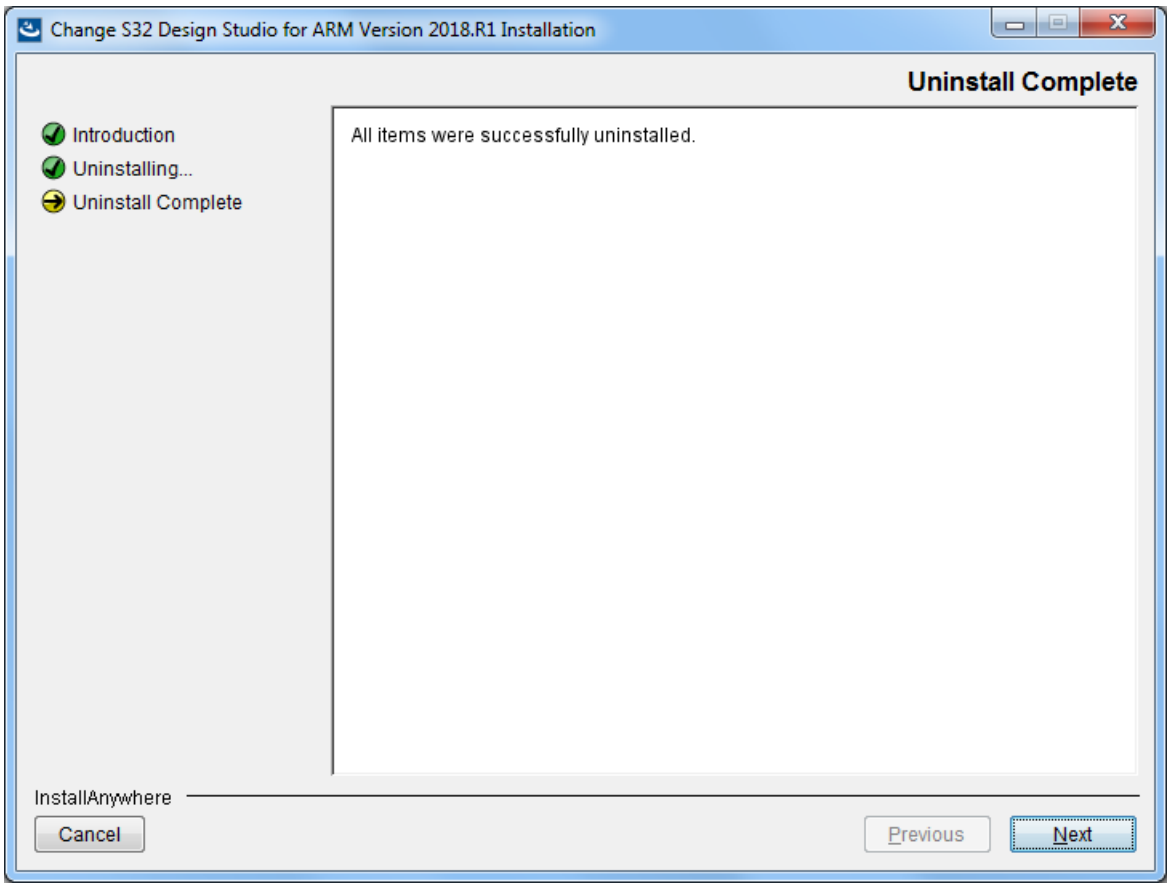


The S32DS ARM v2018.R1 uninstallation starts.



License removing window appears.

4. Wizard displays the **Uninstall complete** page. Click **Next** button to close the wizard:



If there are some issues then the wizard displays the corresponding messages on the **Uninstall complete** page.

Note: The uninstall wizard does not remove third party software installed during the installation of the S32DS ARM v2018.R1.

Chapter 2

Working with Projects

Topics:

This chapter explains how to use the S32DS ARM v2018.R1 to create and work with projects.

Types of projects

Projects organize files and various compiler, linker, and debugger settings associated with the applications or libraries you develop. You can use one of the S32DS Project wizards to create new projects that group these files and settings into build and launch configurations.

Following types of S32DS projects can be created with project creation wizards:

- Application Project

  This project can be created by using the S32DS Application Project wizard. The build artifact created when you build this type of a project is an ELF executable.
- Library Project

  This project can be created by using the S32DS Libarary Project wizard. The build artifact created when you build this type of a project is an A file of a static library.

This section describes how you can create the application and library projects in S32DS ARM v2018.R1. The type of project is based on wizard that you select in the **File** menu to create the project. In S32DS ARM v2018.R1 you can only create projects for the standalone applications (also known as bare metal) that run without the operating system on the target SoC. Creating ELF files for a specific Linux platform is not supported.

The wizards allow you to build applications and static libraries for the following processor families: KEA, S32K1xx, MAC57D5xx, and S32V.

You can use the S32DS Example Project wizard to import example projects into workspace, see section **Common Features Guide** > **IDE Extensions** > **Importing projects and files** > **Import example project**.

New project wizards

This topic describes the various pages that the **New S32DS Project** wizard displays as it assists you to create a project (**Application** project or **Library** project).

The **New S32DS Project** wizard creates a project with the necessary startup code and linker configuration to bring the generated executable out of reset and into main with the supported run control solution.

The pages of the wizard can differ based on the project type or processor.

New S32DS Application Project wizard

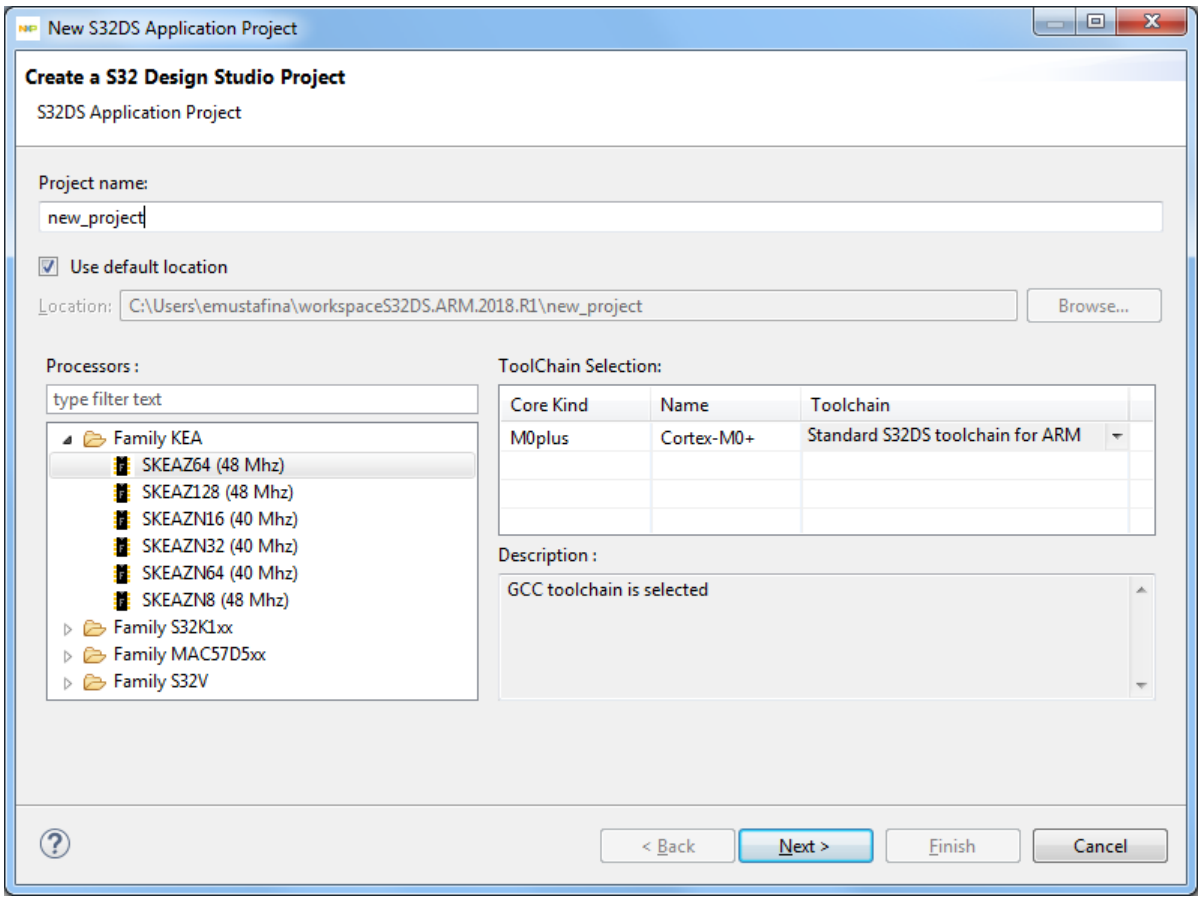
This topic describes pages that the **New S32DS Application Project** wizard displays as it assists you in creating a S32DS Application Project.

The pages of the **New S32DS Application Project** wizard:

1. Project name and processor (the **Create a S32 Design Studio Project** page)
2. Cores and parameters for them (the **New S32DS Project for < processor name>** page)

The pages of the **New S32DS Application Project** wizard can differ based on the processor.

Project name and type



Use this page to specify the project name, the directory where the project files are located and processor. The table below describes the various options available in the S32DS Application Project wizard.

Table 3: S32DS Application Project wizard – Page 1 settings

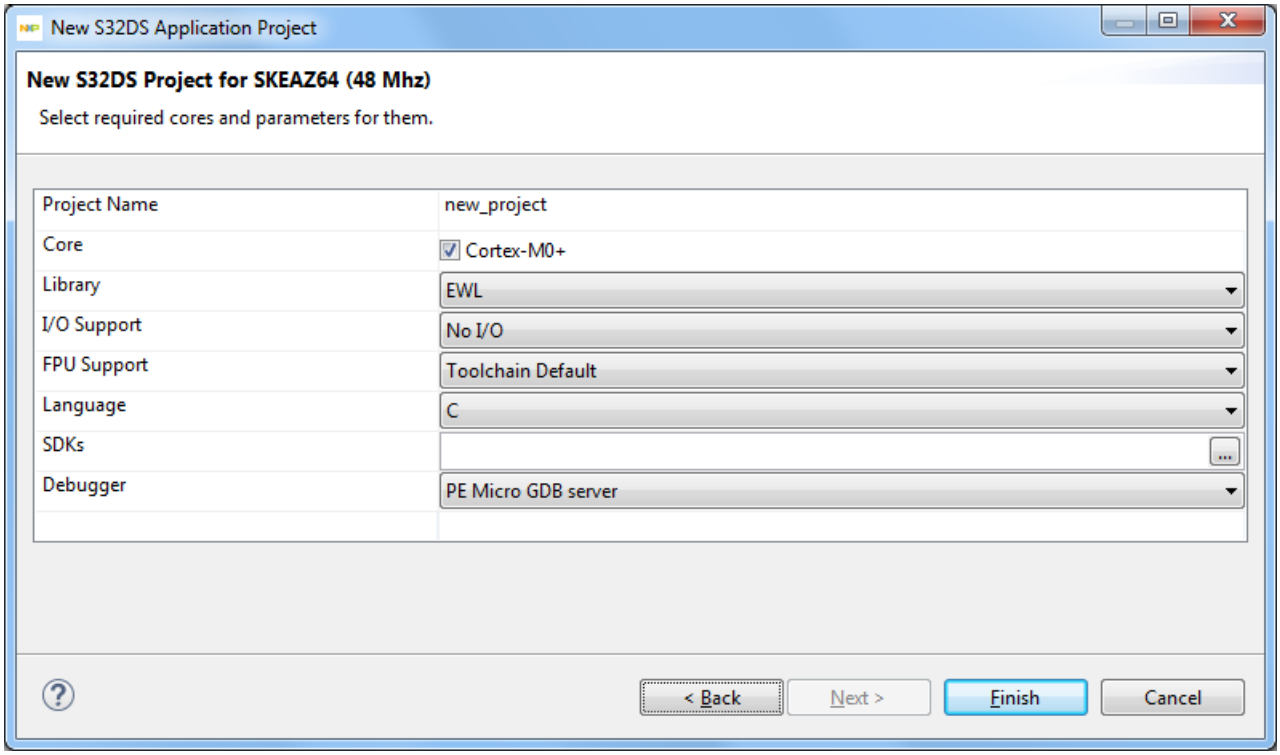
Option	Description
Project name	Enter the name for the new project in this text box. Do not use these reserved/special characters/symbols in the project name: < (less than) > (greater than) : (colon) / (forward slash) \ (backslash) | (vertical bar or pipe) ? (question mark) * (asterisk) " (double quote) @ (at) # (number sign) = (equals sign) ; (semicolon) " " (space) ' (single quote) ` (grave accent) {} (curly braces) () (parentheses) [] (square brackets) $ (dollar sign) Special characters/symbols in the project name cause the error message. Project name should be unique inside your workspace.

Option	Description
Use default location	Stores the files required to build the program in the Workbench's current workspace directory. The project files are stored in the default location. Clear the **Use default location** checkbox and click **Browse…** to select a new location.
Location	Specifies the folder that contains the project files. Click **Browse…** to navigate to the desired folder. This option is available only when **Use default location** checkbox is clear. Do not use the “Space” symbol in the folder name.
Processors tree control	Specifies the processor you would like to use in new project
ToolChain Selection table	Details on the selected processor: core kind, core name, and GCC toolchain that will be used to build the project. Depending on the selected processor, following options are available: • **Standard S32DS toolchain for ARM** Available for the following processors that you select in the **Processors** tree: • All processors except for S32V234 Cortex-A53. • **ARM Bare-Metal 32-bit Target Binary Toolchain** Available for the following processors that you select in the **Processors** tree: • All processors of S32K1xx family • S32V234 Cortex-M4 • **Standard S32DS toolchain for ARM Cortex-A** Available for the following processors that you select in the **Processors** tree: • Only S32V234 Cortex-A53. • **IAR Toolchain for ARM - (7.x)** **Note:** Requires an installed IAR Embedded Workbench for ARM 7.x and IAR plugin for ARM 7.x. If you do not see this option in the list, make sure to install IAR Embedded Workbench for ARM 7.x on the current computer first. After that, install IAR Embedded Workbench plugin manager in S32DS ARM v2018.R1 first, and then install IAR Embedded Workbench plugin for ARM 7.x by using the plugin manager. Lastly, verify that you have selected a supported processor. **Important:** If you have both IAR Embedded Workbench for ARM 7.x and IAR Embedded Workbench for ARM 8.x installed at the same time, this option is replaced by **IAR Toolchain for ARM - (legacy, 7.x)** once you install the IAR plugin for ARM 8.x by using IAR Embedded Workbench plugin manager. Available for the following processors that you select in the **Processors** tree: • All processors except for S32V234 Cortex-A53. • **IAR Toolchain for ARM - (legacy, 7.x)** **Note:** Requires an installed IAR Embedded Workbench for ARM 8.x and IAR plugin for ARM 8.x. If you do not see this option in the list, make sure to install IAR Embedded Workbench for ARM 8.x on the current computer first. After that, install IAR Embedded Workbench plugin manager in S32DS ARM v2018.R1 first, and then install IAR Embedded Workbench plugin for ARM 8.x by using the plugin manager. Lastly, verify that you have selected a supported processor. Available for the following processors that you select in the **Processors** tree: • All processors except for S32V234 Cortex-A53. • **IAR Toolchain for ARM - (8.x)**

Option	Description
	Note: Requires an installed IAR Embedded Workbench for ARM 8.x and IAR plugin for ARM 8.x. If you do not see this option in the list, make sure to install IAR Embedded Workbench for ARM 8.x on the current computer first. After that, install IAR Embedded Workbench plugin manager in S32DS ARM v2018.R1 first, and then install IAR Embedded Workbench plugin for ARM 8.x by using the plugin manager. Lastly, verify that you have selected a supported processor. Available for the following processors that you select in the **Processors** tree: • All processors of S32K1xx family • S32V234 Cortex-M4 • **GHS ARM Standalone Executable Toolchain** **Note:** Requires an installed GHS plugin v2017.1.4. If you do not see this option in the list, make sure to install GHS Toolchain v2017.1.4 on the current computer first. Then verify that you have selected a supported processor. The only available debug option for project with GHS toolchain is Lauterbach TRACE32 debugger (and only PE Micro GDB server for MAC57D54H). Available for the following processors that you select in the **Processors** tree: • All processors except for S32V234 Cortex-A53.
Description field	Displays the project description.

After you select a processor, next page of the wizard appears. Its options depend on the selected processor.

Cores and parameters



A few projects can be presented on this step if a multicore processor has been selected on the first step and only one project is presented on the this step if a single-core processor has been selected on the first step.

Use this page to select processors cores and parameters for them to build the application project: the programming language etc.

Table 4: New S32DS Application Project wizard – Page 2 settings

Option	Description
Project Name	The application project name for the core displayed in the **Project Explorer** pane. Read only.
Core	Select cores for the project. The set of cores depends on the selected processor. By default, all core checkboxes are selected. **Note:** In case of MAC57D54H please make sure to select only those cores that you want to use in your application, because the wizard creates a separate project for each of the selected cores. If you delete projects for Cortex-A5 or Cortex-M0+ cores later, make sure to remove the dependencies for these projects from the project for the Boot Cortex-M4F core. Refer to Closing and deleting projects for details.
Library (except S32V234 Cortex-A53)	Use to specify support of the library linked to project: • **EWL** - c99 compliant Embedded Warrior Library • **EWL Nano** - a lightweight version of Embedded Warrior Library • **NewLib** - standard C/C++ library that is included with our ARM toolchain • **NewLib Nano** - a lightweight version of the standard C/C++ library. Default: **EWL**.
I/O Support (except S32V234 Cortex-A53)	Use to specify I/O modes used for project: • **Debugger Console** - configures how the GCC-ARM library deals with the console (e.g. printf() or puts()). The library uses a virtual connection with the debugger (also known as “semihosting”) • **No I/O** - no console support Default: **No I/O**.
FPU Support (except S32V234 Cortex-A53)	Select to include floating point support in the project: • **Toolchain Default** – enables or disables support for FPU unit based on settings in the toolchain that you have selected on the previous page of the wizard. • **Software: No FPU (-mfloat-abi=soft)** - select to include software floating point support in the project • **Hardware: -mfloat-abi=hard** - select to support hardware floating point. Using this option GCC build tools generates the code using hardware floating point instructions and uses FPU-specific calling convention: - mfloat-abi=hard • **Hardware: -mfloat-abi=softfp** - select to support hardware floating point. This option allows GCC build tools to generate code using hardware floating point instructions, but still uses the soft-float calling conventions • **Single Precision HW: -mfloat-abi=softfp -fshort-double** - select to support hardware floating point. Using this option GCC build tools generates code using hardware floating-point instructions, but still uses the soft-float calling conventions. Also, use same size for double as for float. Default: **Toolchain Default**.  **Warning:** The **-fshort-double** switch causes GCC to generate code that is not binary compatible with code generated without that switch. Use it to conform to a non-default application binary interface.
RAM Start Address	See the RAM Start Address values for each core. Read only. The values depend on the selected RAM size.

Option	Description
(only S32V234 Cortex-A53)	Default: for the Boot Cortex-A53_1 core - **0x3E900000**
RAM Size, KB (only S32V234 Cortex-A53)	Use to specify the size of RAM-memory: from 0 to 1024 with step 32. Default: **256**
Unused RAM, KB (only S32V234 Cortex-A53)	See the unused RAM value for the core. Read only. The values depend on the selected RAM Size. By default, this field is empty.
Language	The option you select sets up default compiler/linker options for the toolchain. Select the programming language that you want to use for writing the program's source code. You can select only one language: • **C** - sets up your application with ANSI C-compliant startup code, and initializes global variables • **C++** - sets up your application with ANSI C++ startup code, and performs global class object initialization. Default: **C**
SDKs	Use to select SDK. For more information, see Working with SDKs chapter. By default, this field is empty.
Debugger	Select a connection type to use for the project: • **PE Micro GDB server** • **Segger J-Link GDB server** Following processors and families are supported: • Family KEA • Family S32K1xx • **iSYSTEM winIDEA Debugger** Following processors and families are supported: • SKEAZ128 • SKEAZN32 • SKEAZN64 • SKEAZ64 • S32K144 • S32K148 **Note:** If you do not see this option in the list, make sure to install iSystem Debug plugin S32DS ARM v2018.R1 first, and then verify that you have selected a supported processor on the previous page. • **Lauterbach TRACE32 Debugger** **Note:** If you do not see this option in the list, make sure to install Lauterbach TRACE32 studio on the current computer first. After that, install Lauterbach TRACE32 plugin in S32DS ARM v2018.R1 first, and then verify that you have selected a supported processor on the previous page. Following processor families are supported: • Family KEA

Option	Description
	• Family S32K1xx • Family S32V • **IAR plugin Debugger** **Note:** If you do not see this option in the list, make sure to install IAR Embedded Workbench for ARM 7.x or IAR Embedded Workbench for ARM 8.x on the current computer first. After that, install IAR Embedded Workbench plugin manager in S32DS ARM v2018.R1 first, and then install IAR Embedded Workbench plugin by using the plugin manager. Lastly, verify that you have selected a supported processor on the previous page. Availability of the option to debug your code with IAR plugin Debugger depends the following: • Processor selected in the **Processor**s tree. • Installed version of IAR Embedded Workbench plugin • Toolchain selected in the **ToolChain Selection** list Only specific processors are supported for debugging your code with IAR plugin Debugger. For example, S32DS ARM v2018.R1 supports building projects with IAR Toolchain for ARM - (7.x) for processors of S32V family based on the Cortex-M4 core. However, debugging is not supported for these processors, and the **IAR plugin Debugger** option is not available if **S32V234 Cortex-M4** is selected in the **Processors** tree on the previous page. For *IAR Embedded Workbench for ARM 7.x and IAR Toolchain for ARM - (7.x)*, following processors are supported: • Family S32K1xx Only S32K144 processor. **Important:** S32DS ARM v2018.R1 supports IAR Toolchain for ARM - (7.x) for all processors of the S32K1xx family. However, debugging is only supported for S32K144. • Family KEA All family processors supported in S32DS ARM v2018.R1. For *IAR Embedded Workbench for ARM 8.x and IAR Toolchain for ARM - (legacy, 7.x)*, following processors are supported: **Note:** Debugging with IAR plugin Debugger is not supported. For *IAR Embedded Workbench for ARM 8.x and IAR Toolchain for ARM - (8.x)*, following processors are supported: • Family S32K1xx Only S32K144, S32K142, and S32K148 processors. Default: **PE Micro GDB Server**

New S32DS Library Project wizard

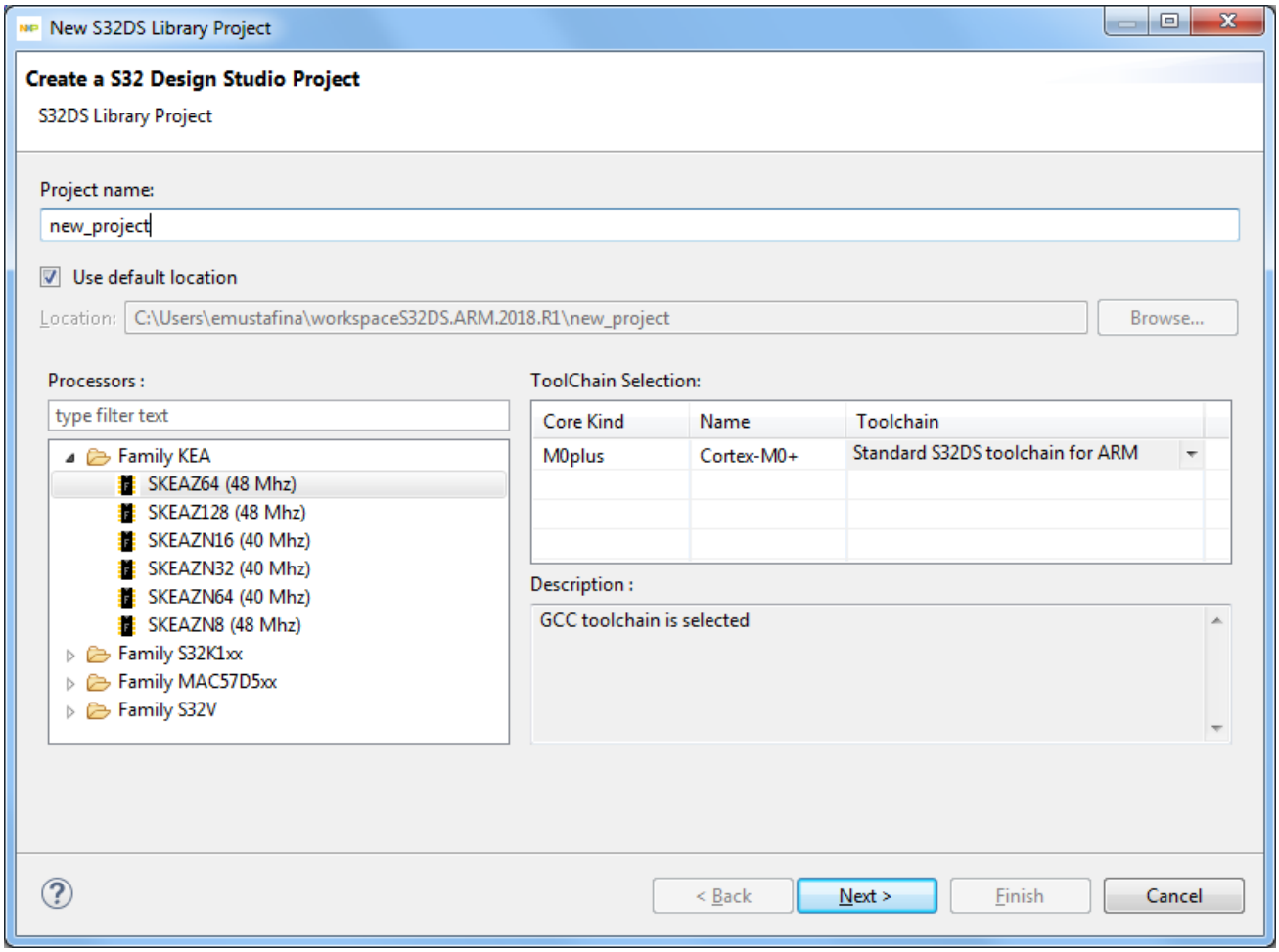
This topic describes pages that the **New S32DS Library Project** wizard displays as it assists you in creating a S32DS Library Project.

The pages of the **New S32DS Library Project** wizard:

1. Project name and type (the **Create a S32 Design Studio Project** page)
2. Cores and parameters (the **New S32DS Project for <processor name>** page)

The pages of the **New S32DS Library Project** wizard can differ based on the processor.

Project name and type



Use this page to specify the project name, the directory where the project files are located and processor. The table below describes the various options available in the S32DS Application Project wizard.

Table 5: S32DS Library Project wizard – Page 1 settings

Option	Description
Project name	Enter the name for the new project in this text box. Do not use these reserved/special characters/symbols in the project name: < (less than) > (greater than) : (colon) / (forward slash) \ (backslash) | (vertical bar or pipe) ? (question mark) * (asterisk) " (double quote) @ (at) # (number sign) = (equals sign) ; (semicolon) " " (space) ' (single quote) ` (grave accent) {} (curly braces) () (parentheses) [] (square brackets) $ (dollar sign) Special characters/symbols in the project name cause the error message.

Option	Description
	Project name should be unique inside your workspace.
Use default location	Stores the files required to build the program in the Workbench's current workspace directory. The project files are stored in the default location. Clear the **Use default location** checkbox and click **Browse…** to select a new location.
Location	Specifies the folder that contains the project files. Click **Browse…** to navigate to the desired folder. This option is available only when **Use default location** checkbox is clear. Do not use the "Space" symbol in the folder name.
Processors tree control	Specifies the processor you would like to use in new project
ToolChain Selection table	Details on the selected processor: core kind, core name, and GCC toolchain that will be used to build the project. Depending on the selected processor, following options are available: • **Standard S32DS toolchain for ARM** Available for the following processors that you select in the **Processors** tree: • All processors except for S32V234 Cortex-A53. • **ARM Bare-Metal 32-bit Target Binary Toolchain** Available for the following processors that you select in the **Processors** tree: • All processors of S32K1xx family • S32V234 Cortex-M4 • **Standard S32DS toolchain for ARM Cortex-A** Available for the following processors that you select in the **Processors** tree: • Only S32V234 Cortex-A53. • **IAR Toolchain for ARM - (7.x)** **Note:** Requires an installed IAR Embedded Workbench for ARM 7.x and IAR plugin for ARM 7.x. If you do not see this option in the list, make sure to install IAR Embedded Workbench for ARM 7.x on the current computer first. After that, install IAR Embedded Workbench plugin manager in S32DS ARM v2018.R1 first, and then install IAR Embedded Workbench plugin for ARM 7.x by using the plugin manager. Lastly, verify that you have selected a supported processor. **Important:** If you have both IAR Embedded Workbench for ARM 7.x and IAR Embedded Workbench for ARM 8.x installed at the same time, this option is replaced by **IAR Toolchain for ARM - (legacy, 7.x)** once you install the IAR plugin for ARM 8.x by using IAR Embedded Workbench plugin manager. Available for the following processors that you select in the **Processors** tree: • All processors except for S32V234 Cortex-A53. • **IAR Toolchain for ARM - (legacy, 7.x)** **Note:** Requires an installed IAR Embedded Workbench for ARM 8.x and IAR plugin for ARM 8.x. If you do not see this option in the list, make sure to install IAR Embedded Workbench for ARM 8.x on the current computer first. After that, install IAR Embedded Workbench plugin manager in S32DS ARM v2018.R1 first, and then install IAR Embedded Workbench plugin for ARM 8.x by using the plugin manager. Lastly, verify that you have selected a supported processor. Available for the following processors that you select in the **Processors** tree: • All processors except for S32V234 Cortex-A53.

Option	Description
	• **IAR Toolchain for ARM - (8.x)** **Note:** Requires an installed IAR Embedded Workbench for ARM 8.x and IAR plugin for ARM 8.x. If you do not see this option in the list, make sure to install IAR Embedded Workbench for ARM 8.x on the current computer first. After that, install IAR Embedded Workbench plugin manager in S32DS ARM v2018.R1 first, and then install IAR Embedded Workbench plugin for ARM 8.x by using the plugin manager. Lastly, verify that you have selected a supported processor. Available for the following processors that you select in the **Processors** tree: • All processors of S32K1xx family • S32V234 Cortex-M4 • **GHS ARM Standalone Static Library Toolchain** **Note:** Requires an installed GHS plugin v2017.1.4. If you do not see this option in the list, make sure to install GHS Toolchain v2017.1.4 on the current computer first. Then verify that you have selected a supported processor. The only available debug option for project with GHS toolchain is Lauterbach TRACE32 debugger (and only PE Micro GDB server for MAC57D54H). Available for the following processors that you select in the **Processors** tree: • All processors except for S32V234 Cortex-A53.
Description field	Displays the project description

After you select a processor, next page of the wizard appears. Its options depend on the selected processor.

Cores and parameters

A few projects can be presented on this step if a multicore processor has been selected on the first step and only one project is presented on the this step if a single-core processor has been selected on the first step.

Use this page to select processors cores and parameters for them to build the library project: the programming language etc.

Table 6: New S32DS Library Project wizard – Page 2 settings

Option	Description
Project Name	See the library project name for the core displayed in the **Project Explorer** pane. Read only.
Core	Select required cores for library project. The set of cores depends on the selected processor. By default, all core checkboxes are checked
FPU Support (except S32V234 Cortex-A53)	Select to include floating point support in the project: • **Toolchain Default** • **Software: No FPU (-mfloat-abi=soft)** - select to include software floating point support in the project • **Hardware: -mfloat-abi=hard** - select to support hardware floating point. Using this option GCC build tools generates the code using hardware floating point instructions and uses FPU-specific calling convention: - mfloat-abi=hard (Cortex-M4 only)

Option	Description
	• **Hardware: -mfloat-abi=softfp** - select to support hardware floating point. This option allows GCC build tools to generate code using hardware floating point instructions, but still uses the soft-float calling conventions (Cortex-M4 only) • **Single Precision HW: -mfloat-abi=softfp -fshort-double** - select to support hardware floating point. Using this option GCC build tools generates code using hardware floating-point instructions, but still uses the soft-float calling conventions. Also, use same size for double as for float. Default: **Toolchain Default**.  **Warning:** The **-fshort-double** switch causes GCC to generate code that is not binary compatible with code generated without that switch. Use it to conform to a non-default application binary interface.
Language	The option you select sets up default compiler/linker options for the toolchain. Select the programming language that you want to use for writing the program's source code. You can select only one language: • **C** - sets up your application with ANSI C-compliant startup code, and initializes global variables • **C++** - sets up your application with ANSI C++ startup code, and performs global class object initialization. Default: **C**

Creating projects

The following topics explain the steps to create S32DS ARM v2018.R1 projects.

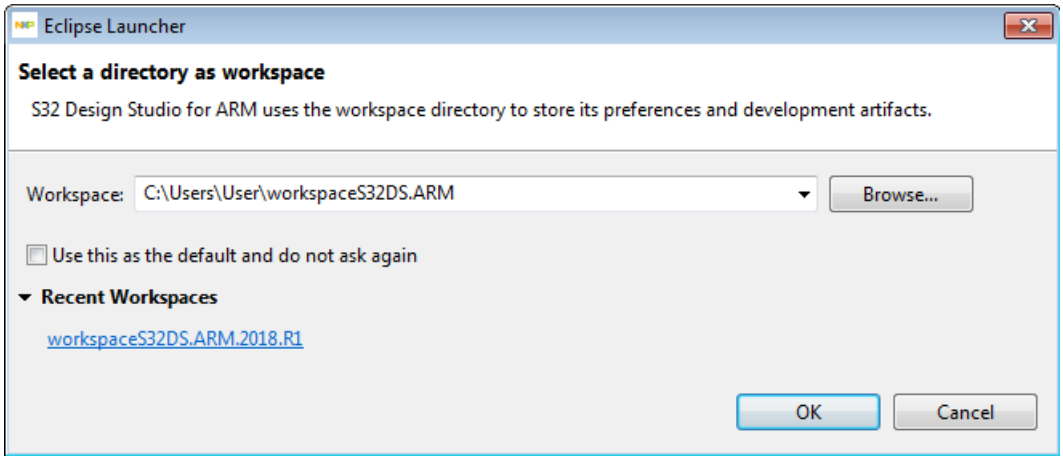
The wizard generates a set of projects with placeholder files and specific settings (build and launch configurations) for selected target. After the projects have been created, you can easily change any generated setting to suit your needs.

Launching Workbench

To start S32 Design Studio for ARM, Version 2018.R1 and begin working with it:

1. Select **Start** > **All Programs** > **NXP S32 Design Studio** > **S32 Design Studio for ARM, Version 2018.R1** > **S32 Design Studio for ARM, Version 2018.R1**. **Eclipse Launcher** dialog box appears and prompts you to select a workspace.

   Note: Workspace is a location on the file system where S32DS ARM v2018.R1 stores projects that you create or import into the IDE.



2. Depending on your choice, do one of the following:

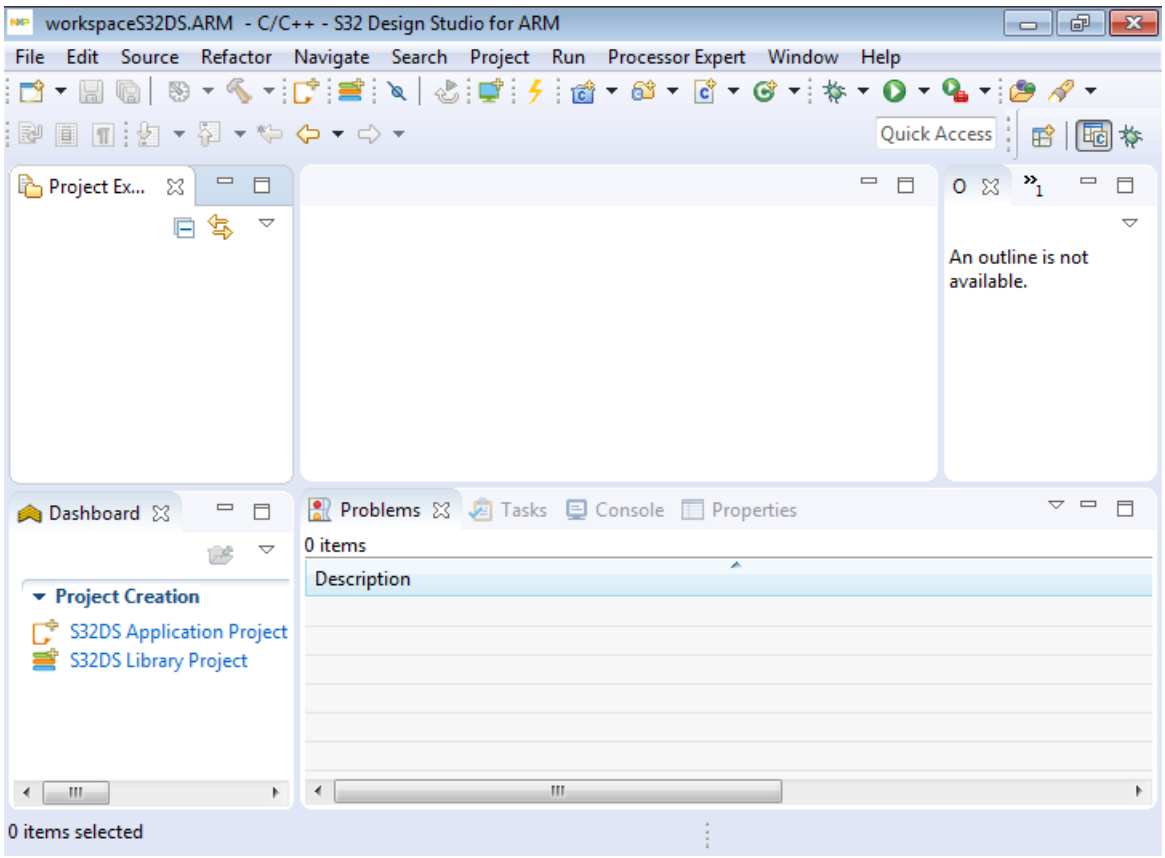
Choice	Option
Use the default workspace	Click **OK** to use the default workspace location.
Use a custom workspace	Create a folder on the file system by using a file manager. Click **Browse** and select the created location. Click **OK** to confirm the selection.
Use a previously used workspace	Expand **Recent Workspaces**, and then click one of the link buttons with the names of work spaces that you have previously used in this installation of S32DS ARM v2018.R1.

   Tip: You can view the list of previously used work spaces by selecting **Window** > **Preferences** > **General** > **Startup and Shutdown** > **Workspaces** in the menu bar.

3. [optional] Select the **Use this as the default and do not ask again** check box if you do not want S32DS ARM v2018.R1 to ask you to select the work space when you start it.

   Tip: You can change this later by clearing the **Prompt for workspace on startup** check box in **Window** > **Preferences** > **General** > **Startup and Shutdown** > **Workspaces**.

4. Click **OK**. S32DS ARM v2018.R1 starts and the main IDE window appears:



Creating S32DS Application Project

To create a new S32DS Application Project for a processor using the New S32DS Application Project wizard:

1. Select **File** > **New** > **S32DS Application Project** from the IDE menu bar. The first page of the **New S32DS Application Project** wizard appears.
2. Specify a name for the new project in the **Project name** text box.
3. Expand the Processors tree control and select the processor you would like to use. For example, **Family S32V** > **S32V234 Cortex-M4**.
4. Expand the **Toolchain** list and select the toolchain you would like to use.

   Note: Availability of the options depends on the processor you selected.
5. Click **Next**. The second page of the wizard appears.
6. Expand the controls and select following project settings:

   - Core
   - RAM Start Address (only S32V234 Cortex-A53)
   - RAM Size, KB (only S32V234 Cortex-A53)
   - Unused RAM, KB (only S32V234 Cortex-A53)
   - Library (except S32V234 Cortex-A53)
   - I/O Support (except S32V234 Cortex-A53)
   - FPU Support (except S32V234 Cortex-A53)
   - Language
   - SDKs
   - Debugger

   Note: Default values of the options depend on the processor you selected.

7. Click **Finish**. The wizard creates a new project according to your specifications. You can access the project from the **Project Explorer** view in the Workbench window. The new project is ready for use. You can now customize it by adding your own source code files, changing debugger settings, or adding libraries.

   Note: You can click **Cancel** at any step in the wizard to stop the project creation, discard all changes and close the wizard dialog box.

Creating S32DS Library Project

To create a new S32DS Library Project for a processor using the **New S32DS Library Project** wizard:

1. Select **File** > **New** > **S32DS Library Project** from the IDE menu bar. The first page of the **New S32DS Library Project** wizard appears.
2. Specify a name for the new project in the **Project name** text box.
3. Expand the **Processors** tree control and select the processor you would like to use. For example, **Family S32V** > **S32V234 Cortex-M4**.
4. Expand the **Toolchain** list and select the toolchain you would like to use.
5. Click **Next**. The second page of the wizard appears.
6. Expand the controls and select following project settings:
   - Core
   - Language
   - FPU Support (except S32V234 Cortex-A53)

   Note: Default values of the options depend on the processor you selected.
7. Click **Finish**. The wizard creates a new project according to your specifications. You can access the project from the **Project Explorer** view in the Workbench window. The new project is ready for use. You can now customize it by adding your own source code files, changing debugger settings, or adding libraries.

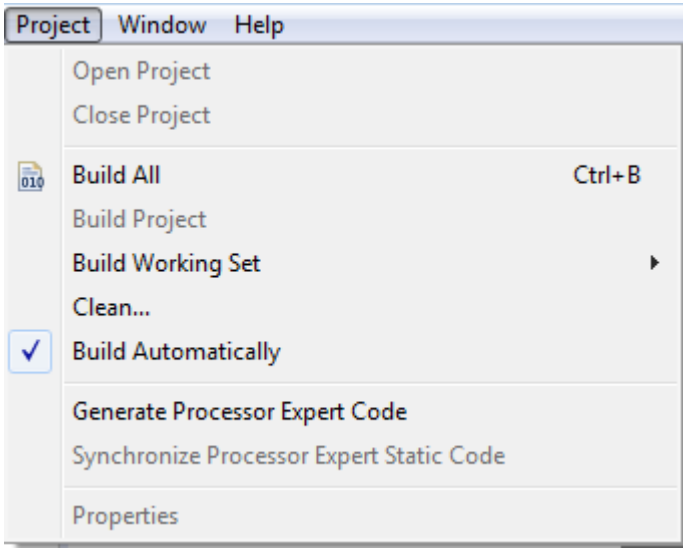
   Note: You can click Cancel at any step in the wizard to stop the project creation, discard all changes and close the wizard dialog box.

Building projects

The recently built S32DS ARM v2018.R1 project is pre-configured and you can easily build the project for your target board. However, if you want to change the configuration of the project you can adjust the build properties. For more information on build properties, see Build Properties for S32DS Projects.

In large workspaces, building the entire workspace can take a long time if you make changes with a significant impact on dependent projects. Often there are only a few projects that really matter to you at a given time.

To build only the selected projects, and any prerequisite projects that need to be built in order to correctly build the selected projects, select **Project** > **Build Project** from the S32DS ARM v2018.R1 menu bar or right-click on a selected project and select **Build Project** or click  .



Alternatively (to build all projects), select **Project** > **Build All**.

Monitor the generated command lines used to build the embedded application in the build **Console** view. Any problems with the build will be reported under the **Problems** view. Assuming the build is successful, the generated binary will be listed under the project in the **Project Explorer** view.

Debugging projects

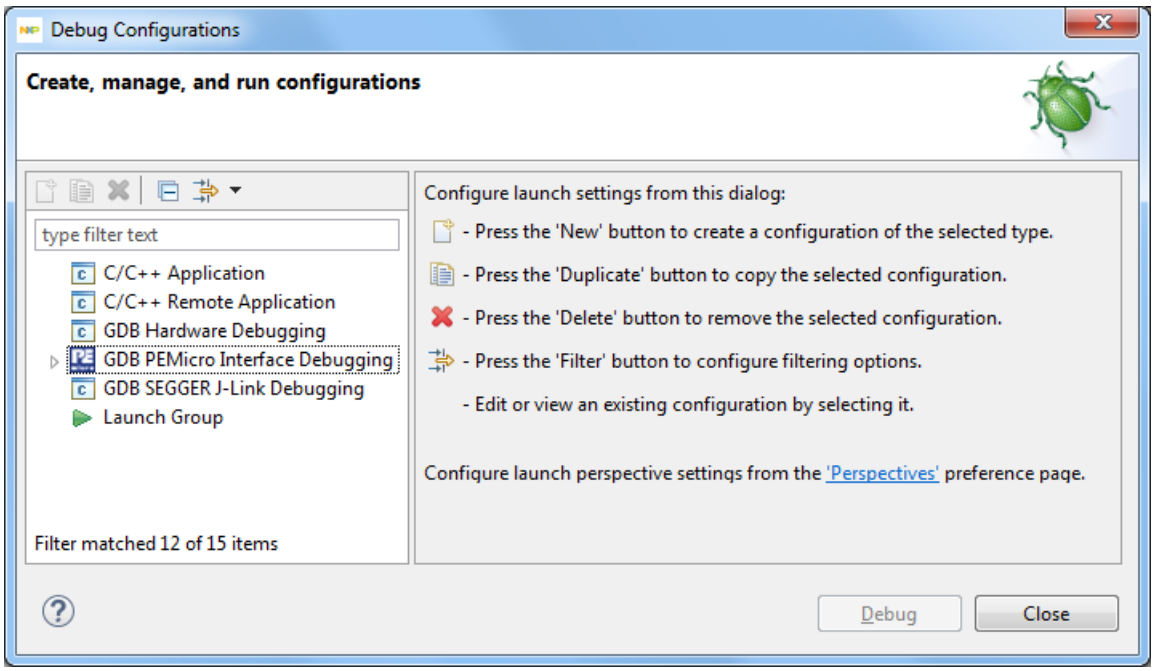
When you use the **S32DS Application Project** wizard to create a set of projects, the wizard sets the debugger settings of the project's launch configurations to specific values. You can change these generated values based on your requirements.

To debug a project, perform these steps:

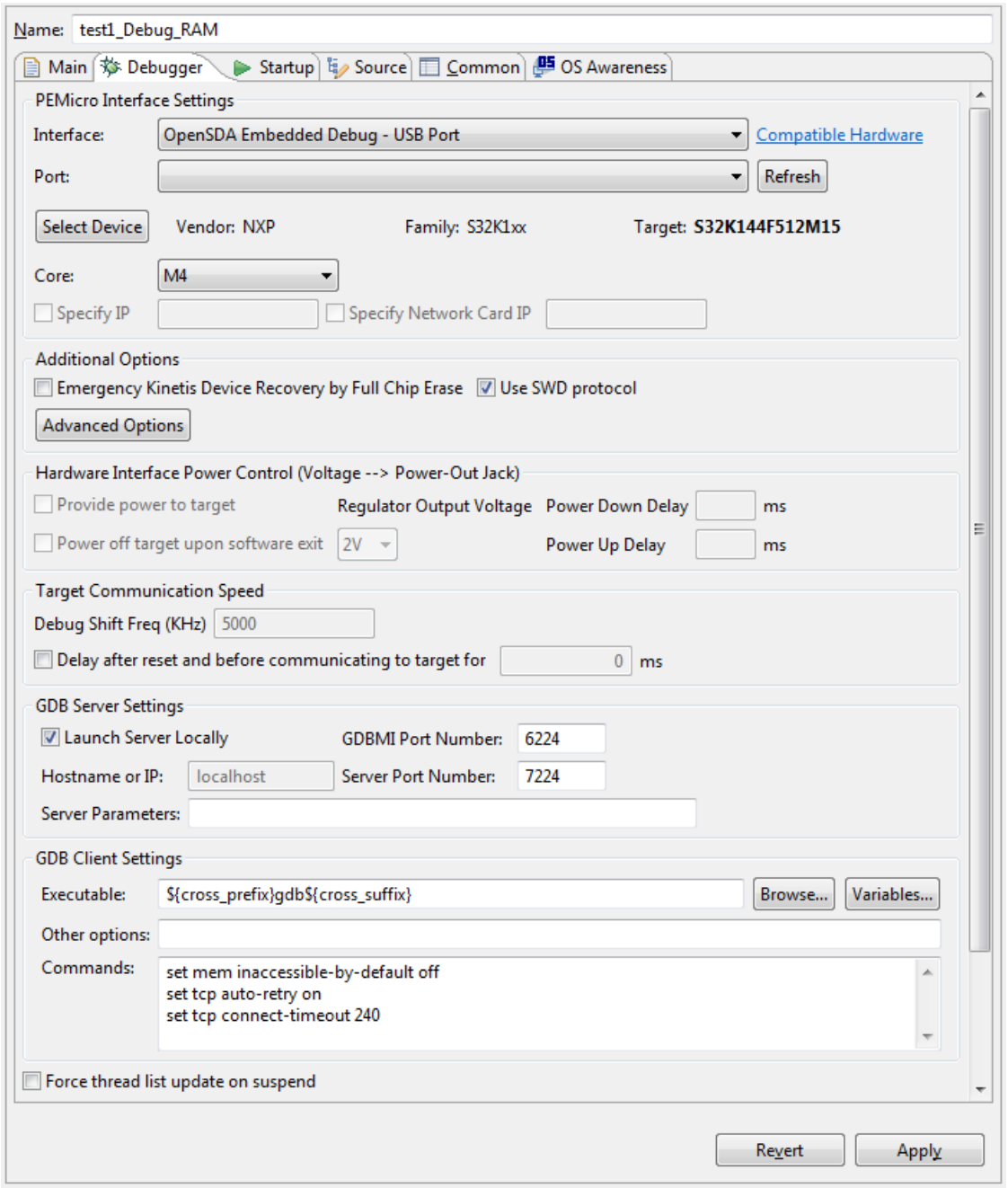
1. Open the **Debug** perspective.
2. From the main menu bar of the S32DS ARM v2018.R1, select **Run** > **Debug Configurations…**. Alternatively, you can click > **Debug Configurations…**. The **Debug Configurations** dialog box appears. The left side of this dialog box has a list of debug configurations that apply to the current application.

   Note: For more information on how to use the debugger, refer to the **S32DS ARM v2018.R1 Common Features Guide** and the Working with Debugger chapter of this manual.
3. Expand the tree.
4. From the expanded list, select the debug configuration that you want to modify.

   The figure below displays the **Debug Configurations** dialog box:



5. In the **Main** tab, ensure that the correct **Project** and **C/C++ Application** are selected.
6. Click the **Debugger** tab.



7. Based on the debug interface you have selected, change the appropriate debugger settings.
8. Select the **Startup** tab. Based on the debug interface you selected, change the appropriate startup settings.

   Note: For more information on debugger, refer to the chapter Working with Debugger.
9. Click **Apply** to save the new settings.
10. Click **Debug** to start the debugging session. The S32DS ARM v2018.R1 uses the settings in the launch configuration to generate debugging information and initiate communications with the target board.

    Note: If the **Debug** perspective is not open, you will be prompted to open the **Debug** perspective. Select **Yes** to switch perspective. The **Debug** perspective will open and the embedded application will break on the breakpoint set on main.

    You just finished starting a debugging session in the **Debug** perspective and attaching the debugger to a process.

Note: If the default USB port identifier in debug configuration does not correspond to the USB port ID of the connected device, some error messages, for example the "No source available" alert message, may show.

To solve this, follow the instruction:

1. Open the **Debug Configuration** window.

2. Select used launch configuration.

3. Open the **Debugger** tab.

4. Check the **Port** value, then check whether you have your device connected to PC via USB or Serial or Ethernet. If the port has assigned value and the **Apply** button is enabled, it means that your device was recognized and the setting was updated.

5. Click **Apply** then **Debug** buttons. You should get debugging started correctly.

For detailed descriptions of debug configuration for GDB *PEMicro* interface please refer to **P&E GDB Server Plug-In for Kinetis Devices. Debug Configuration User Guide** in the S32DS ARM v2018.R1 Help (S32DS ARM v2018.R1 > Common Manuals).

You can click **Revert** to undo any of the unsaved changes. The S32DS ARM v2018.R1 restores the last set of saved settings to all pages of the **Debug Configurations** dialog box. Also, the IDE disables **Revert** until you make new pending changes.

Note: If you have closed the project, related debug configurations are displayed only if they are located in the workspace, not in the project itself, and **Filter Closed Projects** is unchecked.

Closing and deleting projects

Close unused projects. Eclipse caches files for all open projects in the workspace. If you need multiple projects open, try to limit the number of projects to no more than 10.

To delete a project, follow these steps.

1. Select the project you want to delete in the **Project Explorer** view.
2. Select **Project** > **Delete**.

   The **Delete Resources** dialog box appears.

   Note: Alternatively, you can also select **Delete** from the context menu when you right-click on the project.
3. Check the **Delete project contents on disk (cannot be undone)** checkbox if you want to delete the contents of the selected project. Else, clear the **Delete project contents on disk (cannot be undone)** checkbox.

   Note: You will not be able to restore your project using **Undo**, if you select the **Delete project contents on disk (cannot be undone)** option.
4. Click **OK**.

   The project is removed from the **Project Explorer** view.

   Note: To clean the project, right click on the project in the **Project Explorer** view and select **Clean Project**. Once cleaned select **Build Project.**

If you delete MAC57D54H projects for Cortex-A5 or Cortex-M0+ cores, make sure to remove the dependencies for these projects from the project for the Boot Cortex-M4F core:

1. Delete unnecessary cores from the list of **Defined symbols** in **Properties** > **C/C++ Build** > **Settings** > **Standard S32DS C Compiler** > **Preprocessor** for all configurations.
2. Delete unnecessary cores from the list of **Other objects** in **Properties** > **C/C++ Build** > **Settings** > **Standard S32DS C Linker** > **Miscellaneous** for all configurations.
3. Uncheck or remove launch configurations for unnecessary cores in **Debug Configurations**.

Exporting project information

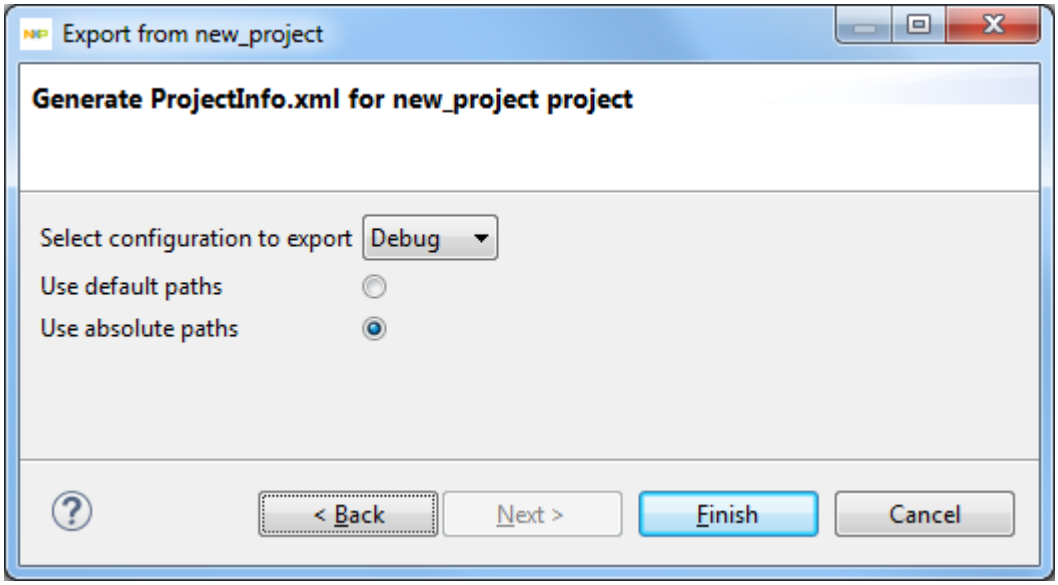
Every application project contains configuration settings also called *project information*. You can export these settings so that you can later use them in newer projects. This section explains how to export project information file (a **ProjectInfo.xml** file) from an existing S32DS application or library project by using the **Export** wizard.

To export a project information file:

1. In the **Project Explorer** view select an application project for which you want to export the **ProjectInfo.xml** file.

   Note: You can export information only one core at a time. If you export project information from a multicore project, repeat export operation for each of the project cores.
2. On the **File** menu, select **Export…**.
   The **Export** wizard opens.
3. Expand the tree control and select **S32 Design Studio** > **Project Info Export Wizard**. Click **Next**. The Generate **ProjectInfo.xml** page appears:

   Note: Exporting is only supported for projects targeted for C language.



4. Select the build configuration to export and choose how you want the wizard to store the paths to files referenced in your project within the `ProjectInfo.xml` file.

Choice	Option
Select configuration to export	Select the build configuration from which to export toolchain settings for compiler, linker, and assembler tools, specific to selected build configuration. Available build configurations are the same as found when you expand the **Build** button on the toolbar. By default, **Debug** is selected.
Use default paths	Select this if you want the wizard to store paths to resources referenced in the project relative to the project root. For example, location of a linker file will be specified in the exported file as a path relative to the current location of the project: `Project_Settings\Linker_Files\<SoC>_<configuration>.ld`

Choice	Option
Use absolute path	Select this if you want the wizard to store full absolute paths to resources referenced in the project. For example, location of a linker file will be specified in the exported file as an absolute path: `C:\Users\User\workspaceS32DS.ARM\Application\Project_Settings\Linker_Files\<SoC>_<configuration>.ld` This is the default setting.

5. Click **Finish**.

The wizard will export project information to the `ProjectInfo.xml` file inside the location of the source project within the current workspace location. For example, if you are importing project information for the project named **Application**, the `ProjectInfo.xml` file will be exported to the **Application** folder in side the current workspace:

```
C:\Users\User\workspaceS32DS.ARM\Application\ProjectInfo.xml
```

Note: If an old `ProjectInfo.xml` file already exists in the project location, it will be replaced with the new one.

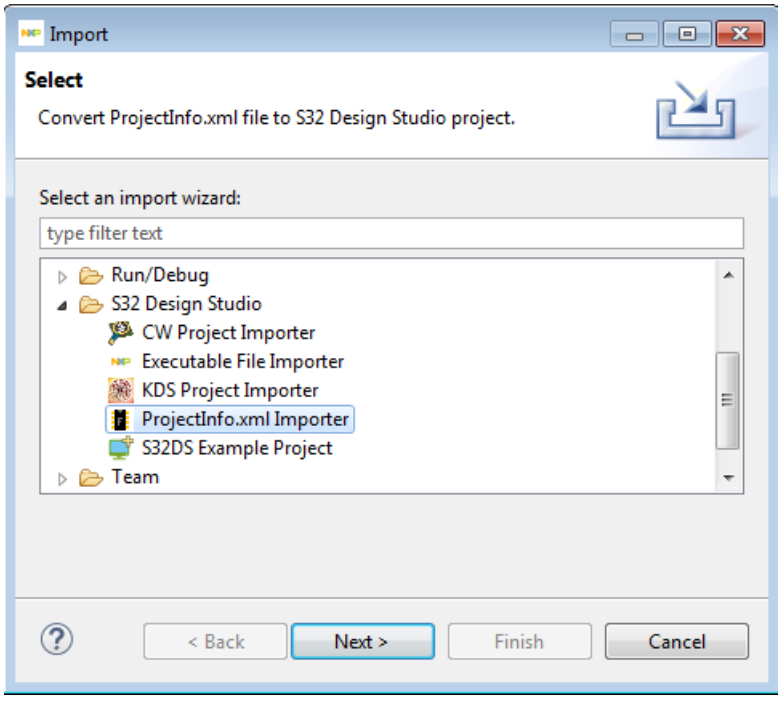
Importing project information

This section explains how to export project information file (a **ProjectInfo.xml** file) from an existing S32DS application or library project by using the **Import** wizard.

To export a project information file:

1. On the **File** menu, click **Import…**.

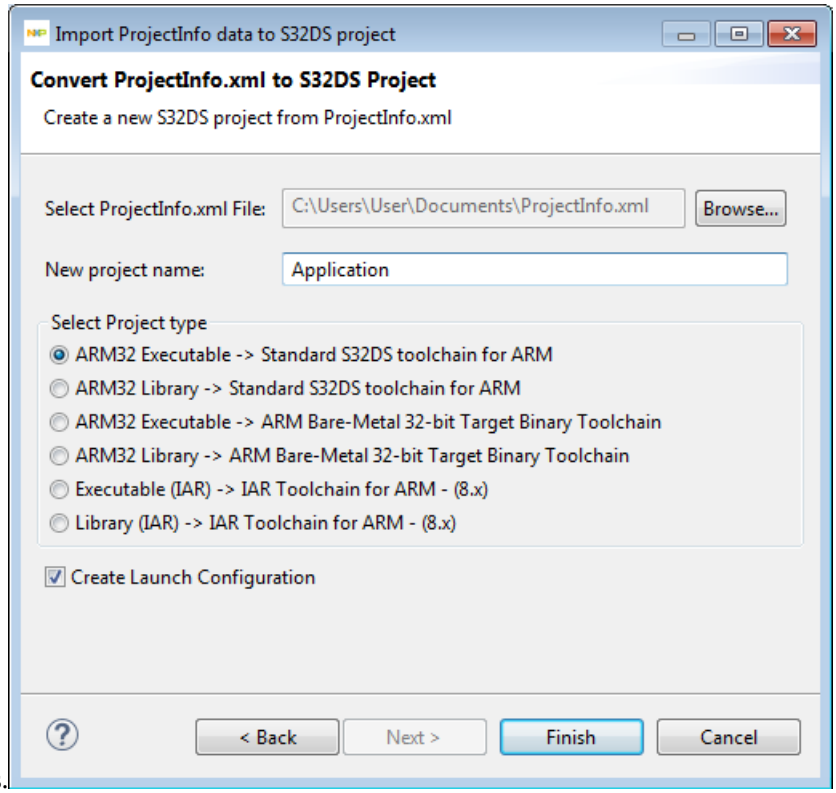
   The **Import ProjectInfo data to S32DS project** wizard opens.
2. Expand **S32 Design Studio**, and then select **ProjectInfo.xml Importer**.



3. Click **Next**.

The **Import Project Info data to S32DS project** window appears with file import options.

The **Import ProjectInfo data to S32DS project** wizard window opens.



4. Click **Browse** to select the project information file to import.
5. In the **New project name** field specify a unique name for the project.

   As a result of importing the project information file, the **Import** wizard creates a new project file in the current workspace and then configures project settings based on the data present in the project information file. Make sure that you specify a unique name for the project. This name does not have to be the same as the name of the source project used to generate the `ProjectInfo.xml` file. If the current workspace already contains a project with the name that you want to use for the new project, give the project some temporary name. You will be able to rename conflicting projects later after the import is done.
6. Select the type of the project that you want to be created.

   Note: Depending on the toolchain specified in the source project, different set of options will be available in the **Select Project type** group. Available options reflect the supported toolchain in the source project. For example, if the source project was created for the Standard S32DS toolchain for ARM toolchain (based on GCC 4.9), available for KEA processor family, only the options for ARM32 are available. If the source project was created for Standard S32DS toolchain for ARM Cortex-A (toolchain available for S32V234 processor family based on Cortex-A53 core), only the options for ARM64 are available. Option to select IAR Toochain for ARM is available only if the source project was created for a toolchain provided with IAR Embedded Workbench®.

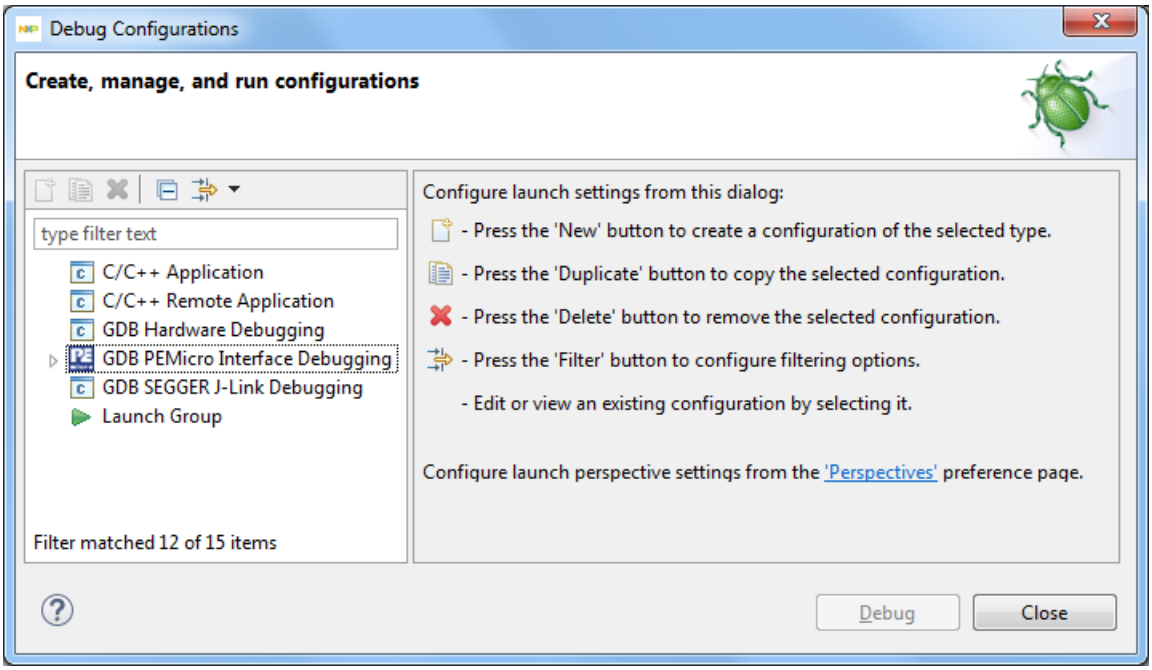
   Note: Always select the type compatible with the toolchain specified in the source project. For example, if the source project was created for a 32-bit GCC 4.9 compiler, select **Standard S32DS toolchain for ARM**.

   - **ARM32 Executable # Standard S32DS toolchain for ARM**
   - **ARM32 Library # Standard S32DS toolchain for ARM**
   - **ARM32 Executable # ARM Bare-Metal 32-bit Target Binary Toolchain**
   - **ARM32 Library # ARM Bare-Metal 32-bit Target Binary Toolchain**
   - **ARM64 Executable # Standard S32DS toolchain for ARM Cortex-A**
   - **ARM64 Library # Standard S32DS toolchain for ARM Cortex-A**
   - **Executable (IAR) # IAR Toolchain for ARM - (8.x)**

- **Library (IAR) # IAR Toolchain for ARM - (8.x)**

7. Select the **Create Launch Configuration** check box if you want the wizard to open **Debug Configurations** window during the import process.

   The window allows you to recreate debug and launch configurations for the newly created project, see "Customizing Launch Configuration" for more details on configuring debug and launch configurations.
8. Click **Finish**.
9. [optional] In the opened **Debug Configurations** window, configure settings, and then click Debug to start the debugging session immediately.



10. Locate the files referenced in the project information file that you import and copy them to the locations specified in the `copyErrorLog.txt` that opens in the **Editor** view.

    Note: If the source folder contained some header files, implementation files, startup files, or any other user files referenced in the project, you have to copy them from the backup. The project information file contains only the project settings and does not contain referenced files.

The new project will be created in your workspace inheriting settings from the imported project information file.

Chapter 3

Build properties for S32DS projects

Topics:

This chapter explains build properties for a project. The project creation wizard, **S32DS Application Project** or **S32DS Library Project**, uses the information it gathers from you to set up the project's build settings and debug configurations.

A project's build settings contain information on the tool settings used to make the program. For example, it describes the compiler and linker settings, and the files involved, such as source and libraries.

A project's debug configuration describes how the IDE starts the program, such as whether it executes by itself on a target, or under debugger control. Debug configurations also specify the core the program executes on (if the target processor has multiple cores). They also specify the connection interface and communications protocol that the debugger uses to control the environment that the program executes in.

When the project wizard completes its process, it generates debug configurations with names that follow the pattern:

<projectname>_<configtype>_<targettype>, where:

- **<projectname>** - represents the name of the project
- **<configtype>** - represents the build configuration's name
- **<targettype>** - represents the type of target hardware on which the debug configuration acts

For each project, you can specify build settings, such as:

- additional libraries to use for building code
- behavior of the compilers, linkers, assemblers, and other build-related tools
- specific build properties, such as the byte ordering of the generated code

Changing Build Properties

The **S32DS Application Project** and **S32DS Library Project** wizards define the build properties for the project. You can modify these build properties to better suit your needs.

Perform these steps to change build properties:

1. Start S32DS ARM v2018.R1.
2. In the S32DS ARM v2018.R1 **Project Explorer** view, select the project for which you want to modify the build properties.
3. Select **Project** > **Properties** from the S32DS ARM v2018.R1 menu bar. The **Properties for <project name>** window appears. The left side of this window has a properties list. This list shows the build properties that apply to the current project.
4. Expand the **C/C++ Build** property. The **C/C++ Build** pane appears on the right.
5. Use the **Configuration** drop-down list (on the right pane) to specify the build configuration for which you want to modify the build settings:
6. Change the settings that appear in the pane.
7. Click the **Apply** button at the bottom of the right pane. The S32DS ARM v2018.R1 saves your new settings. You can select other tool pages and modify their settings.
8. When you finish, click **OK** to save your changes and close the **Properties for <project name>** window.

Restoring build properties

If you modify a build configuration that the new project wizard generates, you can restore that configuration to its default state. You might want to restore the build properties in order to have a factory-default configuration, or to revert to a last-known working build configuration. To undo your modifications to build properties, click the **Restore Defaults** button at the bottom of the **Properties for <project name>**window.

This changes the values of the options to the absolute default of the toolchain. The **Restore Defaults** button changes the values to the toolchain-default NOT to the specific values were set after the project creation was finished.

For example, when a project is created the **Tool Settings** > **Target Processor** panel has some values set, which are specific to the project. Clicking the **Restore Defaults** button defaults the values of options.

Defining C/C++ Build Settings

The **Properties for <project>** > **C/C++ Build** page includes all builder-specific property pages.

Note: Modifying settings such as the **Generate makefiles automatically** option, might enable or disable some parameters in some situations and change the availability of other property pages.

To define C/C++ build settings, perform these steps:

1. Start the S32DS ARM v2018.R1.
2. In the S32DS ARM v2018.R1 **Project Explorer** view, select the project for which you want to modify the builder settings.
3. Select **Project** > **Properties**.

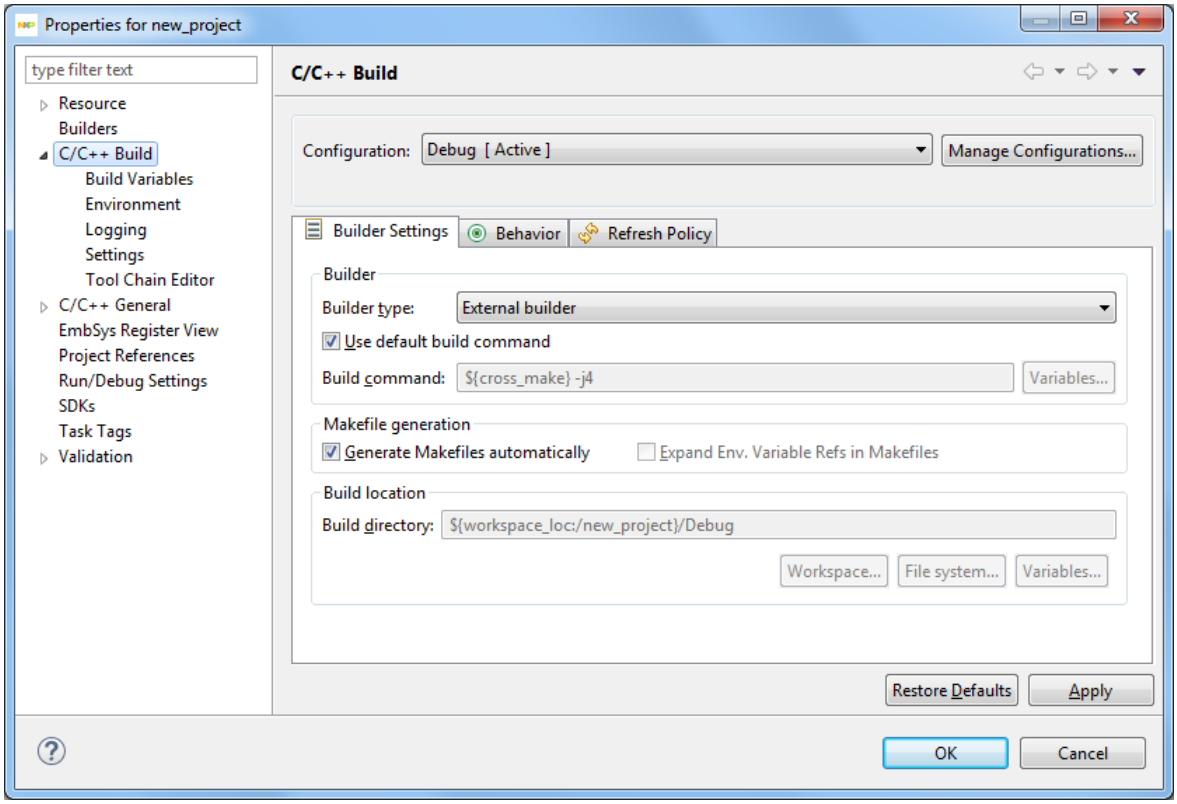
   The **Properties for <project name >** window appears. The left side of this window has a properties list. This list shows the build properties that apply to the current project.
4. Select **C/C++ Build**.

   The **C/C++ Build** page appears. Select one of the tabs:

   - **Builder Settings** to define builder settings
   - **Behavior** to define build behavior

Define Builder Settings

To define builder settings, on the **C/C++ Build** page click the **Builder Settings** tab.



The builder settings for the selected build configuration appear. The table below describes the builder settings options.

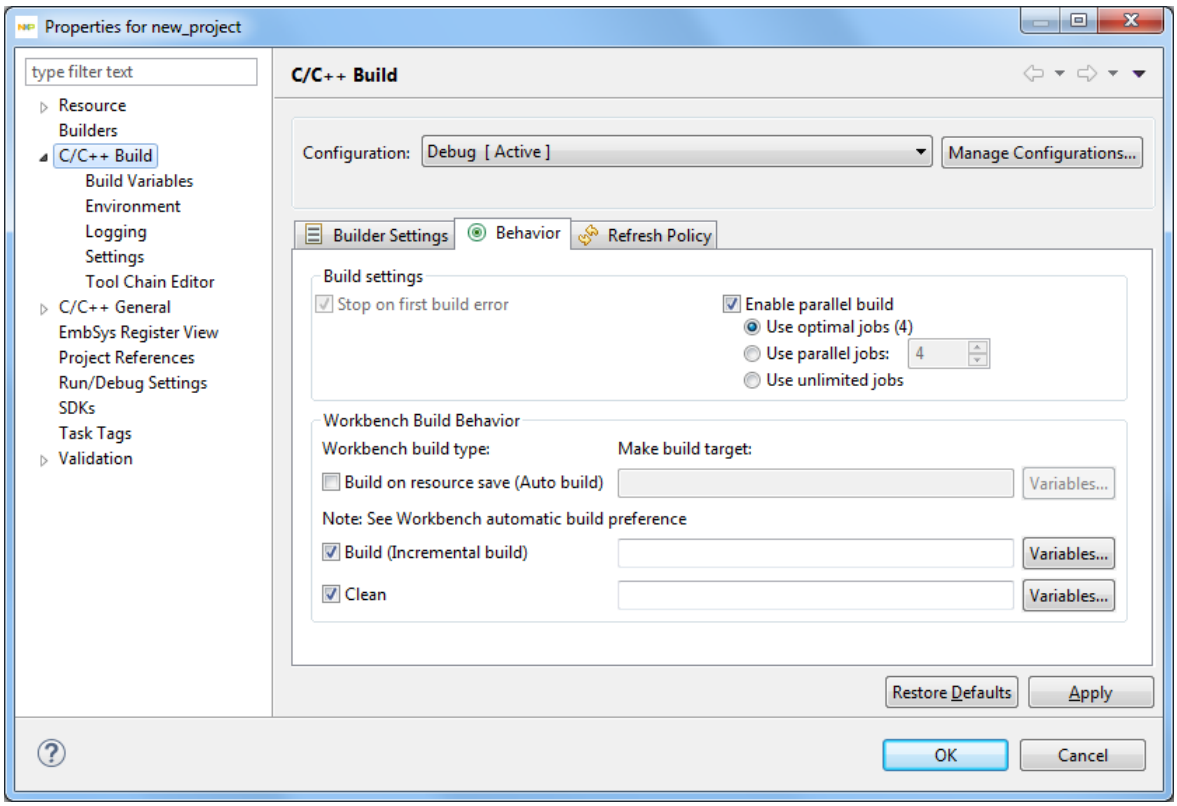
Table 7: Builder Settings Options

Group	Option	Description
	Configuration	Specifies the build configurations for the selected project.
	Manage configurations	Click to open the **Manage Configurations** dialog box that lets you set configurations based on the specified toolchains of the selected project. You can also create new build configurations, rename an existing configuration, or remove the ones that are no longer required.
Builder	Builder type	Specifies the type of builder to use: • **Internal builder** – Builds C/C++ programs using a compiler that implements the C/C++ Language Specifications. • **External builder** – External tools let you configure and run programs and Ant buildfiles using the Workbench, which can be saved and run at a later time to perform a build.
	Use default build command	Check to indicate that you want to use the default make command. Clear when you want to use a new make command. This option is only available when the **Builder type** option is set to **External**.

Group	Option	Description
	Build command	Specifies the default command used to start the build utility for your specific toolchain. Use this field if you want to use a build utility other than the default make command.
	Variables…	Click to open the **Select build variable** dialog box and add the desired environment variables and custom variables to the build command.
Makefile generation	Generate Makefiles automatically	Check to enable S32DS ARM v2018.R1 change between two different CDT modes: it either uses the customer's makefile for the build, if one exists, or it generates makefiles for the user.
	Expand Env. Variable Refs in Makefiles	Check to define whether environment variables should be expanded in makefile.
Build location	Build directory	Specifies the location where the build operation takes place. This location will contain the generated artifacts from the build process. This option appears disabled when the **Generate Makefiles automatically** option is enabled.
	Workspace…	Click to open the **Folder Selection** dialog box and specify the build directory by selecting from *workspace*. The directory will contain all artifacts created during project build.
	File system…	Click to open the **Browse For Folder** dialog box and specify the build directory by selecting it from *file system*.
	Variables…	Click to open the **Select build variable** dialog box and select a variable to specify as an argument for the build directory, or create and configure simple build variables which you can reference in build configurations that support variables.

Define Build Behavior

To define build behavior, on the **C/C++ Build** page switch to the **Behavior** tab.



The build behavior settings for the selected build configuration appear. The table below describes the build behavior options.

Table 8: Behavior Options

Group	Option	Description
Build settings	**Stop on first build error**	Check to stop building when S32DS ARM v2018.R1 encounters an error. Clearing this option is helpful for building large projects as it enables make to continue making other independent rules even when one rule fails.
	Enable parallel build	Activates generation of parallel builds. You need to determine the number of parallel jobs to perform: • **Use optimal jobs number** - Lets the system determine the optimal number of parallel jobs to perform. • **Use parallel jobs** – Lets you specify the maximum number of parallel jobs to perform. • **Use unlimited jobs** – Lets the system perform unlimited jobs. By default, this check box is selected.
Workbench Build Behavior	**Workbench build type**	Specifies the builder settings when instructed to build, rebuild, and clean.
	Build on resource save (Auto build)	Build your project whenever resources are saved.

Group	Option	Description
		By default, S32DS ARM v2018.R1 rebuilds the project automatically whenever you modify project resources. Clear the check box if you want to control the build manually. By default, this check box is selected.
	Build (Incremental build)	Defines what the standard builder will call when an incremental build is performed.
	Variables	Click to open the **Select build variable** dialog box and add variables to the make build target command.
	Clean	Defines what the standard builder calls when a clean is performed. The make clean is defined in the makefile.

C/C++ Build Tool Settings

Each project contains a collection of settings that define how you want S32DS ARM v2018.R1 to build the project. This collection is known as build configuration and contains information about *build tool settings*.

The build tool settings configure S32DS ARM v2018.R1 to use proper tools based on the target SoC platform and sets tool options to handle all stages of the build process. Each tool is part of the GNU Arm Embedded Toolchain prebuilt by NXP and is used by S32DS ARM v2018.R1 to handle specific task during the process of building your projects. The settings are available on the **Settings** page under **C/C++ Build** section in project properties.

Tip: Right-click the project in the **Project Explorer** view, and then select **Properties** to open project properties.

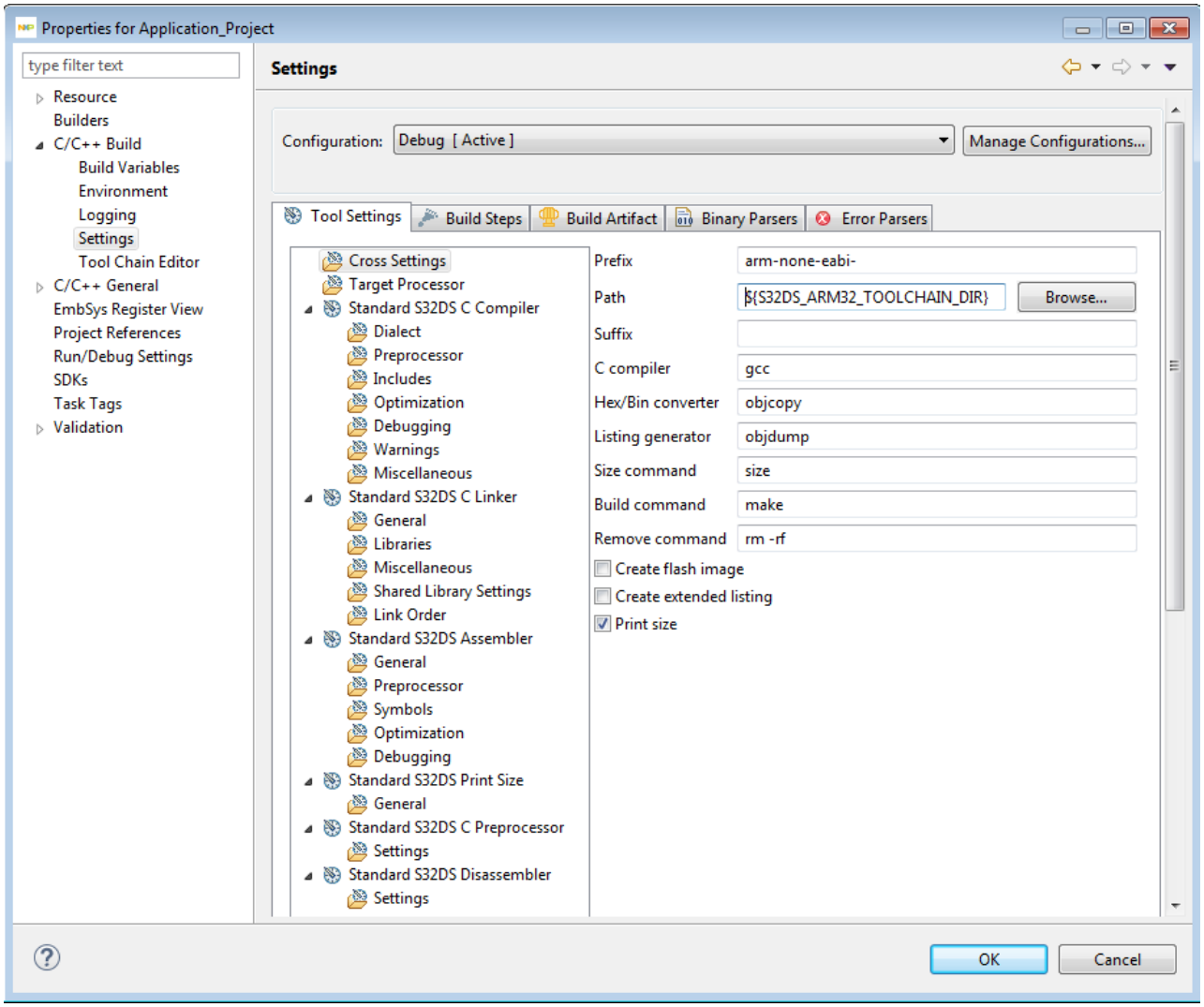
These build tool settings are automatically configured based on the target processor selected in the project creation wizard and configurations you have made to core settings within the wizard. A tree on the **Tool Settings** tab lists tools available in the toolchain and depends on the chosen wizard and build artifact it creates — application or library. A single tree item represents a group of settings for a specific tool in the toolchain or general settings that configures the toolchain as a whole. For example, **Cross settings** groups settings for the toolchain, when **Standard S32DS C Compiler** and **Standard S32DS C++ Compiler** items group settings for C and C++ compiler tools. If you have specified **Standard S32DS toolchain for ARM**, you would see the list of settings for GNU Arm Embedded Toolchain. If you used **IAR Toolchain for ARM - (7.x/7.x, legacy/8.x)**, you would see the list of settings for IAR Toolchain. Each node provides a set of settings for a specific tool provided by the selected toolchain.

Following toolchains are supported:

- Standard S32DS toolchain for ARM (GCC toolchain 4.9.3 is used)
- ARM Bare-Metal 32-bit Target Binary Toolchain (GCC toolchain 6.3.1 is used)
- IAR Toolchain for ARM – (7.x/7.x, legacy/8.x)
- Green Hills Software (GHS) Toolchain (v2017.1.4)

Each page that opens as you click items in the tree on the **Tool Settings** tab provides settings specific for the selected tool. For example, the **Standard S32DS C Compiler** page provides settings that configure compiler that will be used to build C code.

Some items may represent a virtual tool and appear only if you turn on some option in **Cross Settings**. For example, the **Standard S32DS Create Listing** tool shows in the tree only after you select the **Create extended listing** check box in **Cross Settings**. The **Standard S32DS Create Listing** tool does not represent a separate tool on the file system. Instead, it groups settings that allow S32DS ARM v2018.R1 to invoke the object dump tool specified in **Listing Generator**.



Default values for the settings available on the pages depend on the debug configuration selected from the **Configuration** list. You can modify the toolchain and its tools on the **Tool Chain Editor** page that opens when you select **C/C++ Settings** > **Tool Chain Editor** in the **Properties for <project_name>** window.

Note: If your project is created for IAR Toolchain for ARM - (7.x), the toolchain that is shipped with IAR Embedded Workbench® 7.x, and you switch to IAR Embedded Workbench® 8.x, make sure to select **IAR Toolchain for ARM - (legacy, 7.x)** to build the project. Building of projects created for IAR Toolchain for ARM - (7.x) by using IAR Toolchain for ARM - (8.x) is not supported.

Note: When building your project, you must use the compiler that was specified during the creation of the project. If your project uses GCC toolchain 4.9.3, make sure to select this specific toolchain to build the project. Building projects that were created for GCC compiler version 4.9.3 by using compiler version 6.3.1 is not supported.

Warning: Building of projects created for GHS Toolchain v2014.x by using GHS Toolchain v2017.x is not supported. If you import a project created for GHS Toolchain v2014.x, some options, such as FPU Support, may not be available.

Build tools and their settings are:

- **Cross settings**

  Location of tools in the toolchain and names of the tools from the GNU Binutils collection: compiler, linker, assembler, disassembler, size, and archiver tools.
- **Target processor**

Core customizations specific to the target processor selected in the project creation wizard. Part of these settings you configure when creating the project. The page allows you to further tailor settings related to the target processor on which you are going to run your application or library.

- **Standard S32DS C Compiler** and **Standard S32DS C++ Compiler**

  Configuration of C and C++ compilers for cross-compilation for the target ARM® core on your current platform. By default the project is configured to use `gcc` for C code and `g++` for C++ code, as specified in **Cross Settings**.

  You can tune optimization by setting specific flags, configure how warning messages are output during the compilation stage of the build process, define `include` location or add separate header files and specify debug settings such as debugging level and format.
- **Standard S32DS C Linker** and **Standard S32DS C++ Linker**

  Configuration of C and C++ linker tool.
- **Standard S32DS C Assembler**

  Configuration of assembler for the target processor.
- **Standard S32DS Archiver**

  Configuration of the tool that creates the A file, archive that wraps the O file and builds a library for the library project.

  Note: This tool is only available for library projects.
- **Standard S32DS Create Flash Image**

  Configuration of the virtual tool that invokes **Hex/Bin converter** tool to create a binary image in Motorola S-record (SREC) format that can be flashed to the target processor.

  Note: This tool shows only if you select **Create flash image** check box on the **Cross Settings** page.
- **Standard S32DS Create Listing**

  Configuration of the virtual tool that invokes **Listing generator** tool to create LST file with disassembly of the produced ELF file.

  Note: This tool shows only if you select **Create extended listing** check box on the **Cross Settings** page.
- **Standard S32DS Print Size**

  Configuration of the size tool that will output size details for the resulting binary after the build is complete.
- **Standard S32DS C Preprocessor** and **Standard S32DS C++ Preprocessor**

  Configuration of the preprocessor tool that allows you to specify how you want the preprocessor to expand macro and processor directives.
- **Standard S32DS Disassembler**

  Configuration of the disassembler tool that allows you to generate the disassembly for targets of the application project (ELF file), and library project (A file), as well as the C source code files in the `./src` folder of your project.

Cross Settings

This section describes cross settings that can be configured on the **Cross Settings** page. The page is available in the properties of the project that you create with S32DS ARM v2018.R1.

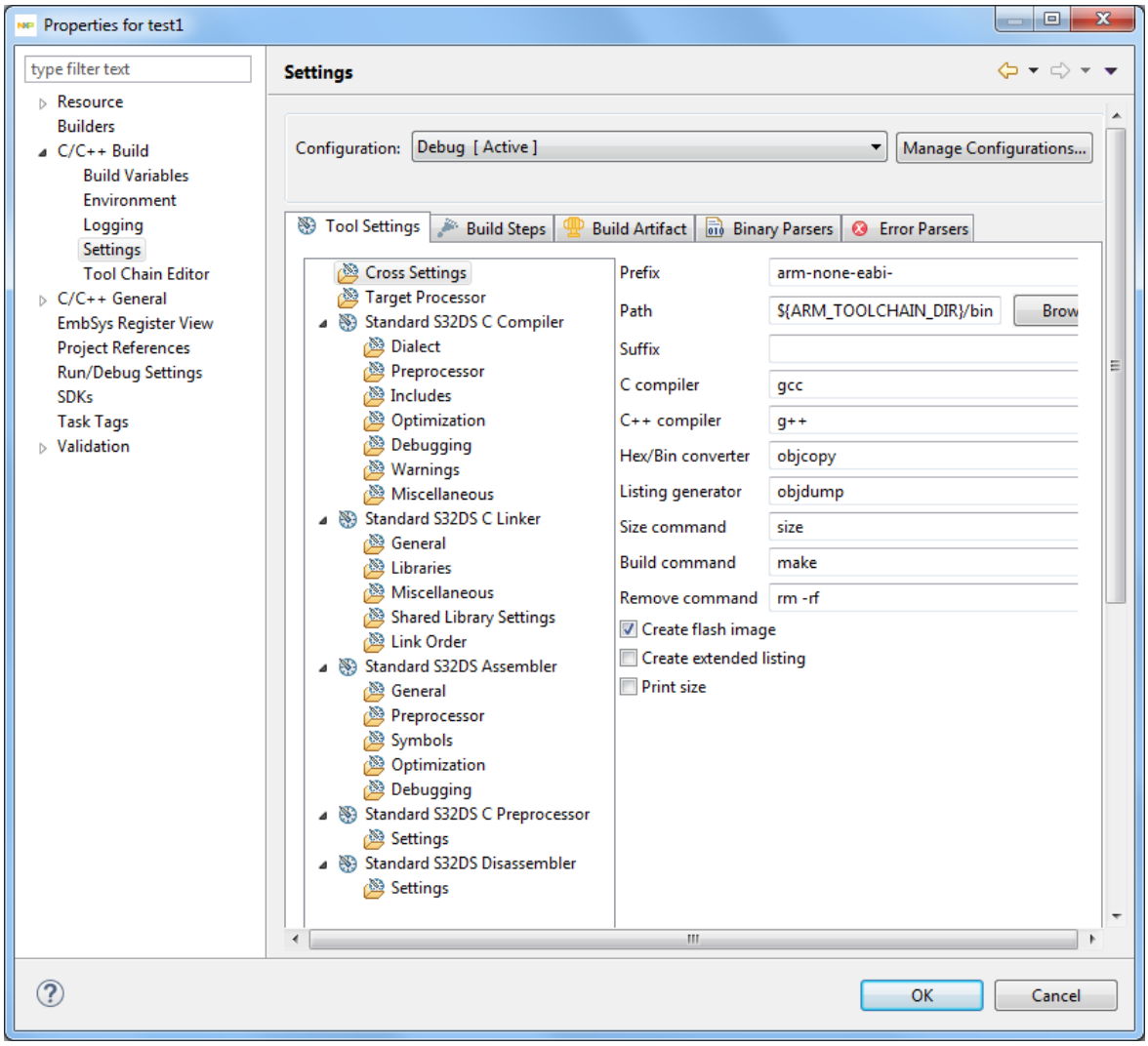
Tip: Right-click the project in the **Project Explorer** view, and then select **Properties** to open project properties. Select **C/C++ Build** > **Settings**, and then, on the **Tools Settings** tab click **Cross Settings** to show the settings of this page.

The **Cross Settings** page allows you to configure the location and names of tools in Linaro binary toolchain and GNU toolchain prebuilt by NXP. The Linary binary toolchain is used if you select the GCC 4.9 toolchain (corresponds to the **Standard S32DS toolchain for ARM** option) on the **Create a S32 Design Studio Project** page of **S32DS Application Project** or **S32DS Library Project** wizards. The GNU toolchain prebuilt by NXP is used if you select

the GCC 6.3 toolchain (corresponds to the **ARM Bare-Metal 32-bit Target Binary Toolchain** option) in the wizards.

You can configure following tools: compiler, linker, assembler, disassembler, and size. Additionally, you can configure names of the `build` and `remove` commands.

Note: Settings on this page depend on the project creation wizard you use and target processor that you select in the wizard.



When S32DS ARM v2018.R1 runs a specific part of the build process — compilation, linking, assembling — it looks for the tools by using path and names specified here. Depending on the target artifact and the wizard you use, different tools may be necessary to build the project. When the wizard creates a project, it prepares the settings on the **Cross Tools** page so that it builds by using the proper tools. For example, projects for the processors based on ARM® A53 core require a 64-bit toolchain. The **Prefix** and **Path** settings for this core will be different that in the project created for M4 core that requires 32-bit tools.

Settings specified here are later used on configuration pages for specific tools. For example, command to start the compiler looks as:

```
${cross_prefix}${cross_c}${cross_suffix}
```

Placeholders correspond to the following settings available on the **Cross Tools** page:

- ${cross_prefix} — takes its value from the **Prefix** field.

- ${cross_c} — takes its value from the **C compiler** field.
- ${cross_suffix} — takes its value from the **Suffix** field.

Available tools can be found in `${eclipse_home}../Cross_Tools`, where `${eclipse_home}` is the install location of S32DS ARM v2018.R1.

The following table describes the options available on the **Cross Settings** page.

Table 9: Cross Settings: Application Project and Library Project

Option	Description
Prefix	Toolchain prefix used by S32DS ARM v2018.R1 to call GNU Tools for ARM® Embedded Processors. Default setting depends on target processor that you have selected in the project creation wizard. Following values are possible: Cortex-M4F/M0+/A5: `arm-none-eabi-` Cortex-A53: `aarch64-elf-`
Path	Location of GNU Tools for ARM Embedded Processors. Default setting depends on target processor that you have selected in the project creation wizard. **Note:** Default settings use system variables in path names. These variables expand to their values at build time. You may find the list of system variables by selecting the **Show system variables** check box on the **WindowPreferencesC/C++BuildBuild Variables** pages in S32DS ARM v2018.R1 preferences. Following values are possible: Cortex-M4F/M0+/A5 and GCC 4.9: `${ARM_TOOLCHAIN_DIR}/bin` Cortex-M4F and GCC 6.3: `${S32DS_ARM32_TOOLCHAIN_DIR}` Cortex-A53: `${CORTEXA_TOOLCHAIN_DIR}/bin`
Suffix	Toolchain suffix used to call GNU Tools for ARM Embedded Processors. By default, this field is empty.
C compiler	Name of the C compiler executable. **Note:** S32DS ARM v2018.R1 uses the specified **Prefix** to locate the compiler. S32DS ARM v2018.R1 concatenates the value of the **Prefix** field with the value specified in this field. For example, name of the C compiler for Cortex-M4F core that S32DS ARM v2018.R1 will look for is: `arm-none-eabi-gcc`. By default, this field is set to: **gcc**
C++ compiler	Name of the C++ compiler executable. **Notice:** This setting is only available in the projects targeted for C++. If you do not see this field, create a new project, and then select **C++** in the **Language** list on the **New S32DS Project for <processor_name>** page of the project creation wizard. **Note:** S32DS ARM v2018.R1 uses the specified **Prefix** to locate the compiler. S32DS ARM v2018.R1 concatenates the value of the **Prefix** field with the value specified in this field. For example, name of the C compiler for Cortex-M4F core that S32DS ARM v2018.R1 will look for is: `arm-none-eabi-g++`.

Option	Description
	By default, this field is set to: **g++**
Archiver	Name of the tool to create archives. **Notice:** This setting is only available in the library project. **Note:** After you build a static library, the tool will automatically place the O object file to A file, archive that represents the library file. The A library file will appear under the **Archives** node in the project structure in the **Project Explore** view. By default, this field is set to: **ar**
Hex/Bin converter	Name of the tool to copy and translate object files. The GNU **objcopy** tools copies the contents of an object file to another. **Note:** This setting is only available in the application project. **Note:** This tool is also used by **Standard S32DS Create Flash Image** virtual tool. Default: **objcopy**
Listing generator	Name of the tool to display information about object files. The GNU **objdump** tool is used by Standard S32DS Disassembler. **Notice:** This setting is only available in the application project. **Note:** This tool is also used by **Standard S32DS Create Listing** virtual tool. Default: **objdump**
Size command	Name of the tool that calculates the size of the resulting ELF file. **Notice:** This setting is only available in the application project. Their GNU **size** tool calculates the size, in bytes, of the text, data and uninitialized sections in the ELF file, and their total. Default: **size**
Build command	Name of the tool that automatically builds executable programs, and libraries from source code in the project. Default: **make**
Remove command	Name of the tool and command parameters to remove built executable programs, libraries, and object files. Default: **rm -rf**
Create flash image	Select to create flash image using defined **Hex/Bin converter** tool. **Notice:** This setting is only available in the application project. As a result of selecting this check box, the tool creates the `<project_name>.srec` file at build time. To view this file, in the **Project Explorer** view expand `Debug`, and then double-click the SREC file. The file contains the binary code in Motorola SREC text format that represents binary data as a hexadecimal text in ASCII format. You can use this file to flash it to the target processor. Selecting this check box shows the **Standard S32DS Create Flash Image** tool in the tree on the **Tool Settings** tab.

Option	Description
	Note: By default, the flag enables S32DS ARM v2018.R1 to run `objcopy` and generate an S-record SREC file from the produced ELF file. Open the **Standard S32DS Create Flash Image** page on the **Tool Settings** tab to find out which command is used to invoke **objcopy**. By default this check box is cleared.
Create extended listing	Select to create listing using defined **Listing generator** tool. **Notice:** This setting is only available in the application project. As a result of selecting this check box, the tool creates the `<project_name>.lst` file at build time. To view this file, in the **Project Explorer** view expand `Debug`, and then double-click the LST file. The file contains the disassembly of the produced ELF file and gives slightly more details on functions than **Standard S32DS Disassembler**. Selecting this check box shows the **Standard S32DS Create Listing** tool in the tree on the **Tool Settings** tab. **Note:** By default, the flag enables S32DS ARM v2018.R1 to run `objdump` and disassemble the produced ELF file. Open the **Standard S32DS Create Listing** page on the **Tool Settings** tab to find out which command is used to invoke **objdump**. By default this check box is cleared.

Print size	Select to call the **size** tool after the project is built.
	Notice: This setting is only available in the application project.
	The tool outputs details to the **Console** pane. A sample output:
	`Executing target #7 Application.siz`
	`Invoking: Standard S32DS Print Size`
	`aarch64-none-elf-size --format=berkeley`
	` Application.elf`
	`   text    data     bss     dec     hex filename`
	`  11996    1904      64   13964    368c`
	` Application.elf`
	`Finished building: Application.siz`
	By default this check box is selected.

Target Processor

This section describes settings that can be configured on the **Cross Settings** page. The page is available in the properties of the project that you create with S32DS ARM v2018.R1. The settings relate to target SoC processor and its ARM® core that will be used to run the code produced by the built project.

Tip: Right-click the project in the **Project Explorer** view, and then select **Properties** to open project properties. Select **C/C++ Build** > **Settings**, and then, on the **Tools Settings** tab click **Target Processor** to show the settings of this page.

The **Target Processor** page allows you to make core customizations of the project that are specific to the target processor selected in the project creation wizard. Part of these settings you have configured when creating the project.

Note: Settings on this page depend on the project creation wizard you use and target processor that you select in the wizard.

Note: For all options that support the **Toolchain default** setting, this setting refers to *factory default setting* for GCC. Factory default setting is the value that is configured in GCC as the default for a specific option.

The following table describes the options available on the **Target Processor** page.

Table 10: Target Processor: Application Project and Library Project

Option	Description
Other target flags	Additional compiler options. Allows you to specify compiler options that are not available in project properties but are supported by compiler. Refer to compiler documentation for information about command line options supported by compiler. **Important:** Settings in lists override options specified in this field. For example, if you specify the `#mcpu=cortex-a53+nofp` command line option here while leaving **cortex-a53** in the **Target processor** field, the command line option will be overridden by the setting in **Target processor** field. To make the command line option to take precedence, select **Toolchain default** in **Target processor** field. By default, this field is empty.
ARM family	Name of the ARM® core family. Allows you to specify the ARM® core used on the target processor. **Note:** This setting is not available in projects targeted for processors based on ARM® Cortex-A53 core. Only the following options are supported: • **Toolchain default** Compiles your code by using the default set of options that were used to build the GCC. **Note:** If this option is selected, S32DS ARM v2018.R1 does not pass any core-specific options to the compiler. Selecting this option is not recommended as the default option set may change without further notice. • **cortex-m4** Compile the project with the option set specific to ARM® Cortex-M4F core. • **cortex-m0plus** Compile the project with the option set specific to ARM® Cortex-M0+ core. • **cortex-a5** Compile the project with the option set specific to ARM® Cortex-A5 core. **Important:** Other cores may be available in the list for your selection. However, these cores are not supported by the project creation wizard. If you select an unsupported core, you will have to manually recreate project structure, startup code and other metadata in the project. By default, the core specified in the project creation wizard is selected. For example, if this project is created for the Cortex-M4F core, **cortex-m4** will be selected.
Architecture	Target ARM® architecture. The compiler can take advantage of the extra instructions that the selected architecture provides and optimize the code to run on a specific core. The inline assembler might display error messages or warnings if it assembles some core-specific instructions for the wrong target architecture. Following options are available: • **Toolchain default**

Option	Description
	Compiles your code by using the default architecture option that was used to build the GCC. • **armv6-m** Corresponds to the `#march=armv6-m` command line option in GCC. Selecting this option is the same as selecting **Toolchain default** here and specifying the `#march=armv6-m` command line option in the **Other target flags** field. • **armv7e-m** Corresponds to the `#march=armv7e-m` command line option in GCC. Selecting this option is the same as selecting **Toolchain default** here and specifying the `#march=armv7e-m` command line option in the **Other target flags** field. • **armv7-a** Corresponds to the `#march=armv7-a` command line option in GCC. Selecting this option is the same as selecting **Toolchain default** here and specifying the `#march=armv7-a` command line option in the **Other target flags** field. • **armv8-a** Corresponds to the `#march=armv8-a` command line option in GCC. Selecting this option is the same as selecting **Toolchain default** here and specifying the `#march=armv8-a` command line option in the **Other target flags** field. **Note:** This setting is only available in projects targeted for processors based on ARM® Cortex-A53 core. By default, **Toolchain default** is selected.
Target processor	A host processor where you will run the code. **Note:** This setting is only available in projects targeted for processors based on ARM® Cortex-A53 core. Following options available: • **Toolchain default** Compiles your code by using the default core option that was used to build the GCC. • **cortex-a53** Compiles your code for Cortex-A53 cores with hardware support for floating-point instructions that will be performed by integrated FPU. Corresponds to the `#mcpu=cortex-a53` command line option in GCC. Selecting this option is the same as selecting **Toolchain default** here and specifying the `#mcpu=cortex-a53` command line option in the **Other target flags** field. By default, **cortex-a53** is selected.
Optimize	Target host processor for which you want to run code optimizations. **Note:** This setting is only available in projects targeted for processors based on ARM® Cortex-A53 core. Following options available: • **Toolchain default** Use the factory default optimization that was configured in GCC as the default for this option. • **cortex-a53**

Option	Description
	Use optimization for the ARM® Cortex-A53 core. By default, **cortex-a53** is selected.
Instruction set	Assembler instruction set for target architecture. Allows you to select the target instruction set for generating code that executes in ARM or Thumb states. **Note:** This setting is not available in projects targeted for processors based on ARM® Cortex-A53 core. Following options are available: • **Toolchain default** Compiles your code by using the default instruction set that was used to build the GCC. • **Thumb (-mthumb)** Forces GCC to generate code for Thumb state. • **ARM (-marm)** Forces GCC to generate code for ARM state You can also override the ARM and Thumb mode for each function by using the **target("thumb")** and **target("arm")** function attributes or pragmas. The default can be changed by configuring GCC with the `-- with-mode={arm,thumb}` option. For example, `--with-mode=thumb` makes the compiler default to generating Thumb code if neither `-marm` nor `-mthumb` are specified. By default, **Toolchain default** is selected.
Thumb interwork (#mthumb-interwork)	Enables assembling the code for interworking between ARM and Thumb states. **Note:** This setting is not available in projects targeted for processors based on ARM® Cortex-A53 core. Select this check box if you write ARM code that you want to interwork with Thumb code or vice versa. This will enable GCC to generate code that supports calls between the ARM and Thumb instruction sets. Linker will add support for interworking veneers that handle the change of instruction set between the ARM and Thumb states. The only functions that need to be compiled for interworking are the functions that are called from the other state. Corresponds to the `-mthumb-interwork` command line option in GCC. By default, this check box is cleared.
Endianness	Order mode of in which bytes are arranged on the target processor. Allows you to generate code for a processor running in little-endian mode by strictly specifying the byte order of the target hardware architecture. **Note:** This setting is not available in projects targeted for processors based on ARM® Cortex-A53 core. Following options are available: • **Toolchain default** By default, the GCC is configured to generate code for a processor running in little-endian mode.

Option	Description
	• **Little endian (-mlittle-endian)** Forces GCC to generate code for target processor using little-endian (LE) byte order (the byte that stores the most significant bit is stored or sent last). Corresponds to the `-mlittle-endian` command line option in GCC. By default, **Toolchain default** is selected.
Float ABI	Floating-point application binary interface ABI. Allows you to specify, which floating-point binary interface ABI the GCC should use when compiling your code. **Note:** This setting is not available in projects targeted for processors based on ARM® Cortex-A53 core. **Note:** Hard-float and soft-float ABIs are not link-compatible; you must compile your entire program with the same ABI, and link with a compatible set of libraries. Following options are available: • **Toolchain default** Compiles your code by using the default binary interface for the **Target Processor** as is specified in the GCC. GCC will detect floating-point operations support based on the support for FPU in the selected processor. On processors based on Cortex-A53 core this defaults to **-mfloat-abi=hard** so that the core is responsible for floating-point operations and calling conventions specific to FPU are used. • **Library (soft)** Enables GCC to generate code with library calls so that floating-point operations are emulated by compiler and not the FPU on the processor. Corresponds to the `-mfloat-abi=soft` command line option in GCC. • **Library with FP (softfp)** Enables GCC to generate code with support for hardware floating# point instructions provided by processor while using soft floating-point ABI calling conventions. Corresponds to the `-mfloat-abi=softfp` command line option in GCC. • **FP instructions (hard)** Enables GCC to generate code with support for hardware floating# point instructions provided by processor, and uses ABI calling convention specific to FPU on the processor. Corresponds to the `-mfloat-abi=hard` command line option in GCC. • **FPU SP only** Enables GCC to generate code with support for hardware floating# point instructions provided by processor while using soft floating-point ABI calling conventions with casting double-precision operations to floating operations (`float` size is used for variables defined as `double`). Corresponds to `-mfloat-abi=softfp -fshort-double` command line options in GCC.  **Warning:** If you select this option, your code will be binary compatible only with the code generated with the same option. For example, if you plan to use some library that was compiled with another option, you may face an issue with calling a library function that expects an argument of type `double`. Your code will cast the value of this argument to `float`, and function will fail to return. To avoid such issues, if you use the **FPU SP only** compiler option in your application, make sure to recompile libraries with **FPU SP only**. By default, **Toolchain default** is selected.

Option	Description
FPU Type	Type of the floating-point unit (FPU) for hardware emulation of floating-point operations on the target processor. Specifies what floating-point (FP) hardware is available on the target processor. **Note:** This setting is not available in projects targeted for processors based on ARM® Cortex-A53 core. **Important:** This setting is only available if hardware or hardware emulated ABI option is selected in **Float ABI** (anything except for **Library (soft)**). Following options are available: • **Toolchain default** Enables GCC to select the floating-point instructions based on the settings specified in **Architecture** and **Target processor**. • **fpv4-sp-d16** This architecture includes support for single-precision FPv4 instructions with registers that can be used by your application as 32 single-precision floating point registers or as 16 double-precision floating point registers. Corresponds to the `mfpu=fpv4-sp-d16` command line option in GCC. • **vfpv3-d16** This architecture includes support for VFPv3 FP instructions with FP registers that can be used by your application as 16 double-precision floating point registers. Corresponds to the `mfpu=vfpv3-d16` command line option in GCC. • **fpv5-d16** This architecture includes support for single-precision FPv5 instructions with 16 double-precision floating point registers. Corresponds to the `mfpu=fpv5-d16` command line option in GCC. • **fpv5-sp-d16** This architecture includes support for FP registers that can be used by your application as 32 single-precision floating point registers or as 16 double-precision floating point registers. Corresponds to the `mfpu=fpv5-sp-d16` command line option in GCC. By default, **Toolchain default** is selected.
Unaligned access	Controls unaligned access. Allows you to enable or disable reading and writing of 16-bit and 32-bit values from addresses that are not aligned to 16- or 32-bits. If unaligned access is not enabled, words in packed data structures are accessed a byte at a time. **Note:** This setting is not available in projects targeted for processors based on ARM® Cortex-A53 core. By default unaligned access is enabled for enabled all ARM® architectures, except for: all pre-ARMv6, all ARMv6-M, and all ARMv8-M Baseline architectures. The options available are: • **Toolchain default** • **Enabled (-munaligned-access)** Enables unaligned access. Corresponds to the `-munaligned-access` command line option in GCC. • **Disabled (-mno-unaligned-access)**

Option	Description
	Enables unaligned access. Corresponds to the `-mno-unaligned-access` command line option in GCC. By default, **Toolchain default** is selected.
Libraries support	Library to be linked to the application. Allows you to specify support of standard C/C++ library, and configure I/O for your application. **Note:** This setting is not available in projects targeted for processors based on ARM® Cortex-A53 core. Following options are available: • **none** Do not link standard C/C++ library and disable support for console I/O. **Note:** Using this option is not recommended as no C/C++ libraries will be used in your application and no I/O interface will be supported. • **newlib_nano no I/O** Link lightweight version of the NewLib and use a no-operation stub for console I/O (disable semihosting). Corresponds to enabling `-specs=nano.specs` (links newlib_nano) and `-specs=nosys.specs` (stubs output by replacing system read/write functions and disabling semihosting) command line options for GCC linker. • **newlib_nano Debugger Console I/O** Link lightweight version of the NewLib and use ARM® semihosting. Corresponds to enabling `-specs=nano.specs` (links newlib_nano) and `-specs=rdimon.specs` (enables semihosting) command line options for GCC linker. • **newlib no I/O** Link standard version of the NewLib with system C/C++ functions and use a no-operation stub for console I/O (disable semihosting). Corresponds to enabling `-specs=nosys.specs` (stubs output by replacing system read/write functions and disabling semihosting) command line options for GCC linker. • **newlib Debugger Console I/O** Link standard version of the NewLib with system C/C++ functions and use ARM® semihosting. Corresponds to enabling `-specs=rdimon.specs` (enables semihosting) command line options for GCC linker. • **ewl_c no I/O** Link standard version of the EWL with system C functions and use a no-operation stub for console I/O (disable semihosting). Corresponds to enabling `-specs=ewl_c9x_noio.specs` (stubs output by replacing system read/write functions and disabling semihosting) command line options for GCC linker. • **ewl_c++ no I/O** Link standard version of the EWL with system C++ functions and use a no-operation stub for console I/O (disable semihosting).

Option	Description
	Corresponds to enabling `-specs=ewl_c9x_c++_noio.specs` (stubs output by replacing system read/write functions and disabling semihosting) command line options for GCC linker. • **ewl_c Debugger Console I/O** Link standard version of the EWL with system C functions and use ARM® semihosting. Corresponds to enabling `-specs=ewl_c9x_hosted.specs` (enables semihosting) command line options for GCC linker. • **ewl_c++ Debugger Console I/O** Link standard version of the EWL with system C++ functions and use ARM® semihosting. Corresponds to enabling `-specs=ewl_c9x_c++_hosted.specs` (enables semihosting) command line options for GCC linker. • **ewl_nano_c no I/O** Link lightweight version of the EWL with system C/C++ functions and use a no-operation stub for console I/O (disable semihosting). **Note:** EWL Nano is a lightweight version of the EWL library that contains optimized versions of some functions such as, for example, the `printf()` and `scanf()` functions. Optimizations are made in regards to reduction in the size of functions, and hence the overall size of EWL libraries, which is achieved by reducing functionality rarely used in embedded software for MCU. Corresponds to enabling `-specs=ewl_c_noio.specs` (stubs output by replacing system read/write functions and disabling semihosting) command line options for GCC linker. • **ewl_nano_c++ no I/O** Link lightweight version of the EWL with system C++ functions and use a no-operation stub for console I/O (disable semihosting). Corresponds to enabling `-specs=ewl_c++_noio.specs` (stubs output by replacing system read/write functions and disabling semihosting) command line options for GCC linker. • **ewl_nano_c Debugger Console I/O** Link lightweight version of the EWL with system C functions and use ARM® semihosting. Corresponds to enabling `-specs=ewl_c_hosted.specs` (enables semihosting) command line options for GCC linker. • **ewl_nano_c++ Debugger Console I/O** Link lightweight version of the EWL with system C++ functions and use ARM® semihosting. Corresponds to enabling `-specs=ewl_c++_hosted.specs` (enables semihosting) command line options for GCC linker. **Note:** Projects targeted for Cortex-A53 core automatically default to NewLib library without I/O redirection, and **newlib no I/O** is selected in this option for projects created for target processors on this core. Other libraries are not supported for this core, but you can specify that S32DS ARM v2018.R1 should use debugger console as a standard output.

Option	Description

	Disambiguation of options. Options are listed in the following format:
	`<standard library> <I/O mode>`
	where:
	• *standard library* — libraries to be linked to the project:·
	  • **newlib**
	  An implementation of `libc` standard C/C++ library of system functions.
	  • **newlib_nano**
	  A lightweight version of newlib library with reduced set of system functions.
	  • **ewl**
	  An implementation of Embedded Warrior Library `EWL` C/C++ library that extends ISO/IEC standard libraries and provides for a smaller footprint than newlib.
	  • **ewl_nano**
	  A lightweight version of ewl library with reduced set of functions.
	• *I/O mode* # I/O modes used for project:
	  • **no I/O**
	  Use a no-operation stub for console I/O (disable semihosting).
	  Note: Console output will be stubbed. If you want to redirect the output from your application, you will have to implement the `_write()` function in your application and then use UART interface to transfer characters. Otherwise, use **newlib_nano Debugger Console I/O**.
	  • **Debugger Console**
	  Use ARM® semihosting though the RDIMON interface for console I/O provided by `printf()` or `puts()`.
	  Note: Semihosting allows to route characters output by console functions such as `printf()` though the debugger connection. Console output will be redirected to RDIMON that provides its implementation of `_write()` function.
	If you are using the `new` operator in your C++ application and want to switch the library that you link your code with, depending on the library, do one of the following:
	• If you switch from EWL to Newlib
	  In the **Project Explorer** view, navigate to `Project_Settings\Linker_Files`, and comment-out the line with `EXTERN` keyword in the LD linker file:
	  `/* EXTERN(_ZN10__cxxabiv119__terminate_handlerE */` line.
	• If you switch from Newlib to EWL
	  In the **Project Explorer** view, navigate to `Project_Settings\Linker_Files`, and remove the commentary from the line with `EXTERN` keyword in the LD linker file:
	  `EXTERN(_ZN10__cxxabiv119__terminate_handlerE` line.
	By default, the following option is selected:
	• For application projects: **ewl_c no I/O (-specs=ewl+c9x_noio.specs)**
	• For library projects: **none**

Sysroot	Logical root location of headers and libraries.

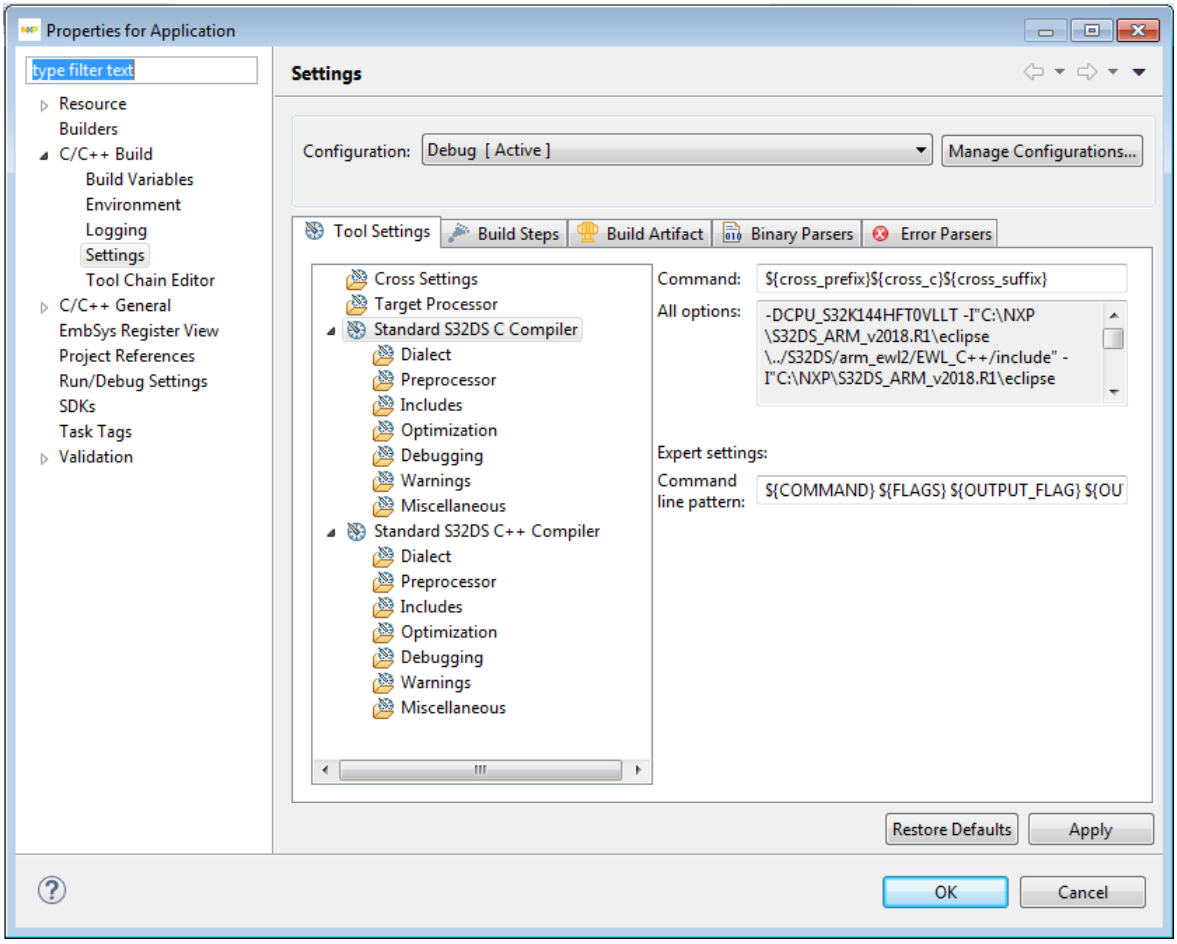
Option	Description
	Note: This setting is not available in projects targeted for processors based on ARM® Cortex-A53 core. By default, this field is set to: • `--sysroot="${S32DS_ARM32_EWL_DIR}"` For projects created for GCC 6.3 and EWL • `--sysroot="${ARM_EWL_DIR}"` For projects created for GCC 4.9 and EWL • `--sysroot="${S32DS_ARM32_NEWLIB_DIR}"` For projects created for GCC 6.3 and Newlib • `--sysroot="${ARM_NEWLIB_DIR}/newlib"` For projects created for GCC 4.9 and Newlib

Standard S32DS C Compiler and Standard S32DS C++ Compiler

This section describes settings that can be configured on **Standard S32DS C Compiler** and **Standard S32DS C++ Compiler** pages. The pages are available in the properties of the project that you create with S32DS ARM v2018.R1. The pages allow you to configure C or C++ compiler options for your application or library project.

Tip: Right-click the project in the **Project Explorer** view, and then select **Properties** to open project properties. Select **C/C++ Build** > **Settings**, and then, on the **Tools Settings** tab click **Standard S32DS C Compiler** or **Standard S32DS C++ Compiler** to show the settings of the pages.

Important: Availability of the **Standard S32DS C++ Compiler** page depends on the language that you have specified in project creation wizard when making core customizations. This page is only available if you have selected C++ in the **Language** list within the project creation wizard to set up your project for ANSI C++ startup code. The **Standard S32DS C Compiler** page is available for both C and C++ languages.



The following table describes the options available on the **Standard S32DS C Compiler** and **Standard S32DS C++ Compiler** pages.

Table 11: Standard S32DS C Compiler/Standard S32DS C++ Compiler: Application Project and Library Project

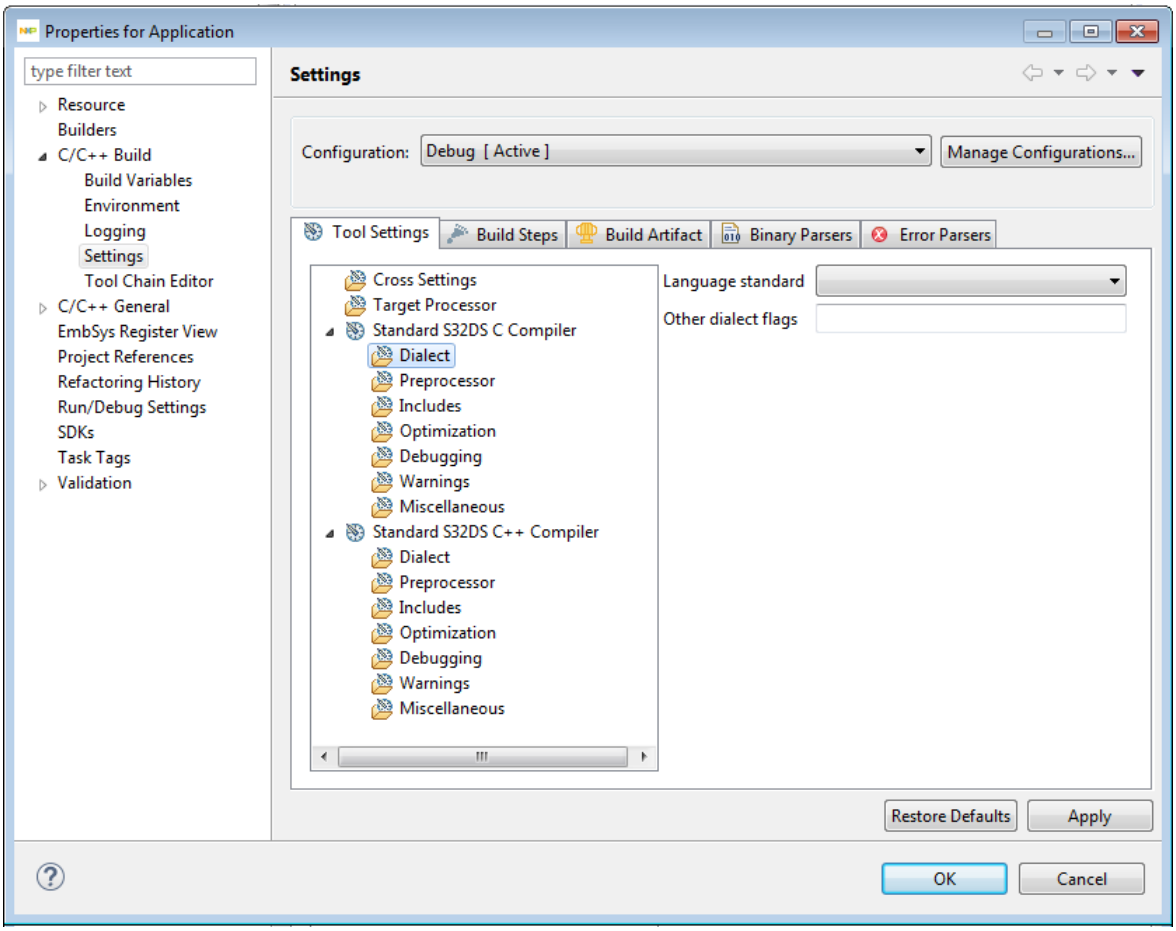
Option	Description
Command	Command pattern for `${COMMAND}` variable. Allows you to specify the command that will be used by S32DS ARM v2018.R1 to run the compiler. You can use system variables specified in toolchain settings on the **Cross Settings** page. For example, `${cross_prefix}` will expand to the value specified in the **Prefix** field on this page. **Note:** Compiler uses same tools as the compiler and thus the commands in the command line reflect those found on **Standard S32DS C Preprocessor** and **Standard S32DS C++ Preprocessor** pages. The difference is in command line options used to run the tools. By default, this field is set to: • **Standard S32DS C Compiler** default: `${cross_prefix}${cross_c}${cross_suffix}` • **Standard S32DS C++ Compiler** default: `${cross_prefix}${cross_cpp}${cross_suffix}`

Option	Description
All options	Actual expanded command line options the compiler will be called with. Shows the expanded command options specified on the pages nested under the **Standard S32DS C Compiler** or the **Standard S32DS C++ Compiler** page. This is the exact set of options that S32DS ARM v2018.R1 will use during the compilation stage of the build process. The default value depends on the following settings specified in the project creation wizard: • Toolchain • Processor • Target Language
Command line pattern	Command line to call the compiler. By default, this field is set to: `${COMMAND} ${FLAGS} ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}`

Dialect

This section describes settings that can be configured on the **Dialect** page. The page is available in the properties of the project that you create with S32DS ARM v2018.R1. The page allows you to configure the compiler to verify your code against the specific standard of C or C++ language when compiling your application or library project.

Tip: Right-click the project in the **Project Explorer** view, and then select **Properties** to open project properties. Select **C/C++ Build** > **Settings**, and then, on the **Tools Settings** tab expand **Standard S32DS C Compiler** or **Standard S32DS C++ Compiler**, and then click **Dialect** to show the settings of the page.



The following table describes the options available on the **Dialect** page.

Table 12: Dialect: Application Project and Library Project

Option	Description
Language standard	Standard of C (for **Standard S32DS C Compiler**) and C++ (for **Standard S32DS C++ Compiler**) language. Allows you to strictly set C language standard to which the GCC compiler should conform. **Note:** More information can be found on the GCC site at gcc.gnu.org. The compiler can accept several base standards, such as ‘c90’ or ‘c++98’. When a base standard is specified, the compiler accepts all programs following that standard plus those using GNU extensions that do not contradict it. For example, **-std=c90** turns off certain features of GCC that are incompatible with ISO C90, such as the **asm** and **typeof** keywords, but not other GNU extensions that do not have a meaning in ISO C90, such as omitting the middle term of a **?:** expression. On the other hand, when a GNU dialect of a standard is specified, all features supported by the compiler are enabled, even when those features change the meaning of the base standard. As a result, some strict-conforming programs may be rejected. The particular standard is used by **-Wpedantic** warning option to identify which features are GNU extensions given that version of the standard. For example **-std=gnu90 -Wpedantic** warns about C++ style ‘//’ comments, while **-std=gnu99 -Wpedantic** does not. Following options are available for Standard S32DS C Compiler:

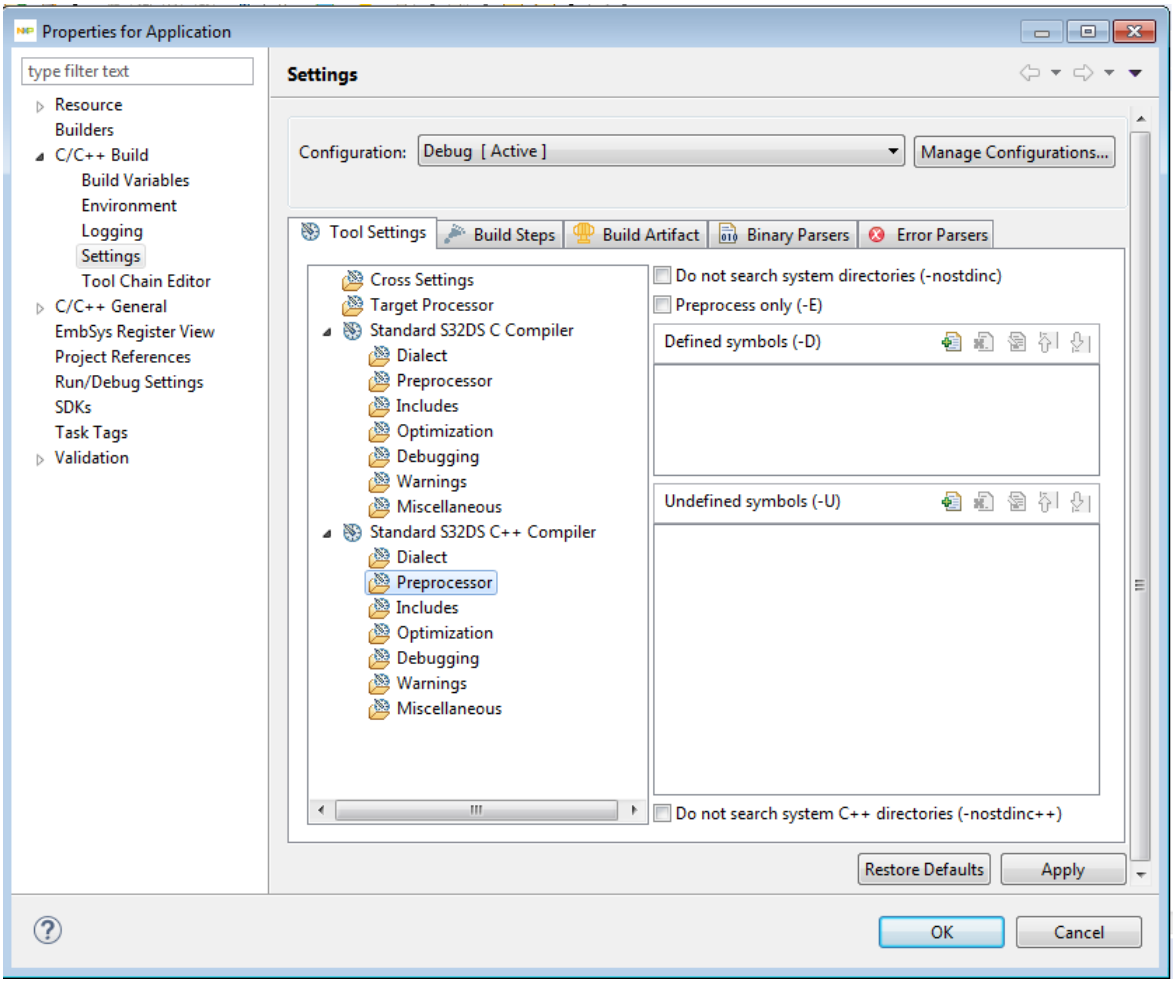
Option	Description
	• **ISO C90 / ANSI C89 (-std=c90)** Supports all ISO C90 programs (certain GNU extensions that conflict with ISO C90 are disabled). Select to compile code written in ANSI standard C. The compiler does not enforce strict standards. For example, code can contain some minor extensions, such as C++ style comments (//), and **$** characters in identifiers. • **ISO C99 (-std=c99)** Instructs the compiler to enforce stricter adherence to the ANSI/ISO standard. This standard is substantially completely supported, modulo bugs and floating-point issues (mainly but not entirely relating to optional C99 features from Annexes F and G). • **ISO C11 (-std=c11)** This is the 2011 revision of the ISO C standard. This standard is substantially completely supported, modulo bugs, floating-point issues (mainly but not entirely relating to optional C11 features from Annexes F and G) and the optional Annexes K (Bounds-checking interfaces) and L (Analyzability). Following options are available for Standard S32DS C++ Compiler:: • **ISO C++98 (-std=c++98)** The 1998 ISO C++ standard plus the 2003 technical corrigendum and some additional defect reports. Same as -ansi for C++ code. • **ISO C++11 (-std=c++0x)** The 2011 ISO C++ standard plus amendments • **ISO C++1y (-std=c++1y)** The 2014 ISO C++ standard plus amendments. By default, no option is selected and factory default standard for GCC is used (the one specified as the default setting in the toolchain).
Other dialect flags	Additional command line options. Allows you to specify dialect flags supported by GCC and not otherwise available on this page. **Note:** More information about supported flags can be found on the GCC site at gcc.gnu.org. By default, this field is empty.

Preprocessor

This section describes settings that can be configured on the **Preprocessor** page. The page is available in the properties of the project that you create with S32DS ARM v2018.R1. The page allows you to configure the C or C++ preprocessor, which is run on each C/CPP source file before actual compilation.

If you want to preprocess your code, you can opt from configuring the preprocessor options here, or going to the configuration of the preprocessor tool by selecting the **Settings** page on the **Preprocessor** page under **Standard S32DS C Preprocessor** or under **Standard S32DS C++ Preprocessor**. Options on this page control preprocessor settings that are used by compiler when you build your project.

Tip: Right-click the project in the **Project Explorer** view, and then select **Properties** to open project properties. Select **C/C++ Build** > **Settings**, and then, on the **Tools Settings** tab expand **Standard S32DS C Compiler** or **Standard S32DS C++ Compiler**, and then click **Preprocessor** to show the settings of the page.



The following table describes the options available on the **Preprocessor** page.

Table 13: Preprocessor: Application Project and Library Project

Option	Description
Do not search system directories (-nostdinc)	Skip standard system locations of header files. Select the check box if you do not want the compiler to search the system locations for header files. Only the locations specified on the Includes page will be used to search for the header files. If this check box is not selected, the compiler performs a full search that includes the system locations. By default, this check box is cleared.
Preprocess only (-E)	Select the check box if you only want to preprocess source files without running the compiler. Nothing is done except preprocessing. **Attention:** Selecting this will cause linker to error when you build the project. This is because linker expects an object file to be available. However, because no compilation is done when this option is enabled, compiler does not create an object file. By default, this check box is cleared.
Defined symbols (-D)	Predefined name as a macro.

Option	Description
	Allows you to specify the substitution strings that the compiler applies to all the assembly-language modules in the build target. Enter just the string portion of a substitution string. The S32DS ARM v2018.R1 prepends the **-D** token to each string that you enter. For example, entering **opt1 x** produces this result on the command line: **-Dopt1 x**. This option is similar to the **#define** directive, but applies to all assembly-language modules in a build target. Following buttons are available that allow you to define names: • **Add** • **Delete** • **Edit** • **Move up** • **Move down** **Note:** Added names are processed in the top-down order. The value depends on the following settings specified in the project creation wizard: • Processor
Undefined symbols (-U)	Remove definition for a previously defined name. Allows you to cancel any previous definition of name, either built-in or provided with a **-D** option By default this list is empty.
Do not search system C++ directories (#nostdinc++)	Select this check box if you do not want the C++ compiler to search the system directories for header files. The C++ compiler performs a full search that includes the system directories. **Note:** This option is only available in projects configured for C++ language. By default this check box is cleared.

Includes

This section describes how to specify the directories paths and files paths in the build properties for C/C++ S32DS project. The table below lists and describes the various options available on the **Includes** panel.

Table 14: Tool Settings - Standard S32DS C Compiler/ Standard S32DS C++ Compiler - Includes

	Option	Description
1.	Include paths (-I)	This option adds the directory to the list of directories to be searched for header files. Directories named by -I are searched before the standard system include directories. If the directory is a standard system include directory, the option is ignored to ensure that the default search order for system directories and the special treatment of system headers are not defeated. This option changes the build target's search order of access paths to start with the system paths list. The compiler can search **#include** files in several different ways. You can also set the search order as follows: For include statements of the form **#include"xyz"**, the compiler first searches user paths, then the system paths.

	Option	Description
		For include statements of the form **#include<xyz>**, the compiler searches only system paths. This option is global. By default, this option is set to `"${ProjDirPath}/include"`
2.	Include files (-include)	Use this option to specify the include file search path. Process file as if **#include "file"**appeared as the first line of the primary source file. However, the first directory searched for file is the preprocessor's working directory instead of the directory containing the main source file. If not found there, it is searched for in the remainder of the **#include** "..." search chain as normal. If multiple **-include** options are given, the files are included in the order they appear on the command line. By default this checkbox is clear.

Optimization

This section describes how to control C/C++ compiler optimizations in the build properties for S32DS project. Compiler optimization can be applied in either global or non-global optimization mode. You can apply global optimization at the end of the development cycle, after compiling and optimizing all source files individually or in groups.

Without any optimization option, the C/C++ compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you expect from the source code.

Turning on optimization flags makes the C/C++ compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

The C/C++ compiler performs optimization based on the knowledge it has of the program. Compiling multiple files at once to a single output file mode allows the compiler to use information gained from all of the files when compiling each of them.

The table below lists and describes the various options available on the **Optimization** panel.

Table 15: Tool Settings - Standard S32DS C Compiler/ Standard S32DS C++ Compiler - Optimization

	Option	Description
1.	Optimization level	Specify the optimizations that you want the C/C++ compiler to apply to the generated object code: • **None (-O0)** - Disable optimizations. This setting is equivalent to specifying the **-O0** command-line option. The compiler generates unoptimized, linear assembly-language code, reduce compilation time. • **Optimize (-O1)** - The compiler performs all target-independent (that is, non-parallelized) optimizations, such as function inlining. Optimizing compilation takes somewhat more time, and a lot more memory for a large function. This setting is equivalent to specifying the **-O1** command-line option. The compiler omits all target-specific optimizations and generates linear assembly-language code. • **Optimize more (-O2)** - The compiler performs all optimizations (both target-independent and target-specific). GCC performs nearly all supported optimiza-tions that do not involve a space-speed tradeoff. As compared to **-O**, this option increases both compila-tion time and the performance of the genera-ted code. This setting is equivalent

	Option	Description
		to specifying the **-O2** com-mand-line option. The compiler outputs optimized, non-linear, parallelized assembly-language code. • **Optimize most (-O3)** - The compiler performs all the level 2 optimizations, then the low-level optimizer performs global-algorithm register allocation. This setting is equivalent to specifying the **-O3** command-line option. At this optimization level, the compiler generates code that is usually faster than the code generated from level 2 optimizations. • **Optimize size (-Os)** - The compiler performs further optimizations designed to reduce code size. **-Os** enables all **-O2** optimizations that do not typically increase code size. The resulting binary file has a smaller executable code size, as opposed to a faster execution speed. This setting is equivalent to specifying the **-Os** command-line option. **Note:** Default value of optimization level depends not only on com-pilers (C or C++) but on configuration (Debug or Release) as well. Standard S32DS C/C++ Compiler defaults: • for debug configuration: **None (-O0)** • for release configuration: **Optimize most (-O3)**
2.	Other optimization flags	Specifies additional command line options and individual optimization flag that can be turned ON/OFF based on the user requirements. Type in custom optimization flags that are not otherwise available in the UI. By default this checkbox is clear.
3.	'char' is signed (-fsigned-char) (Standard S32DS C Compiler **only**)	Check to treat char declarations as signed char declarations. This setting is equivalent to **-fsigned-char** command-line option. By default this checkbox is clear.
4.	Function sections (-ffunction-sections)	Check to use function sections. Place each function into its own section in the output file if the target supports arbitrary sections. The name of the function determines the section's name in the output file. Use these options on systems where the linker can perform optimizations to improve locality of reference in the instruction space. Most systems using the ELF object format have linkers with such optimizations. Only use these options when there are significant benefits from doing so. This setting is equivalent to **-ffunction-sections** command-line option. By default, this check box is selected.
5.	Data sections (-fdata-sections)	Check to use short data sections. Place each data item into its own section in the output file if the target supports arbitrary sections. The name of the data item determines the section's name in the output file. Use these options on systems where the linker can perform optimizations to improve locality of reference in the instruction space. Most systems using the ELF object format have linkers with such optimizations.

	Option	Description
		Only use these options when there are significant benefits from doing so. When you specify these options, the assembler and linker create larger object and executable files and are also slower. You cannot use **gprof** on all systems if you specify this option, and you may have problems with debugging if you specify both this option and **-g**. This setting is equivalent to **-ffunction-sections** command-line option. By default, this check box is selected.
6.	No common uninitialized (-fno-common)	In C code, controls the placement of uninitialized global variables. Unix C compilers have traditionally permitted multiple definitions of such variables in different compilation units by placing the variables in a common block. This is the behavior specified by **-fcommon**, and is the default for GCC on most targets. On the other hand, this behavior is not required by ISO C, and on some targets may carry a speed or code size penalty on variable references. The **-fno-common** option specifies that the compiler should place uninitialized global variables in the data section of the object file, rather than generating them as common blocks. This has the effect that if the same variable is declared (without extern) in two different compilations, you get a multiple-definition error when you link them. In this case, you must compile with **-fcommon** instead. Compiling with **-fno-common** is useful on targets for which it provides better performance, or if you wish to verify that the program will work on other systems that always treat uninitialized variable declarations this way. This setting is equivalent to **-fno-common** command-line option. By default this checkbox is clear.
7.	Do not inline functions (-fno-inline-functions)	Do not consider any functions for inlining, even if they are not declared inline. This setting is equivalent to **-fno-inline-functions** command-line option. By default this checkbox is clear.
8.	Assume freestanding environment (-ffreestanding)	Assert that compilation targets a freestanding environment. This implies **-fno-builtin**. A freestanding environment is one in which the standard library may not exist, and program startup may not necessarily be at main. The most obvious example is an OS kernel. This is equivalent to **-fno-hosted**. By default this checkbox is clear.
9.	Disable builtin (-fno-builtin)	Don't recognize built-in functions that do not begin with **__builtin_** as prefix. This setting is equivalent to **-fno-builtin** command-line option. By default this checkbox is clear.
10.	Single precision constants (-fsingle-precision-constant)	Check to enable single precision constants. Treat floating-point constants as single precision instead of implicitly converting them to double-precision constants. This setting is equivalent to **-single-precision-constant** command-line option.

	Option	Description
		By default this checkbox is clear.
11.	Link-time optimizer (-flto)	Run the standard link-time optimizer. When invoked with source code, it generates GIMPLE (one of GCC's internal representations) and writes it to special ELF sections in the object file. When the object files are linked together, all the function bodies are read from these ELF sections and instantiated as if they had been part of the same translation unit. This setting is equivalent to **-flto** command-line option. By default this checkbox is clear.
12.	Disable loop invariant move (-fno-move-loop-invariants)	Disable the loop invariant motion pass in the RTL loop optimizer. Enabled at level **-O1**. This setting is equivalent to **-fno-move-loop-invariants** command-line option. By default this checkbox is clear.

Debugging

This section describes how to control debugging in the build properties for the C/C++ S32DS project.

The table below lists and describes the various options available on the **Debugging** panel.

Table 16: Tool Settings - Standard S32DS C Compiler/ Standard S32DS C++ Compiler - Debugging

	Option	Description
1.	Debug Level	Specify the debug levels: • **None** - Level 0 produces no debug information at all, • **Minimal (-g1)** - Level 1 produces minimal information, enough for making backtraces in parts of the program that you don't plan to debug. This includes descriptions of functions and external variables, and line number tables, but no information about local variables, • **Default (-g)** - The default level is 2. The compiler generates DWARF 1.x conforming debugging information, • **Maximum (-g3)** - Level 3 includes extra information, such as all the macro definitions present in the program. So the compiler provides maximum debugging support. Default: **Maximum (-g3)**
2.	Other debugging flags	Specify additional command line options; type in custom debugging flags that are not otherwise available in the UI. By default, this field is clear.
3.	Generate gcov information (-ftest-coverage -fprofile-arcs)	Select to enable Profile Code Coverage in your application. Remember to enable this option in both the **Compiler** > **Miscellaneous** and **Linker** > **General**. Then rebuild your project and run Code Coverage again. By default, this field is clear.
4.	Debug format	Specify the debug formats for the compiler. Use it if you want to control for certain whether to generate the extra information.

	Option	Description
		• **Toolchain default**, • **Gdb** • **stabs** - Generates STABS-conforming debugging information, • **stabs+**, • **dwarf-2** - Generates DWARF 2.x-conforming debugging information, • **dwarf-3** - Generates DWARF 3.x-conforming debugging information, • **dwarf-4** - Generates DWARF 4.x-conforming debugging information. Default: **Toolchain default.**

Note:

Only **Other debugging flags** field is available. If a project contains some debug flags set in **Other options** then these options will present in build system but not available for changing or removing. The user shall edit .cproject file to move this flags to the **Other debugging flags** field.

For example, the line in **.cproject** file:

```
<option id="com.freescale.s32ds.cross.gnu.tool.c.compiler.option.debugging.other.842367527"
superClass="com.freescale.s32ds.cross.gnu.tool.c.compiler.option.debugging.other"
value="test_opt"
valueType="string"/>
```

should be changed to

```
<option id="gnu.c.compiler.option.debugging.other.1735145584"
superClass="gnu.c.compiler.option.debugging.other"
value="test_opt"
valueType="string"/>
```

The options will be moved to the **Other debugging flags** field.

Warnings

This section describes how to control C/C++ compiler warnings in the build properties for S32DS project.

The table below lists and describes the various options available on the **Warnings** panel.

Table 17: Tool Settings - Standard S32DS C Compiler/ Standard S32DS C++ Compiler - Warnings

	Option	Description
1.	Check syntax only (-fsyntax-only)	Check to check the code for syntax errors, but do not do anything beyond that. By default, this field is clear.
2.	Pedantic (-pedantic)	Check to issue all the warnings demanded by strict ISO C and ISO C ++; reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++. For ISO C, follows the version of the ISO C standard specified by any **-std** option used. By default, this field is clear.
3.	Pedantic warnings as errors (-pedantic-errors)	Like pedantic, except that errors are produced rather than warnings. By default, this field is clear.

	Option	Description
4.	Inhibit all warnings (-w)	Check to enable all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros. By default, this field is clear.
5.	All warnings (-Wall)	Check to enable all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros. This also enables some language-specific warnings. Default: on
6.	Extra warnings (-Wextra)	Check to enable some extra warning flags that are not enabled by **-Wall**. By default, this field is clear.
7.	Warnings as errors (-Werror)	Check to make all warnings into hard errors. Source code which triggers warnings will be rejected. The specifier for a warning is appended. By default, this field is clear.
8.	Implicit conversions warnings (-Wconversion)	Check to warn for implicit conversions that may alter a value. This includes conversions between real and integer, like **abs (x)** when x is double; conversions between signed and unsigned, like unsigned **ui = -1**; and conversions to smaller types, like **sqrtf (M_PI)**. Do not warn for explicit casts like **abs ((int) x)** and **ui = (unsigned) -1**, or if the value is not changed by the conversion like in **abs (2.0)**. Warnings about conversions between signed and unsigned integers can be disabled by using **-Wno-sign-conversion**. By default, this field is clear.
9.	Warn on uninitialized variables (-Wuninitialized)	Check to warn if an automatic variable is used without first being initialized or if a variable may be clobbered by a **setjmp** call. In C++, warn if a non-static reference or non-static const member appears in a class without constructors. To warn about code that uses the uninitialized value of the variable in its own initializer, use the **-Winit-self** option. These warnings occur for individual uninitialized or clobbered elements of structure, union or array variables as well as for variables that are uninitialized or clobbered as a whole. They do not occur for variables or elements declared volatile. Because these warnings depend on optimization, the exact variables or elements for which there are warnings depend on the precise optimization options and version of GCC used. Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed. By default, this field is clear.
10.	Warn on various unused elements	Check to warn if various constructs are unused (parameters, local variables, functions, labels, typedef locally defined, function parameters).

	Option	Description
	(-Wunused)	By default, this field is clear.
11.	Warn if padding is included (-Wpadded)	Check to warn if padding is included in a structure, either to align an element of the structure or to align the whole structure. Sometimes when this happens it is possible to rearrange the fields of the structure to reduce the padding and so make the structure smaller. By default, this field is clear.
12.	Warn if floats are compared as equal (-Wfloat-equal)	Check to warn if floating-point values are used in equality comparisons. The idea behind this is that sometimes it is convenient (for the programmer) to consider floating-point values as approximations to infinitely precise real numbers. If you are doing this, then you need to compute (by analyzing the code, or in some other way) the maximum or likely maximum error that the computation introduces, and allow for it when performing comparisons (and when producing output, but that's a different problem). In particular, instead of testing for equality, you should check to see whether the two values have ranges that overlap; and this is done with the relational operators, so equality comparisons are probably mistaken. By default, this field is clear.
13.	Warn if shadowed variable (-Wshadow)	Check to warn whenever a local variable or type declaration shadows another variable, parameter, type, or class member (in C++), or whenever a built-in function is shadowed. Note that in C++, the compiler warns if a local variable shadows an explicit typedef, but not if it shadows a struct/class/enum. By default, this field is clear.
14.	Warn if pointer arithmetic (-Wpointer-arith)	Check to warn about anything that depends on the **sizeof** a function type or of void. GNU C assigns these types a size of 1, for convenience in calculations with void ***** pointers and pointers to functions. In C++, warn also when an arithmetic operation involves **NULL**. This warning is also enabled by **-Wpedantic**. By default, this field is clear.
15.	Warn if suspicious logical ops (-Wlogical-op)	Check to warn about suspicious uses of logical operators in expressions. This includes using logical operators in contexts where a bit-wise operator is likely to be expected. By default, this field is clear.
16.	Warn if struct is returned (-Wagreggate-return)	Check to warn if any functions that return structures or unions are defined or called. (In languages where you can return an array, this also elicits a warning). By default, this field is clear.
17.	Warn on undeclared global function (-Wmissing-declaration)	Check to warn if a global function is defined without a previous declaration. Do so even if the definition itself provides a prototype. Use this option to detect global functions that are not declared in header files.

	Option	Description
		In C, no warnings are issued for functions with previous non-prototype declarations; use **-Wmissing-prototype** to detect missing prototypes. In C++, no warnings are issued for function templates, or for inline functions, or for functions in anonymous namespaces. By default, this field is clear.
18.	Other warning flags	Specifies additional command line options and individual warning flag that can be turned ON/OFF based on the user requirements. Type in custom warning flags that are not otherwise available in the UI. By default, this field is clear.

Miscellaneous

This section describes how to specify miscellaneous C/C++ compiler options.

The table below lists and describes the various options available on the **Miscellaneous** panel.

Table 18: Tool Settings - Standard S32DS C Compiler/ Standard S32DS C++ Compiler - Miscellaneous

	Option	Description
1.	Other flags	Specify additional command line options; type in custom flags that are not otherwise available in the UI. Default: **-c -fmessage-length=0**
2.	Verbose (-v)	Check to show each command-line that it passes to the shell, along with all progress, error, warning, and informational messages that the tools emit. This setting is equivalent to specifying the **-v** command-line option. By default this checkbox is clear. The S32DS ARM v2018.R1 displays just error messages that the compiler emits. The IDE suppresses warning and informational messages.
3.	Support ANSI programs (-ansi) (Standard S32DS C Compiler **only**)	Check this option if you want the assembler to operate in strict ANSI mode. In this mode, the compiler strictly applies the rules of the ANSI/ ISO specification to all input files. This turns off certain features of GCC that are incompatible with ISO C90 (when compiling C code), or of standard C++ (when compiling C++ code), such as the **asm** and **typeof** keywords, and predefined macros such as **unix** and **vax** that identify the type of system you are using. It also enables the undesirable and rarely used ISO trigraph feature. For the C compiler, it disables recognition of C++ style '//' comments as well as the inline keyword. The alternate keywords **__asm__**, **__extension__**, **__inline__** and **__typeof__** continue to work despite **-ansi**. You would not want to use them in an ISO C program, of course, but it is useful to put them in header files that might be included in compilations done with **-ansi**. Alternate predefined macros such as **__unix__** and **__vax__** are also available, with or without **-ansi**.

	Option	Description
		The **-ansi** option does not cause non-ISO programs to be rejected gratuitously. For that, **-Wpedantic** is required in addition to **-ansi**. The macro **__STRICT_ANSI__** is predefined when the **-ansi** option is used. Some header files may notice this macro and refrain from declaring certain functions or defining certain macros that the ISO standard doesn't call for; this is to avoid interfering with any programs that might use these names for other things. Functions that are normally built in but do not have semantics defined by ISO C (such as **alloca** and **ffs**) are not built-in functions when **-ansi** is used. This setting is equivalent to specifying the **-ansi** command-line option. The compiler issues a warning for each ANSI/ISO extension it finds. By default this checkbox is clear. The assembler does not operate in strict ANSI mode
4.	Position Independent Code (-fPIC)	If supported for the target, emit position-independent code, suitable for dynamic linking and avoiding any limit on the size of the global offset table. Position-independent code requires special support, and therefore works only on certain machines. When this flag is set, the macros **__pic__** and **__PIC__** are defined to 2. By default this checkbox is clear.
5.	Save temporary files (--save-temps)	Enables you to save temporary intermediate files and place them in the current directory and name them based on the source file. Use with caution! By default this checkbox is clear.
6.	Generates assembler listing (-Wa, -adhlns="$@.lst")	Enables the assembler to create a listing file as it compiles assembly language into object code. By default this checkbox is clear.

Standard S32DS C Linker and Standard S32DS C++ Linker Panels

This section describes how to specify the C/C++ linker behavior in the build properties for ARM project. The table below lists and describes the various options available on the **Standard S32DS C Linker** and **Standard S32DS C++ Linker** panels.

Table 19: Tool Settings - Standard S32DS C Linker/ Standard S32DS C++ Linker panel

	Option	Description
1.	Command	Shows the location of the linker executable file. • Standard S32DS ARM v2018.R1 C Linker default: **gcc** • Standard S32DS ARM v2018.R1 C++ Linker default: **g++**
2.	All options	Shows the actual command line the linker will be called with. • Standard S32DS ARM v2018.R1 C Linker default: **-Wl,-Map,"1.map" -mthumb -specs= ewl_c_noio.specs**

	Option	Description
		• Standard S32DS ARM v2018.R1 C++ Linker default: **-Wl,-Map,"ARM_CPP_project.map" -mthumb -specs=nano.specs -specs=nosys.specs**
3.	Command line patterns	Shows the expert settings command line parameters. Default: **${COMMAND} ${FLAGS} ${OUTPUT_FLAG} ${OUTPUT_PREFIX} ${OUTPUT} ${INPUTS}**

General

This section describes how to specify the C/C++ linker behavior in the build properties for S32DS project. The table below lists and describes the various options available on the **General** panel.

Table 20: Tool Settings - Standard S32DS C Linker/ Standard S32DS C++ Linker - General

	Option	Description
1.	Do not use standard start files (-nostartfiles)	This option passes the **-nostartfiles** argument to the C/C++ linker file. It does not allow the use of the standard system startup files when linking. The standard system libraries are used normally, unless **-nostdlib** or **-nodefaultlibs** is used By default this checkbox is clear.
2.	Do not use default libraries (-nodefaultlibs)	This option passes the **-nodefaultlibs** argument to the C/C++ linker file. It does not allow the use of the default system libraries when linking. Only the libraries you specify are passed to the linker, and options specifying linkage of the system libraries, such as **-static-libgcc** or **-shared-libgcc**, are ignored By default this checkbox is clear.
3.	No startup or default libs (-nostdlib)	This option passes the **-nostdlib** argument to the C/C++ linker file. It does not allow the use of startup system files or libraries when linking. No startup files and only the libraries you specify are passed to the linker, and options specifying linkage of the system libraries, such as **-static-libgcc** or **-shared-libgcc**, are ignored. The **libgcc.a** library standard bypasses **–nostdlib** and **–nodefaultlibs**. By default this checkbox is clear.
4.	Omit all symbols information (-s)	This option passes the **-s** argument to the C/C++ linker file. This option omits all symbol information and remove all symbol table and relocation information from the executable By default this checkbox is clear.
5.	No shared libraries (-static) (Standard S32DS C Linker **only**)	This option passes the **-static** argument to the C linker file. It does not allow the use of the shared libraries on systems that support dynamic linking By default this checkbox is clear.
6.	Script files (-T)	This option passes the **-T** argument to the C/C++ linker file. This option is supported by most systems using the GNU linker. On some targets, such

	Option	Description
		as bare-board targets without an operating system, the **-T** option may be required when linking to avoid references to undefined symbols The default linker file depends on the selected processor.

Libraries

This section describes how to specify libraries behavior in the build properties for S32DS project. The table below lists and describes the various options available on the **Libraries** panel.

Table 21: Tool Settings - Standard S32DS C Linker/ Standard S32DS C++ Linker - Libraries

	Option	Description
1.	Libraries (-l)	This option changes the build target's search order of access paths to start with the system paths list. The compiler can search **#include** files in several different ways. You can also set the search order as follows: • For include statements of the form **#include"xyz"**, the compiler first searches user paths, then the system paths • For include statements of the form **#include<xyz>**, the compiler searches only system paths This option is global. By default, this field is clear.
2.	Library search path (-L)	Use this option to specify the include library search path. By default, this field is clear.

Miscellaneous

This section describes how to specify miscellaneous C/C++ linker options. The table below lists and describes the various options available on the **Miscellaneous** panel.

Table 22: Tool Settings - Standard S32DS C Linker/ Standard S32DS C++ Linker - Miscellaneous

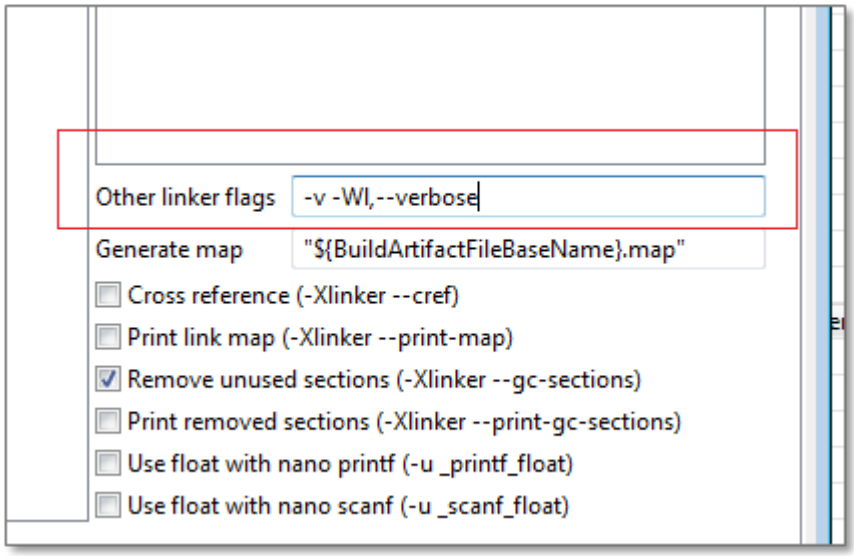
Option	Description
Linker flags	Specify additional command line options for the linker; type in custom flags that are not otherwise available in the UI. By default, this field is empty.
Other options	Specify additional command line options; type in custom flags that are not otherwise available in the UI. By default, this field is empty.
Other objects	This option lists paths that the linker searches for objects. The linker searches the paths in the order shown in this list. By default, this field is empty.
Generate map	This option specifies the map filename. Default: **$ {BuildArtifactFileBaseName}.map**

Option	Description
Cross reference **(-Xlinker --cref)**	Check this option to instruct the linker to list cross-reference information on symbols. This includes where the symbols were defined and where they were used, both inside and outside macros. By default this checkbox is cleared.
Print link map **(-Xlinker --printf-map)**	Check this option to instruct the linker to print the map file. By default this checkbox is cleared.
Remove unused sections **(-Xlinker --gc-sections)**	This option passes the -**Xlinker --gc-sections** argument to the linker file. It removes the unused sections. By default this checkbox is cleared.
Print removed sections **(-Xlinker --print-gc-sections)**	Output unused sections that were removed by garbage collection to the standard error (STDERR) output. **Note:** The console provided by the Console view combines STDOUT and STDERR output. Removed sections will be output the console. `<S32DS_Install>/cross_tools/gcc-6.3-arm32-eabi/bin/ ../lib/gcc/arm-none-eabi/6.3.1/../../../../ arm-none-eabi/bin/real-ld.exe: Removing unused section '.data' in file '<S32DS_Install>/cross_tools/gcc-6.3-arm32-eabi/bin/ ../lib/gcc/arm-none-eabi/6.3.1/thumb/v7e-m/crtbegin.o' <S32DS_Install>/cross_tools/gcc-6.3-arm32-eabi/bin/ ../lib/gcc/arm-none-eabi/6.3.1/../../../../ arm-none-eabi/bin/real-ld.exe: Removing unused section '.text.SystemCoreClockUpdate' in file './Project_Settings/Startup_Code/system_ARMCM7.o'` By default, this check box is cleared.
Support print float format for newlib_nano library **(-u _printf_float)**	This option is active if the **newlib_nano Debugger Console** option was selected from the **Libraries support** list in the **Target Processor** panel. Use this option to support float format from printf. This option affect heap and stack sizes. If it isn't used heap and stack sizes can be decreased. By default, this check box is cleared.
Support scan float format for newlib_nano library **(-u _scanf_float)**	This option is active if the **newlib_nano Debugger Console** option was selected from the **Libraries support** list in the **Target Processor** panel. Use this option to support float format from scanf. This option affect heap and stack sizes. If it isn't used heap and stack sizes can be decreased. By default, this check box is cleared.
EWL print formats	This option is active if the **ewl_nano_c Debugger Console /ewl_nano_c++ Debugger Console** option was selected from the **Libraries support** list in the **Target Processor** panel. Use this option to support "int", "float" or "float and long long" formats from printf. Default: **int**

Option	Description
EWL scan formats	This option is active if the **ewl_nano_c Debugger Console /ewl_nano_c++ Debugger Console** option was selected from the **Libraries support** list in the **Target Processor** panel. Use this option to support "int", "float" or "float and long long" formats from scanf. Default: **int**

Note: The Other linker flags option was available in S32 Design Studio for ARM, Version 1.0. Now (in S32 Design Studio for ARM, Version 2018.R1) only Linker flags field is available. If a project contains some linker flags set in Other linker flags then these options will present in build system but not available for changing or removing. The user shall edit .cproject file to move this flags to the Linker flags field.

For example, in the project the following option was set in the Other linker flags field:



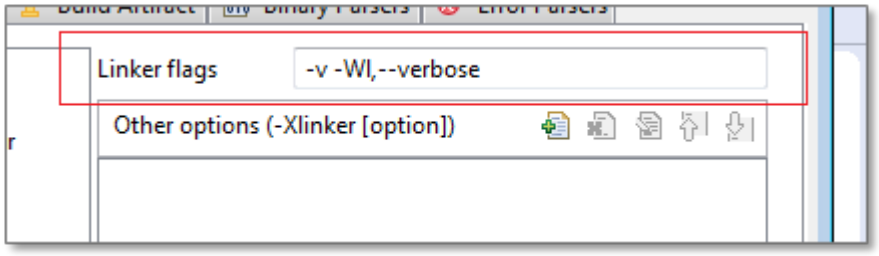
The user should change the line in .cproject file:

```
<option
 id="com.freescale.s32ds.cross.gnu.tool.c.linker.option.other.1380347977"
name="Other linker flags"
superClass="com.freescale.s32ds.cross.gnu.tool.c.linker.option.other"
value="-v -Wl,--verbose"
valueType="string"/>
```

to

```
<option id="gnu.c.link.option.ldflags.2073440811"
superClass="gnu.c.link.option.ldflags"
value="-v -Wl,--verbose"
valueType="string"/>
```

Open the project again. The options will be moved to the Linker flags field:



Standard S32DS Assembler Panel

This section describes how to specify the Assembler behavior in the build properties for ARM C/C++ project. The table below lists and describes the various options available on the Standard S32DS Assembler panel.

Table 23: Tool Settings - Standard S32DS Assembler pane

	Option	Description
1.	Command	Shows the location of the assembler executable file. Default: **${cross_prefix}${cross_c}${cross_suffix}**
2.	All options	Shows the actual command line the assembler will be called with. Default: **-x assembler -O0 -g -mthumb -specs=nano.specs -specs=nosys.specs**
3.	Command line pattern	Shows the expert settings command line parameters. Default: **${COMMAND} ${FLAGS} -c ${OUTPUT_FLAG} ${OUTPUT_PREFIX} ${OUTPUT} ${INPUTS}**

General

This section describes how to set assembler options in the build properties for C/C++ S32DS project. The table below lists and describes the various options available on the **General** panel.

Table 24: Tool Settings - Standard S32DS Assembler - General

	Option	Description
1.	Assembler flags	Specify the flags that need to be passed with the assembler By default this checkbox is clear.
2.	Include paths (-I)	This option changes the build target's search order of access paths to start with the system paths list. The compiler can search **#include** files in several different ways. You can also set the search order as follows: • for include statements of the form **#include"xyz"**, the compiler first searches user paths, then the system paths • for include statements of the form **#include<xyz>**, the compiler searches only system paths. This option is global. By default, this parameter is set to `"${ProjDirPath}/include"`.
3.	Suppress warnings (-W)	This enables some extra warning flags that are not enabled by **-Wall**. By default this checkbox is clear.

	Option	Description
4.	Announce version (-v)	Check this option if you want the S32DS ARM v2018.R1 to show each command-line that it passes to the shell, along with all progress, error, warning, and informational messages that the tools emit. This setting is equivalent to specifying the **-v** command-line option. The IDE displays just error messages that the compiler emits. The IDE suppresses warning and informational messages. By default this checkbox is clear.

Preprocessor

This section describes how to specify assembler options in the build properties for S32DS C/C++ project. The table below lists and describes the various options available on the **Preprocessor** panel.

Table 25: Tool Settings - Standard S32DS Assembler - Preprocessor

	Option	Description
1.	Use preprocessor	Check this option to use the preprocessor for the assembler. By default, this check box is selected.
2.	Do not search system directories (-nostdinc)	Check if you do not want the assembler to search the system directories. The assembler performs a full search that includes the system directories. By default this checkbox is clear.
3.	Preprocess only (-E)	Check if you want the assembler to preprocess source files and not to run the compiler. Nothing is done except preprocessing. By default, this checkbox is clear and the source files are not preprocessed.

Symbols

This section describes how to set assembler options in the build properties for S32DS C/C++ project. The table below lists and describes the various options available on the **Symbols** panel.

Table 26: Tool Settings - Standard S32DS Assembler - Symbols

	Option	Description
1.	Defined symbols (-D)	Use this option to specify the substitution strings that the assembler applies to all the assembly-language modules in the build target. Enter just the string portion of a substitution string. The S32DS ARM v2018.R1 prepends the **-D** token to each string that you enter. For example, entering **opt1 x** produces this result on the command line: **-Dopt1 x** **Note:** This option is similar to the DEFINE directive, but applies to all assembly-language modules in a build target By default this checkbox is clear.
2.	Undefined symbols (-U)	Undefines the substitution strings you specify in this panel

	Option	Description
		By default this checkbox is clear.

Optimization

This section describes how to control optimizations in the build properties for S32DS C/C++ project. The table below lists and describes the various options available on the **Optimization** panel.

Table 27: Tool Settings - Standard S32DS Assembler - Optimization

	Option	Description
1.	Optimization level	Specify the optimizations that you want the assembler to apply to the generated object code: • **None (-O0)** - Disable optimizations. This setting is equivalent to specifying the **- O0** command-line option. The assembler generates unoptimized, linear assembly-language code, reduce compilation time. • **Optimize (-O1)** - The assembler performs all target-independent (that is, non-parallelized) optimizations, such as function inlining. Optimizing takes somewhat more time, and a lot more memory for a large function. This setting is equivalent to specifying the **-O1** command-line option. The assembler omits all target-specific optimizations and generates linear assembly-language code. • **Optimize more (-O2)** - The assembler performs all optimizations (both target-independent and target-specific). GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to **-O**, this option increases both compilation time and the performance of the generated code. This setting is equivalent to specifying the **-O2** command-line option. The compiler outputs optimized, non-linear, parallelized assembly-language code. • **Optimize most (-O3)** - The assembler performs all the level 2 optimizations then the low-level optimizer performs global-algorithm register allocation. This setting is equivalent to specifying the **-O3** command-line option. At this optimization level, the compiler generates code that is usually faster than the code generated from level 2 optimizations. • **Optimize size (-Os)** - The assembler performs further optimizations designed to reduce code size. **-Os** enables all **-O2** optimizations that do not typically increase code size. The resulting binary file has a smaller executable code size, as opposed to a faster execution speed. This setting is equivalent to specifying the **-Os** command-line option. Default: **None (-O0)**
2.	Other optimization flags	Specifies additional command line options and individual optimization flag that can be turned ON/OFF based on the user requirements. Type in custom optimization flags that are not otherwise available in the UI. By default this checkbox is clear.

Debugging

This section describes how to control debugging in the build properties for C/C++ S32DS project. The table below lists and describes the various options available on the **Debugging** panel.

Table 28: Tool Settings - Standard S32DS Assembler - Debugging

	Option	Description
1.	Debug Level	Specify the debug levels: • **None** - Level 0 produces no debug information at all, • **Minimal (-g1)** - Level 1 produces minimal information, enough for making backtraces in parts of the program that you don't plan to debug. This includes descriptions of functions and external variables, and line number tables, but no information about local variables, • **Default (-g)** - The default level is 2. The compiler generates DWARF 1.x conforming debugging information, • **Maximum (-g3)** - Level 3 includes extra information, such as all the macro definitions present in the program. So the compiler provides maximum debugging support. Default: **Maximum (-g3)**
2.	Other debugging flags	Specify additional command line options; type in custom debugging flags that are not otherwise available in the UI. By default, this field is clear.

Standard S32DS C Preprocessor and Standard S32DS C++ Preprocessor Panels

This section describes how to specify the C/C++ preprocessor behavior in the build properties for S32DS project. The table below lists and describes the various options available on the **Standard S32DS C Preprocessor** and **Standard S32DS C++ Preprocessor** panels.

Table 29: Tool Settings - Standard S32DS C Preprocessor/ Standard S32DS C++ Preprocessor pane

	Option	Description
1.	Command	Shows the location of the preprocessor executable file. • Standard S32DS C Preprocessor default: **${cross_prefix}${cross_c}${cross_suffix}** • Standard S32DS C++ Preprocessor default: **${cross_prefix}${cross_cpp}${cross_suffix}**
2.	All options	Shows the actual command line the preprocessor will be called with. Default: **-E**
3.	Command line patterns	Shows the expert settings command line parameters. By default, this field is set to: **${COMMAND} ${FLAGS} ${INPUTS}**

Settings

This section describes how to specify the preprocessor settings in the build properties for S32DS project. The table below lists and describes the various options available on the **Settings** panel.

Table 30: Tool Settings - Standard S32DS C Preprocessor/ Standard S32DS C++ Preprocessor - Settings

	Option	Description
1.	Handle Directives Only fdirectives-only)	Check to specify the **-fdirectives-only** command to the C/C++ preprocessor to handle only directives, but do not expand macros. The option's behavior depends on the **-E** and **-fpreprocessed** options. By default this checkbox is clear.
2.	Print Header File Names (-H)	Check to specify the **-H** command to the C/C++ preprocessor to print the name of each header file used, in addition to other normal activities. Precompiled header files are also printed, even if they are found to be invalid. By default this checkbox is clear.

Standard S32DS Disassembler Panel

This section describes how to specify the disassembler behavior in the build properties for C/C++ S32DS project. The table below lists and describes the various options available on the **Standard S32DS Disassembler** panel.

Table 31: Tool Settings - Standard S32DS Disassembler pane

	Option	Description
1.	Command	Shows the location of the disassembler executable file. Default: **${cross_prefix}objdump${cross_suffix}**
2.	All options	Shows the actual command line the disassembler will be called with. Default: **-d -S -x**
3.	Command line pattern	Shows the expert settings command line parameters. Default: **${COMMAND} ${FLAGS} ${INPUTS}**

Settings

This section describes how to specify the disassembler settings in the build properties for S32DS project. The table below lists and describes the various options available on the **Settings** panel.

Table 32: Tool Settings - Standard S32DS Disassembler - Settings

	Option	Description
1.	Disassemble All Section Content (including debug information) (-D)	Check to specify the **-D** command to the disassembler, to disassemble all section content and send the output to a file. This command is global and case sensitive. By default this checkbox is clear.
2.	Disassemble Executable Section Content (-d)	Check to specify the **-d** command to the disassembler, to disassemble all executable content and send output to a file. By default this checkbox is checked.

	Option	Description
3.	Intermix Source Code With Disassembly (-S)	Check to specify the **-S** command to the disassembler, to convert jbsr into jsr. By default this checkbox is checked.
4.	Display All Header Content (-x)	Check to specify the **-x** command to the disassembler, to display the contents of all headers. By default this checkbox is checked.
5.	Display Archive Header information (-a)	Check to specify the **-a** command to the disassembler, to display the archive header information. By default this checkbox is inactive.
6.	Display Overall File Header content (-f)	Check to specify the **-f** command to the disassembler, to display the contents of the overall file header. By default this checkbox is inactive.
7.	Display Object Format Specific File Header Contents (-p)	Check to specify the **-p** command to the disassembler, to display the file header contents and object format. By default this checkbox is inactive.
8.	Display Section Header Content (-h)	Check to specify the **-h** command to the disassembler, to display the section header of the file. By default this checkbox is inactive.
9.	Display Full Section Content (-s)	Check to specify the **-s** command to the disassembler, to display the full section of the file. By default this checkbox is clear.
10.	Display Debug Information (-g)	Check to specify the **-g** command to the disassembler, to display debug information in the object file. By default this checkbox is clear.
11.	Display Debug Information Using ctag Style (-e)	Check to specify the **-e** command to the disassembler, to display debug information using the ctags style. By default this checkbox is clear.
12.	Display STABS Information (-G)	Check to specify the **-G** command to the disassembler, to display any STABS information in the file, in raw form. By default this checkbox is clear.
13.	Display DWARF Information (-W)	Check to specify the **-W** command to the disassembler, to display any DWARF information in the file. By default this checkbox is clear.
14.	Display Symbol Table Content (-t)	Check to specify the **-t** command to the disassembler, to display the contents of the symbol tables.

	Option	Description
		By default this checkbox is clear.
15.	Display Dynamic Symbol Table Content (-T)	Check to specify the **-T** command to the disassembler, to display the contents of the dynamic symbol table. By default this checkbox is clear.
16.	Display Relocation Entries (-r)	Check to specify the **-r** command to the disassembler, to display the relocation entries in the file. By default this checkbox is clear.
17.	Display Dynamic Relocation Entries (-R)	Check to specify the **-R** command to the disassembler, to display the dynamic relocation entries in the file. By default this checkbox is clear.

Toolchain customization

User can specify which tools the builder needs to include when it builds the project for a specified toolchain and configuration. Normally, users need not to edit toolchains manually.

To customize the toolchain used in your build configuration, perform these steps:

1. Follow the steps listed in the "Changing Build Properties" section.
2. Select from build properties list the **Toolchain Editor** item. The **Toolchain Editor** pane appears on the right. The **Used tools** list presents the set of default tools, which are available by default just after the project creation.
3. Define toolchain settings as specified in sections "Defining C/C++ Build Settings" and "C/C++ Build Tool Settings".

   Note: Tools from the set generated after the project creation cannot be removed from this set. Tools which were added manually to toolchain can be deleted by using the **Select Tools** window.

View/manage resources in build configurations

Resources can be viewed/ managed in the **Build configurations** dialog. For open **Build configurations** dialog:

1. Select the project in the **Project Explorer** view.
2. Call the context menu by right click.
3. Select **Build Configurations Explorer** in the context menu. The **Build configurations** dialog will be displayed.

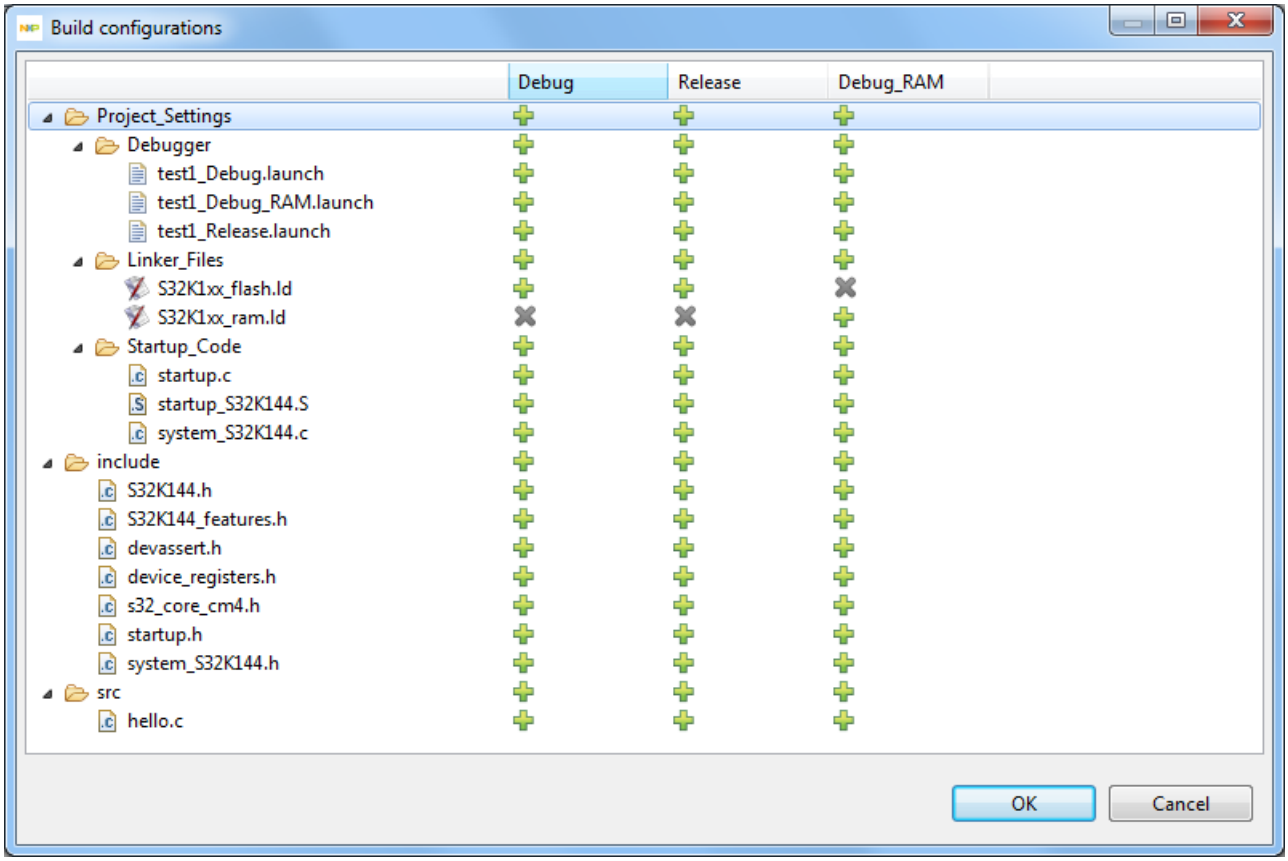
   A tree of project elements is displayed on the left part of the window. Folders and files compose a hierarchy of the project elements. Build configurations can be viewed or modified by selecting files.

   The columns corresponding to build configuration are presented on the right from the elements tree. In our example we have two configurations: **Debug** and **Release**.

   Green plus or grey cross signs can be set in each column:

   - Green plus sign - the checked element is included in the build configuration (configuration name is the column header);
   - Grey cross sign - the checked element is excluded from the configuration.

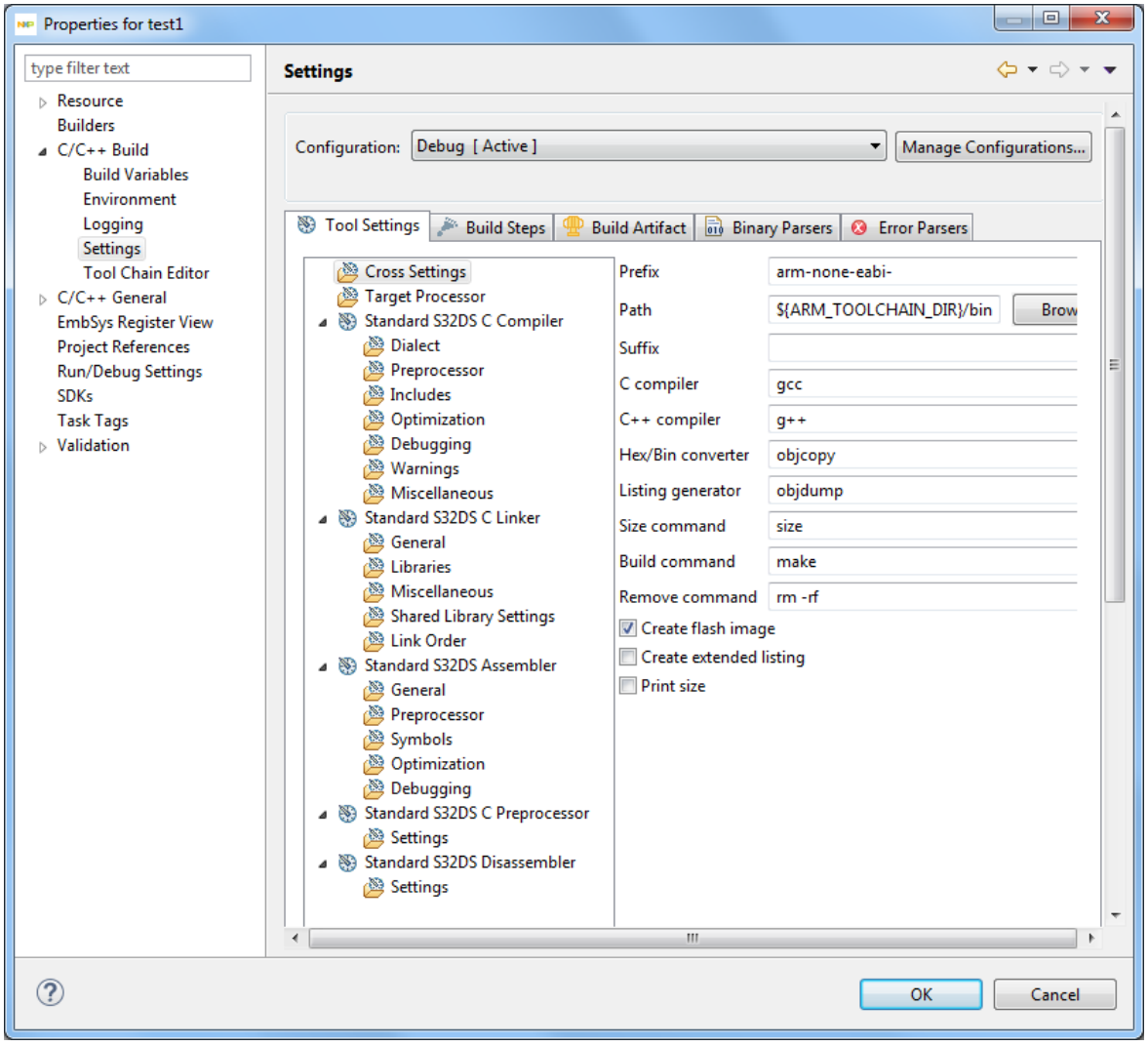
   To change the state of the selected element, click on the icon in configuration column.



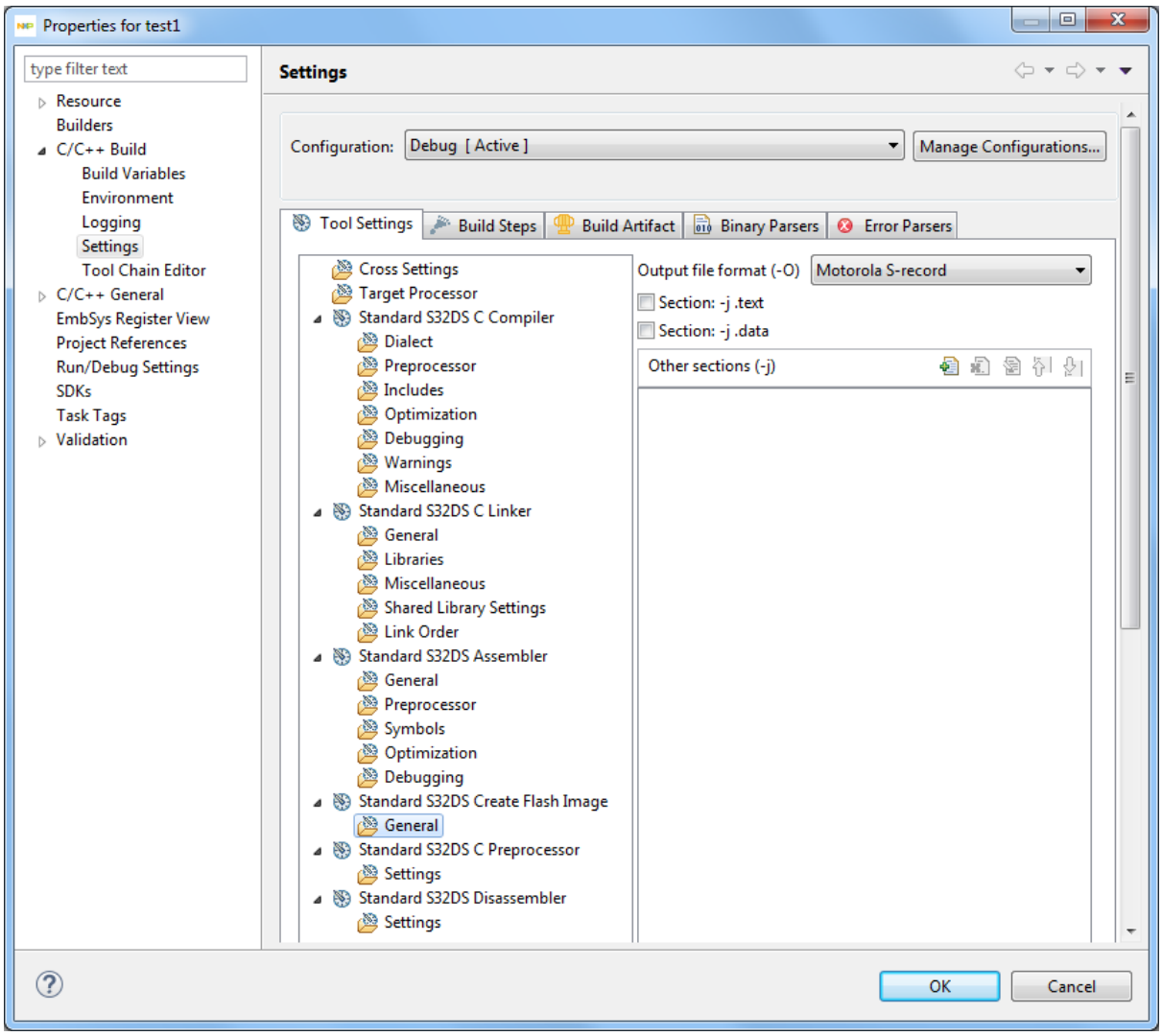
Generate S-record image

This section describes how to generate S-record image. To generate S-record image, perform the following steps:

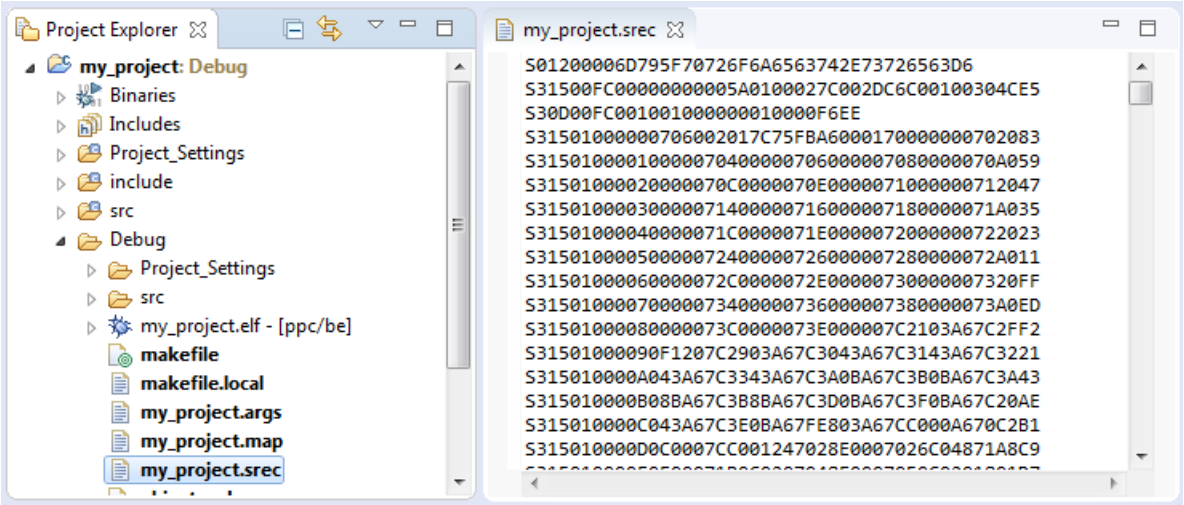
1. Go to the project **Properties** > **C/C++ Build** > **Settings** > **Cross Settings** and enable **Create flash image** option.



2. Press **Apply** button and **Standard S32DS Create Flash Image** settings should appear in the list.
3. Select **Standard S32DS Create Flash Image** > **General** settings. The **Motorola S-record** value is selected by default from the **Output file format (-O)** list. Press **Apply** and **OK** buttons.



4. Rebuild the project and check **<project name>.srec** file in the **Debug** output directory.



Chapter 4

Working with debugger

Topics:

S32DS ARM v2018.R1 project can have multiple associated launch configurations. A launch configuration is a named collection of settings that the S32DS ARM v2018.R1 tools use.

Customizing Launch Configuration

The **Debug Configurations** dialog box contains 6 tabs allowing you to customize all aspects of a launch configuration:

- Main
- Debugger
- Startup
- Source
- Common
- OS Awareness

As you modify a debug configuration's settings, you create pending, or unsaved, changes to that debug configuration. To save the pending changes, you must click the **Apply** button at the bottom of the **Debug Configurations** dialog box, or click the **Close** button and then the **Yes** button of the **Save Changes** dialog box.

You can revert pending changes and restore their last saved settings. To undo pending changes, click the **Revert** button at the bottom of the **Debug Configurations** dialog box. The S32DS ARM v2018.R1 restores the last set of saved settings to all pages of the **Debug Configurations** dialog box. Also, the IDE disables the **Revert** button until you make new pending changes.

The settings of specific debug configurations depend on the used connection type (GDB Hardware, GDB PEMicro Interface or GDB SEGGER J-Link).

Note: More information about customizing connection parameters for GDB PnE Micro connection interface can be found in **PnE GDB Server Plug-In for ARM Devices. Debug Configuration User Guide**. The document can be found on the **Reference Manuals** page of S32DS ARM v2018.R1 Documentation Suite. The Documentation Suite is available when you open the **Documentation** link located under S32 Design Studio for ARM, Version 2018.R1 in the Start menu (on Windows platform) or on the desktop (on Linux platform). For information about customizing debug configuration for GDB *SEGGER J-Link*, please visit the The J-Link debugging Eclipse plug-in webpage.

Main

Use the **Main** pane to specify the project and the application you want to run or debug. The **Main** tab presents groups of options for specifying different settings.

Table 33: Main tab options

Group	Option	Description
N/A These options are ungrouped.	**Project**	Specifies the project to associate with the selected debug launch configuration. Click **Browse** to select a different project.
	Specify the number of additional Elf Files you wish to program	Enter the number of additional Elf-files and click **Generate Elf Fields**. Specify path to the additional elf(-s) in the field(-s) **Specify Additional Elf <elf number>.**
	C/C++ application	Specifies the name of the C or C++ application
	Variables…	Click to open the **Select Variable** dialog box and choose the build variables to be associated with the program. See details in **Common Features Guide** (**Environment variables in debug configuration** topic)
	Search Project…	Click to open the Program Selection dialog box and select a binary

Group	Option	Description
Build (if required) before launching		Controls how auto build is configured for the launch configuration. Changing this setting overrides the global workspace setting and can provide some speed improvements.
	Build configuration	Specify the build configuration either explicitly, or use the current active configuration. By default, the list is set to the currently active build configuration, which is selected in the tree of debug configurations available in the left pane of **Debug Configurations** window.
	Enable auto build	Always build project before launching. This may slow down launch performance. By default, this check box is cleared.
	Disable auto build	Disables auto build for the debug configuration. No build action will be performed before starting the debug session. Requires manually building project before launching. This may improve launch performance. By default, this check box is cleared.
	Use workspace settings	Uses the global auto build settings. By default this check box is selected.
	Configure Workspace Settings…	The hyperlink opens the **Launching** preference panel where you can change the workspace settings. It will affect all projects that do not have project specific settings.

Debugger

Use the **Debugger** pane to tune a debugger to use when debugging an application. The **Debugger** tab presents groups of options for specifying different settings.

Note:

The options set under the Debugger tab change depending on the derivative and connection selected while creating the project.

About connections settings see Connections chapter.

If the user creates the new debug configuration (for existed project or after elf import) then following settings shall be checked:

- for *GDB PnE Micro Interface* debug configuration - the device and GDB client are set correctly, so the **Target** field and **Executable** textboxes of the **Debugger** pane should not be empty.
- for *SEGGER J-Link* debug configuration - the device and GDB server are set correctly, so the **Device name** and **Executable** textboxes of the **Debugger** pane should not be empty.

Startup

Use the **Startup** pane to specify specific options used to configure the debug session.

Note: The options set under the **Startup** tab change depending on the derivative and connection selected while creating the project.

Source

Use the **Source** pane to specify the location of source files used when debugging a C or C++ application. By default, this information is taken from the build path of your project. The **Source** tab options are explained in the following table.

Note: The options set under the **Source** tab change depending on the derivative and connection selected while creating the project.

Common

Use the **Common** pane to specify the location to store your run configuration, standard input and output, and background launch options. The **Common** tab presents groups of options for specifying different settings.

Note: The options set under the **Common** tab change depending on the derivative and connection selected while creating the project.

OS Awareness

The **OS Awareness** tab enables you to inform the debugger of the operating system (OS) the target is running. This enables the debugger to provide additional functionality specific to the selected OS.

Use the **OS Awareness** pane to select the **OS** option. The **OS Awareness** tab presents one drop-down list. The next options are available in the drop-down list:

- FreeRTOS
- OSEK

OS awareness depends on having debug symbols for the OS loaded within the debugger.

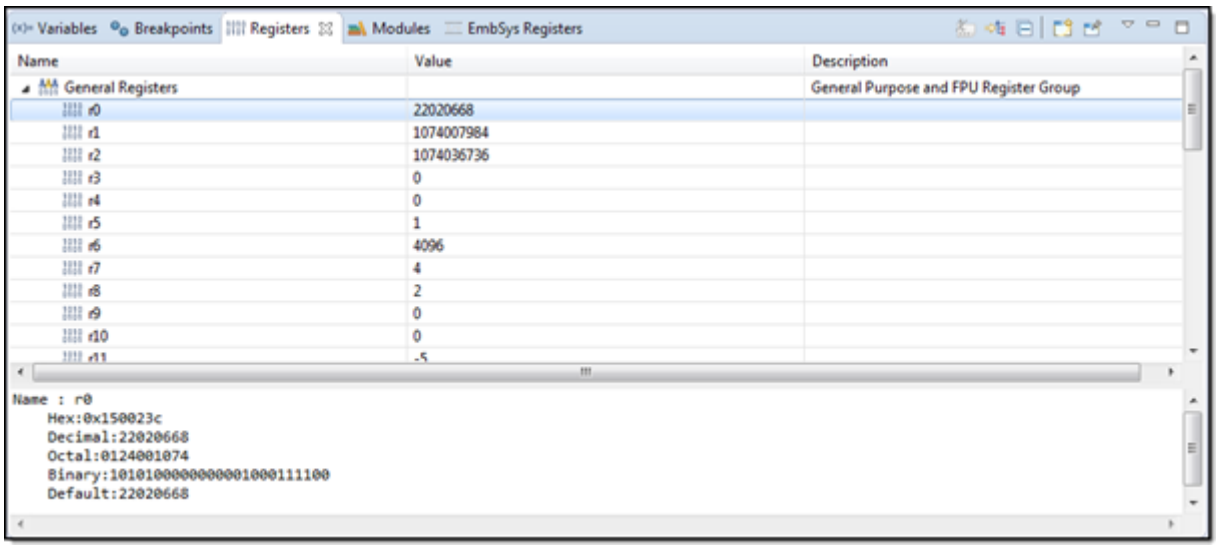
After selecting an **OS** the **OS Resources** view is available. See Common Features Guide for S32 Design Studio for ARM, Version 2018.R1.

Debugging Bareboard Software

S32DS ARM v2018.R1 cannot export/import register data to/from file and show registers information offline (without a debug session).

Displaying register contents

Use the **Registers** view to display and modify the contents of the registers of the processor on your target board. To display this view from the **Debug** perspective, select from the S32DS ARM v2018.R1 menu bar**Window** > **Show View** > **Registers**, and the **Registers** view appears. The following figure shows the **Registers** view with the **General Registers** tree element expanded:



The **Registers** view displays categories of registers in a tree format. To display the contents of a particular category of registers, expand the tree element of the register category of interest.

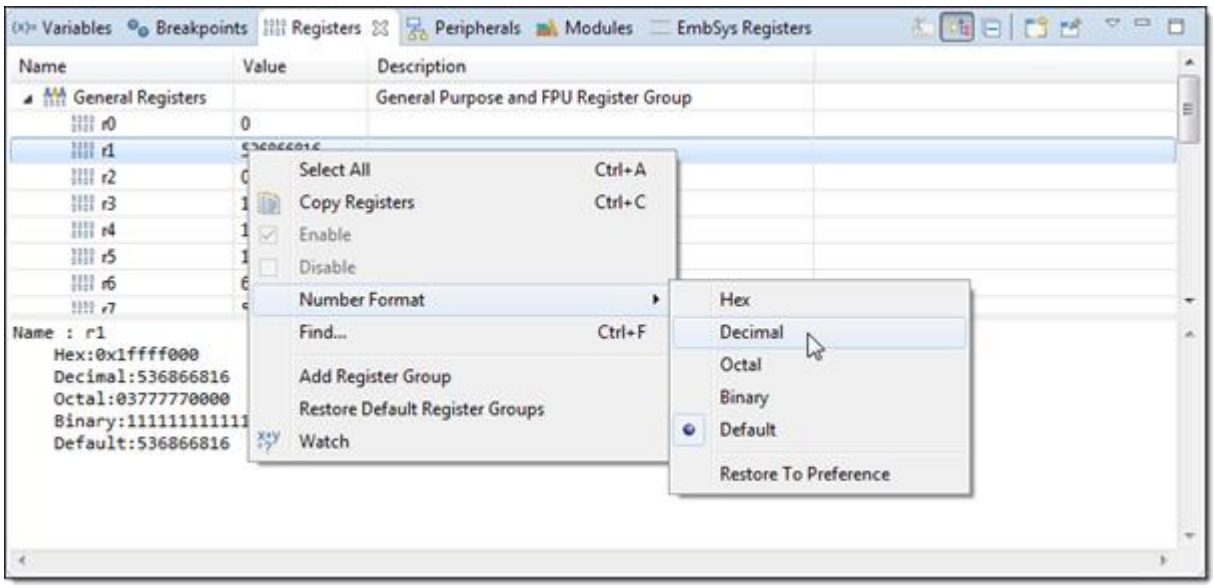
Changing register data display format

User can change the format in which the debugger displays the contents of registers. For example, user can specify that a register's contents be displayed in hexadecimal, rather than binary. The debugger provides these data formats:

- Default
- Decimal
- Hexadecimal
- Octal
- Binary

To change register display format:

1. Open the **Registers** view. Expand the hierarchical list to reveal the register for which you want to change the display format.
2. Select the register value that you want to view in a different format. The value highlights.
3. Right-click to display the pop-up menu and choose **Number Format** > **<data format>** from the context menu that appears, where **<data format>** is the data format in which you want to view the register value.



The selected register value changes format.

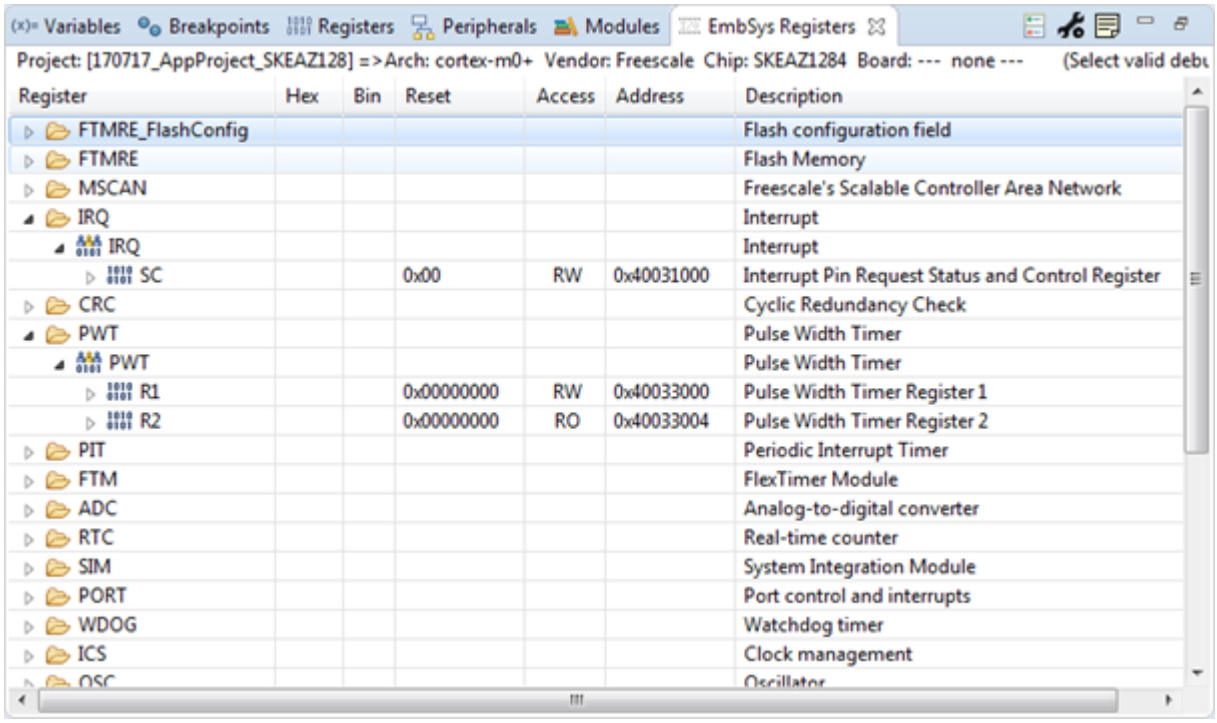
View peripheral registers

EMBedded **SYS**tems Registers view is designed for monitoring and modifying memory values of embedded devices. Therefore it offers a structured display of the special functions registers (SFR).

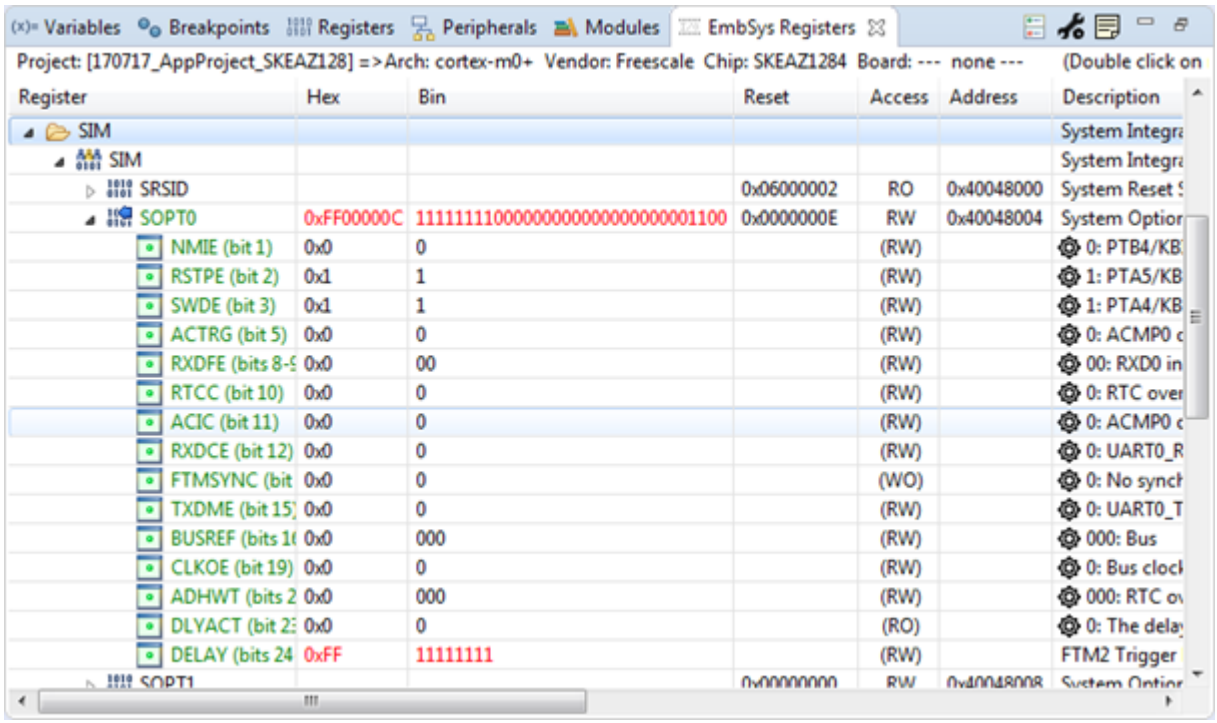
EmbSys Registers view is available on the **Debug** Perspective. To display this view from the **Debug** perspective, select **Window** > **Show View** > **Other** > **Debug** > **EmbSysRegisters** from the S32DS ARM v2018.R1 menu bar, and the **EmbSys Registers** view appears.

The **EmbSys Registers** view displays categories of registers in a tree format. To display the contents of a particular category of registers, expand the tree element of the register category of interest.

The register values are presented in the Hexadecimal (**HEX**) and Binary (**Bin**) column of the view. As well as information about value out of Reset, Access type, Address and description:



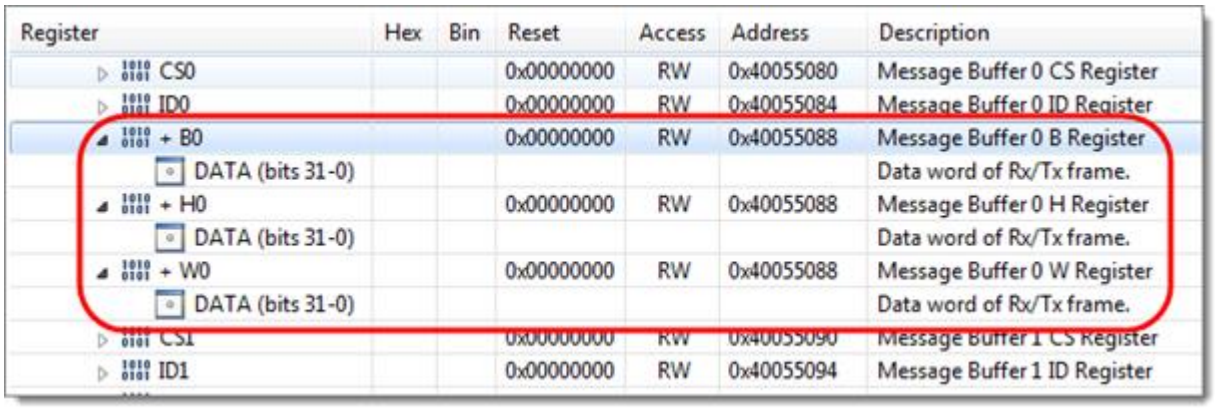
Bit level details as well available for registers. To read actual register value from target – double-click on the register name, the name of register will change its color to green, only one register will be read from memory on each debug action like step, resume/stop, breakpoint. If value of register changed, its color will change to red.



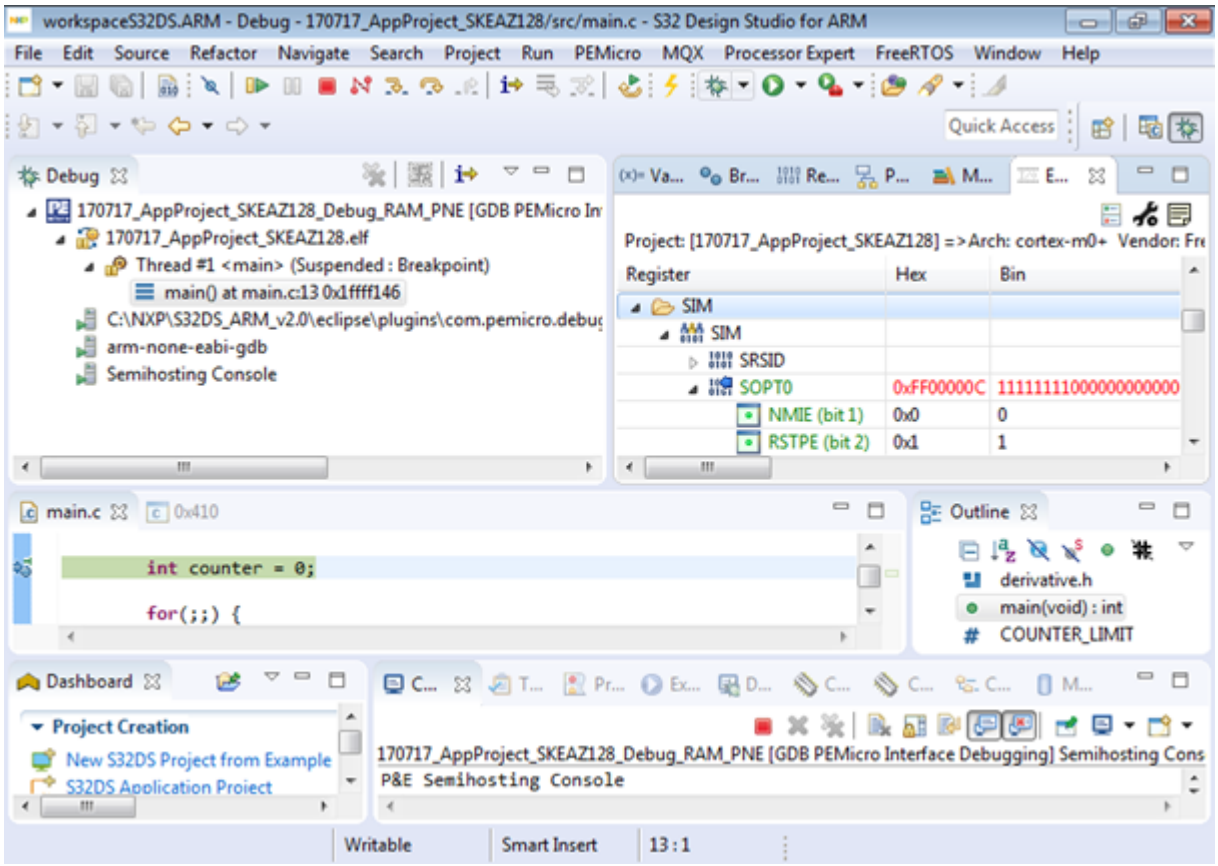
To stop reading the register value – double-click on its name again. To read entire group of registers – double-click on the group name.

Note: Be careful when selecting register group with big number of registers for reading as it might slow down the debugging.

The **Register** field can contain the "+" sign placed before the register name. These marked rows are aliases of a register having one common address. See example:

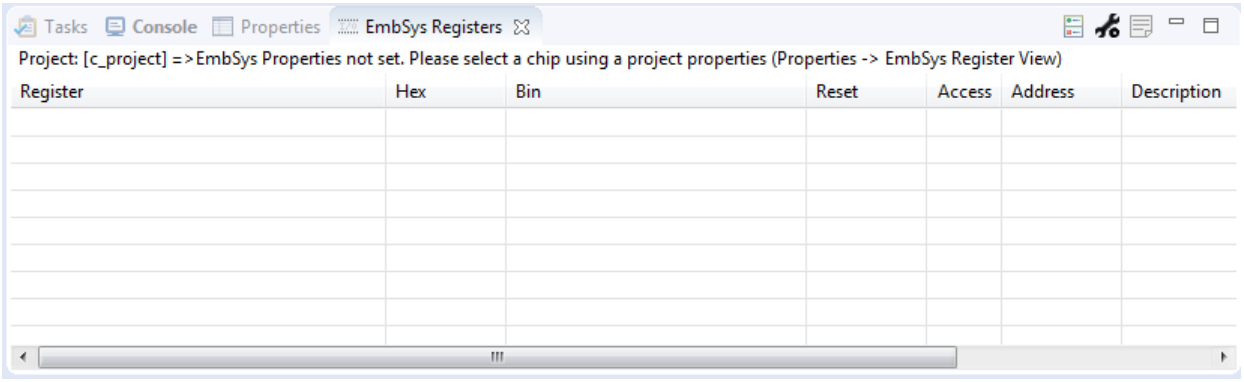


View is updated when user choose a project in **Project Explorer**. View displays register info including when debug session is off. View is updated by choosing debug context in the **Debug** view.



If different projects were chosen in some project's view and in the **Debug** view, **EmbSys Register** view is updated with the values of project, chosen in the just activated view.

If project, chosen in some project's view, have no settings for **EmbSys Register** view, view becomes empty and the instruction message is shown to user.



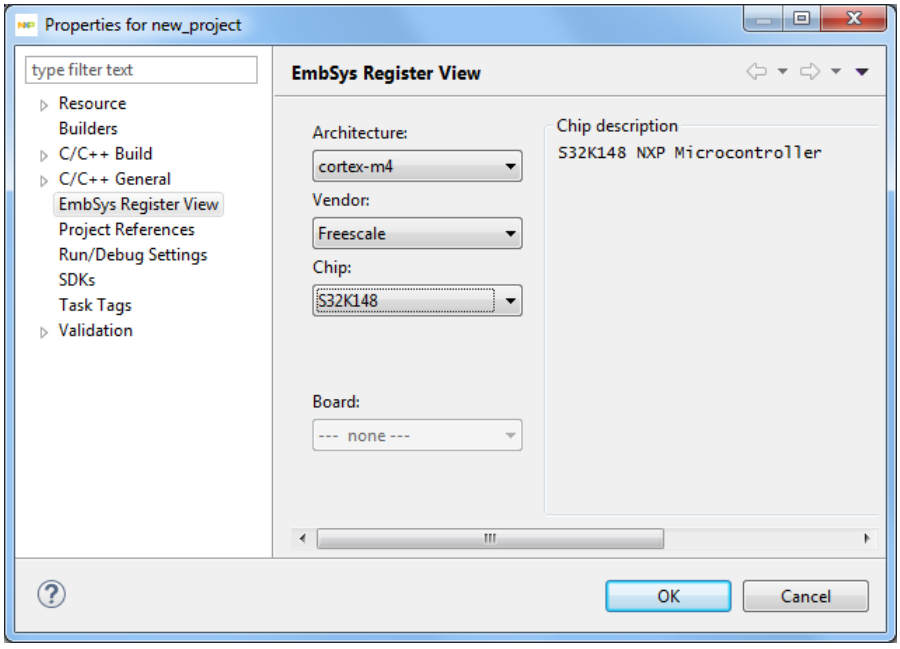
View responds to changes of project properties including when debug session is on.

The table below describes configuration buttons on the view toolbar.

Table 34: EmbSys Registers pane – toolbar buttons

Button	Description
Picked cherries	Filters and displays the selected rows of registers
EmbSysRegView Project Properties	Allows to directly access to the Properties for <project name> dialog box with XML-settings of a selected project
Copy selection to clipboard	Copies information from selected row (or rows) to clipboard: **Register**, **Hex**, and **Address** fields

XML data can be configured on project properties page.



View behavior can be configured for all projects in the global settings:

Window > **Preferences** > **C/C++** > **Debug** > **EmbSys Register View Behavior**:

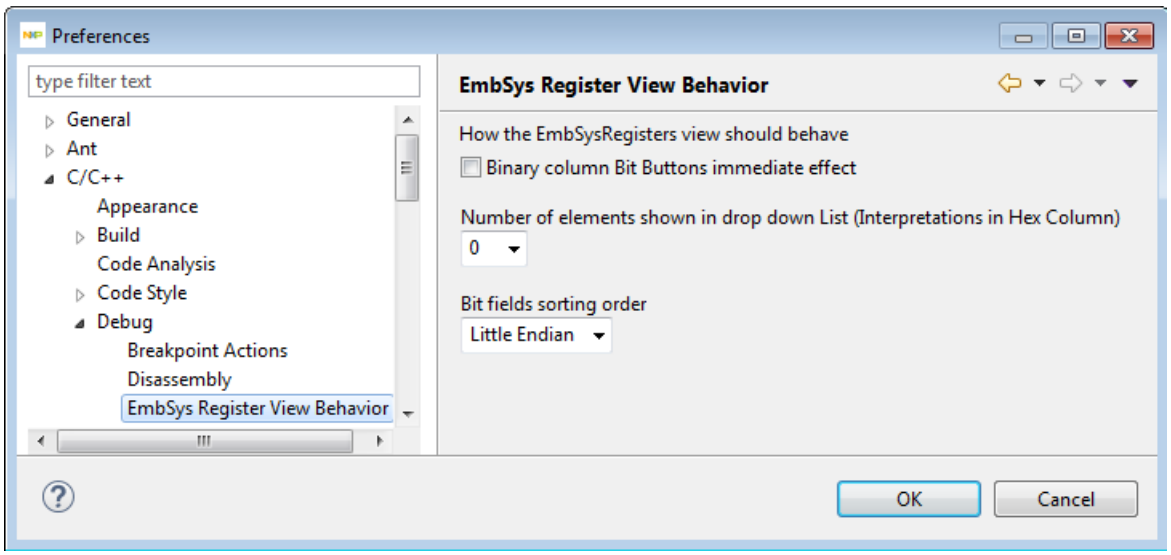


Table 35: Tool Settings - EmbSys Register View Behavior pane

	Option	Description
1.	Binary column Bit Buttons immediate effect	Check to enable Binary column Bit Buttons immediate effect
2.	Number of elements shown in the drop list (interpretation in the Hex column)	Use to specify number of elements shown in the drop list

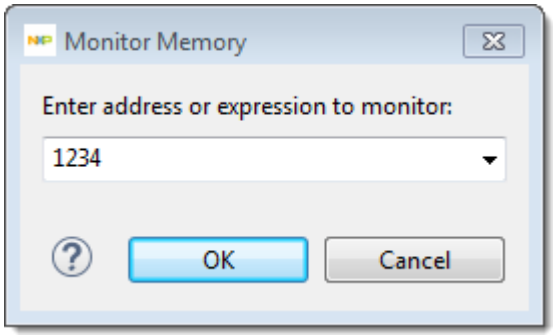
Viewing memory

Use the **Memory** view to examine the active memory rendering of a specified expression or address. The **Memory** view supports the display of multiple memory spaces. See details in **S32DS ARM v2018.R1 Common Features Guide** (**Memory view** topic)

Adding memory monitor

You can add multiple memory monitors to the **Memory** view. To add a new memory monitor, perform these steps:

1. Start a debugging session.
2. Open the **Memory** view.
3. Click the plus-sign icon on the **Monitors** pane toolbar: Alternatively, right-click in the **Monitors** pane and select **Add Memory Monitor** from the context menu. The **Monitor Memory** dialog box appears.



4. Enter address or expression to monitor in decimal or hexadecimal values. You can use the drop-down list to select a previously specified expression.

Important: S32DS ARM v2018.R1 allows you to evaluate any expression, not only addresses, but also the function calls. For example, if you specify a function call such as `main()` as an expression, adding the memory monitor will fail. However, the expression will be evaluated. The main function will be called from the debugger. Note that it may be called multiple times. For example, the first call may happen when the entry point of your application is processed. Second, it will be invoked when evaluating the expression that yields the call of `main()`. The function will be called to call itself within the expression. If your `main()` contains a loop, evaluating it in the expression specified for the memory monitor will result in an endless loop.

5. Click **OK**. New memory monitor appears in the **Memory** view.

Adding memory rendering

You can use the Renderings pane of the **Memory** view to examine the memory content, starting at any valid address. The information displayed in this page is read only and cannot be used to modify the memory content.

To add a new memory rendering, perform these steps.

1. Start a debugging session.
2. Open the **Memory** view.
3. In the Monitors pane, select the memory monitor for which you want to add a memory rendering.
4. Click the **New Renderings…** tab to select renderings.
5. Select a rendering type from the **Select rendering(s) to create** list and click the **Add Rendering(s)** button. Alternatively, right-click in the Renderings pane and select **Add Rendering** from the context menu. The selected memory rendering type appears in the **Memory** view.

Removing memory rendering

To remove a memory rendering from the **Memory** view, perform these steps:

1. Open the **Memory** view.
2. In the Renderings pane, select the tab that corresponds to the memory rendering that you want to remove.
3. Click the cross-sign icon on the Renderings pane toolbar: ✖ . Alternatively, right-click on the Renderings pane and select **Remove Rendering** from the context menu. The memory rendering is removed from the **Memory** view.

Resetting to base address

To reset the memory rendering and display the base address of the rendering, perform these steps.

1. Open the **Memory** view.
2. In the Renderings pane, select the tab that corresponds to the disassembly rendering that you want to reset to the base address.
3. Right-click in the Renderings pane and select **Reset to Base Address** from the context menu.
4. The disassembly rendering scrolls to the line that contains the base address of the displayed rendering.

Go to address

The **Memory** view provides graphical controls to display memory at a specific address.

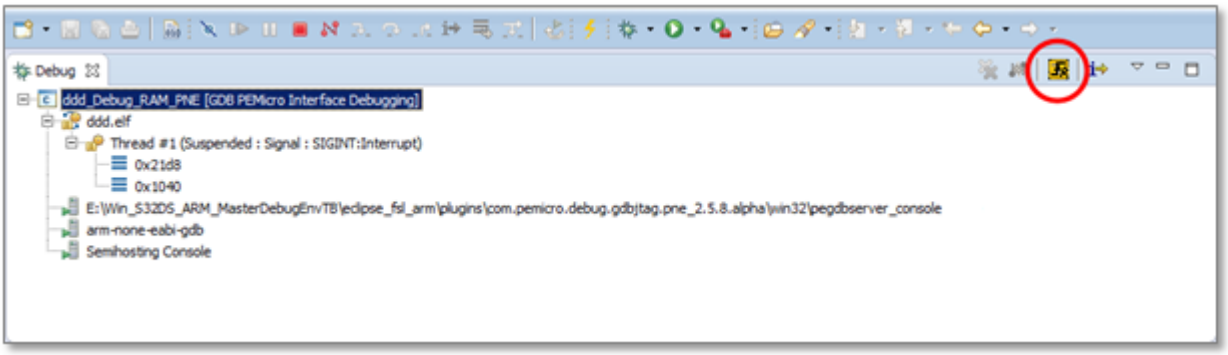
To go to a specific address, perform these steps:

1. Open the **Memory** view.
2. In the Renderings pane, select the tab that corresponds to the disassembly rendering for which you want to display a specific address.
3. Right-click in the Renderings pane and select **Go to Address…** from the context menu. A group of controls appears on the Renderings pane.
4. In the blank text box, enter the address that you want to display.

   Note: Check the **Input as Hex** checkbox only if you enter the address in hexadecimal notation.
5. Click **OK** to have the Disassembly rendering scroll to the specified address. Alternatively, click **Cancel** to abort the operation and hide the group of controls.

Target resetting

Use the **Send HW Reset to target** button to reset your target board. To send HW Reset from the **Debug** perspective, select from the Debug toolbar the **Send HW Reset to target** button and control command execution in the **Console** view. The following figure shows the **Debug** view with the **Send HW Reset to target** button:



If the **Send HW Reset to target** button is enabled then the restart button in the main toolbar is disabled.

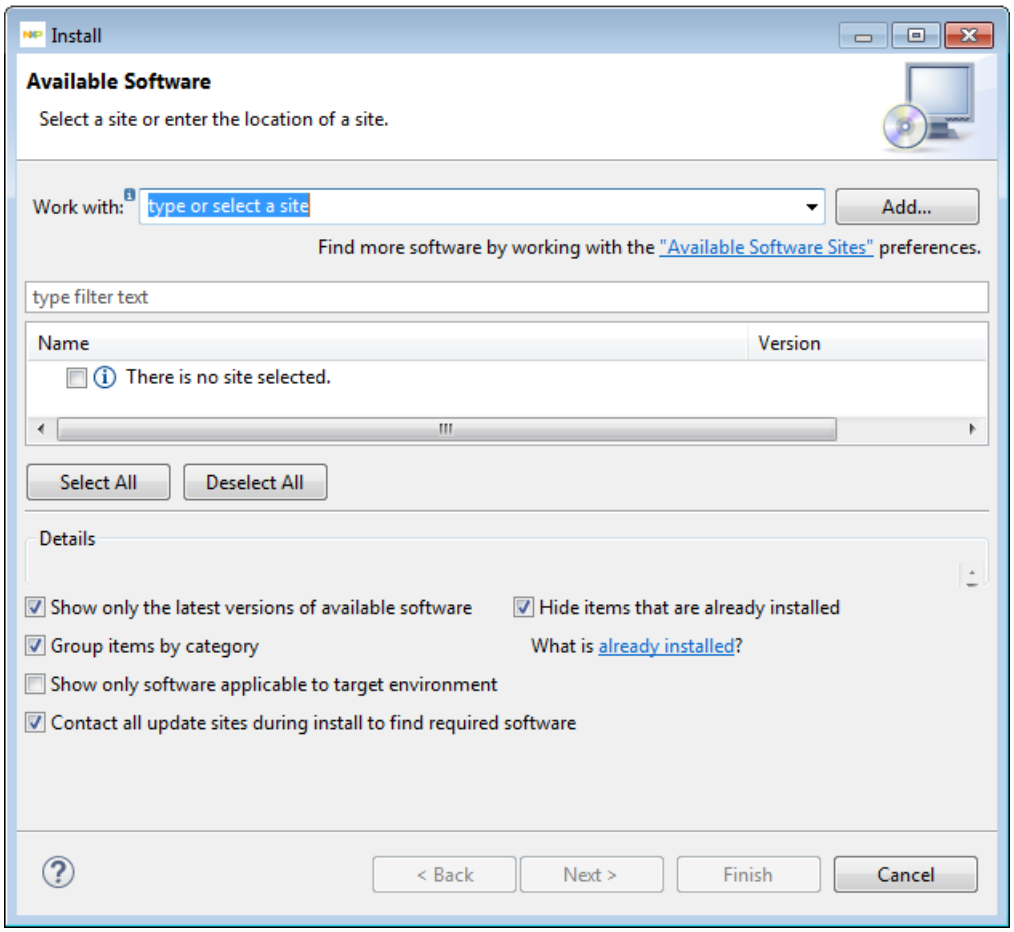
Use the Lauterbach plugin for debugging

The following section explains how to start the debugging session for your project by using the Lauterbach TRACE32 debugger.

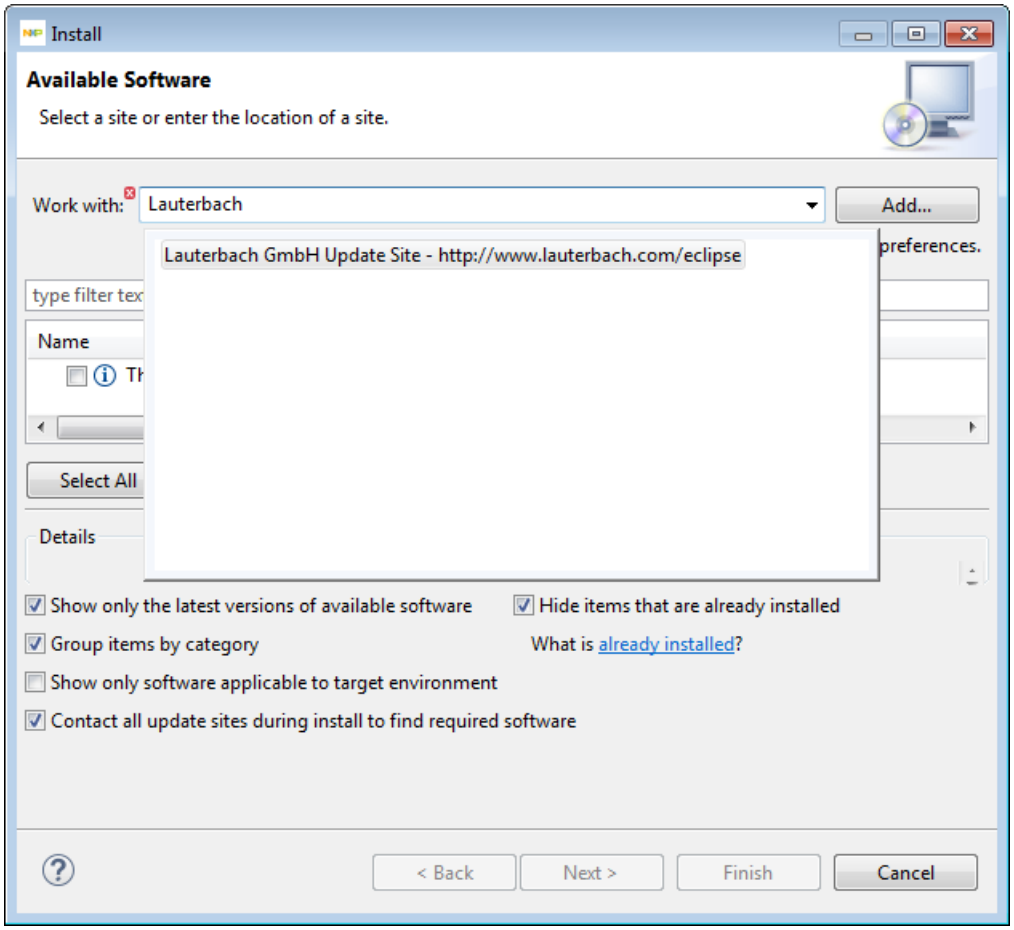
To start the debugging session by using the Lauterbach TRACE32 debugger:

1. On the **Help** menu, select **Install New Software...**

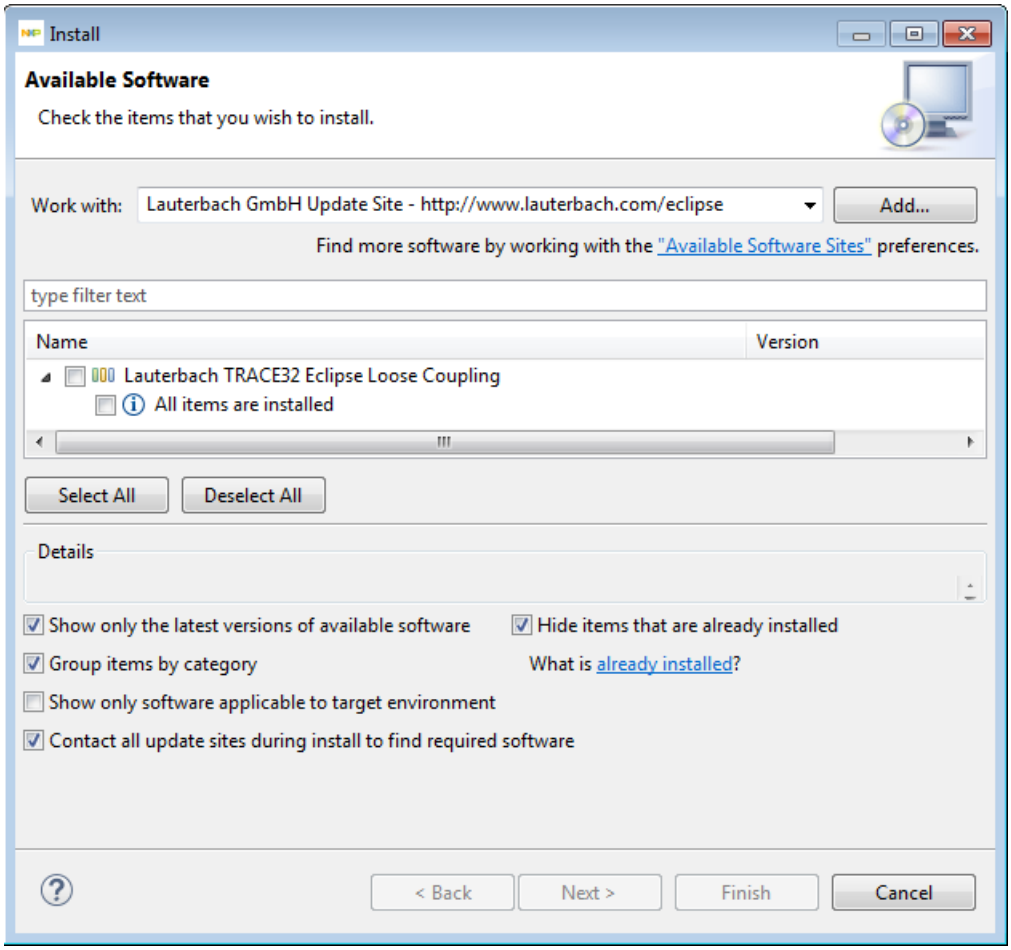
Install wizard opens.



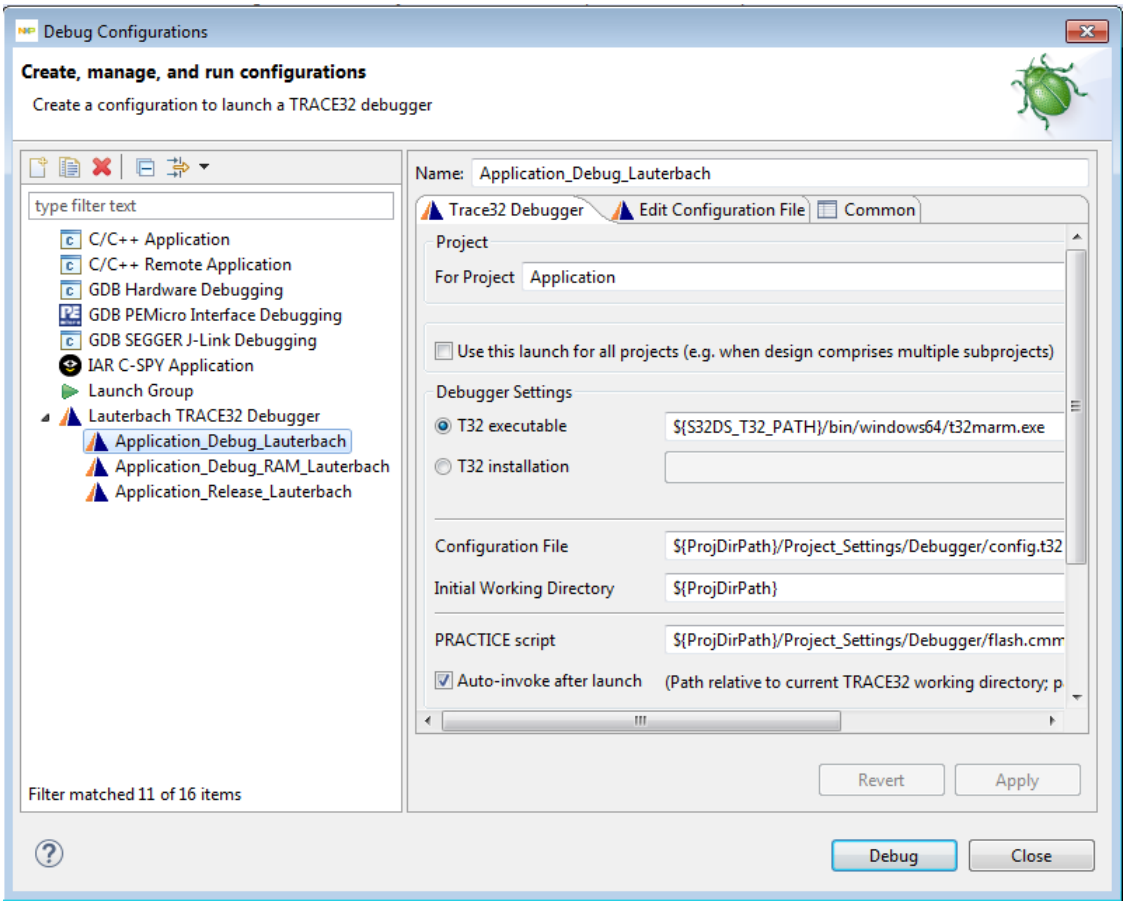
2. In the **Work with** field type: `Lauterbach` and wait while the wizard fetches the location of Lauterbach GmbH Update Site



3. Double-click on the link to the site.
4. In the list of installed software, make sure that there only the **All items are installed** item under the Lauterbach node, and then click **Cancel** to exist the **Install** wizard.



5. Make sure the target project has been created with support for Lauterbach TRACE32 debugger.
6. In the **Project Explorer** view, right-click the target project, and then, in the context menu, select **Debug As** > **Debug Configurations...**
   If Lauterbach TRACE32 is supported in this project, in the opened **Debug Configurations** window you will see the **Lauterbach TRACE32 Debugger** node.



7. Click **Debug**

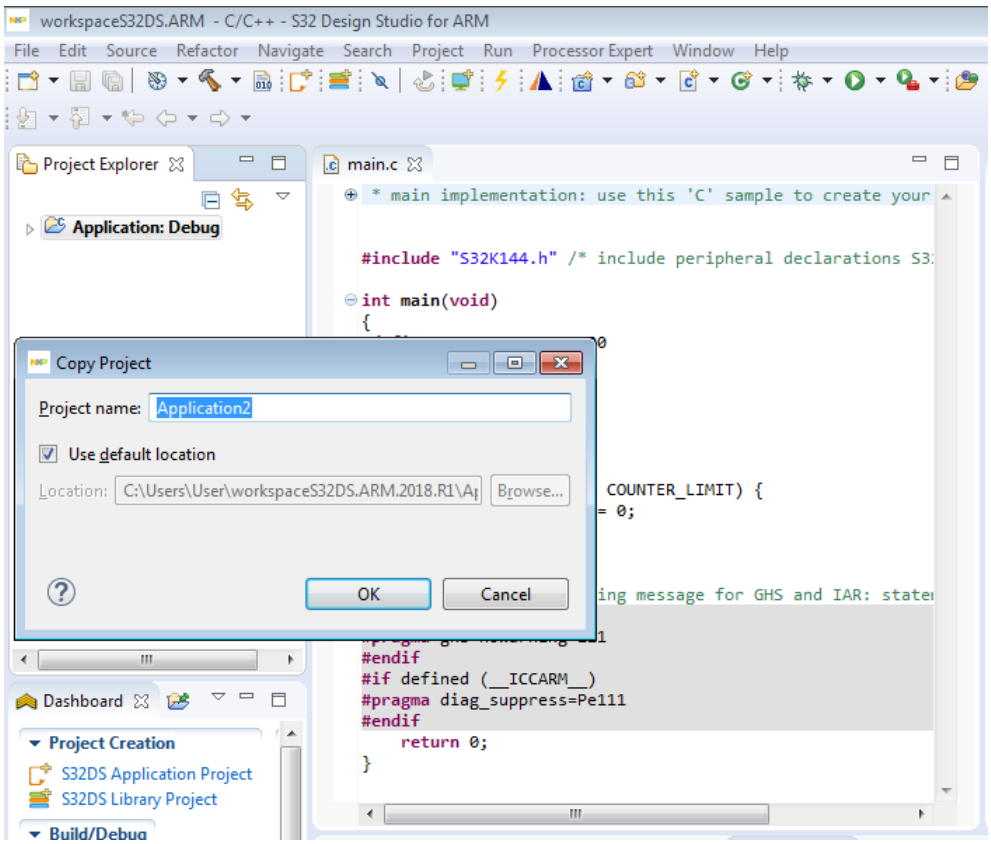
Lauterbach TRACE32 debugger starts and opens a new debugging session.

Duplicating a project

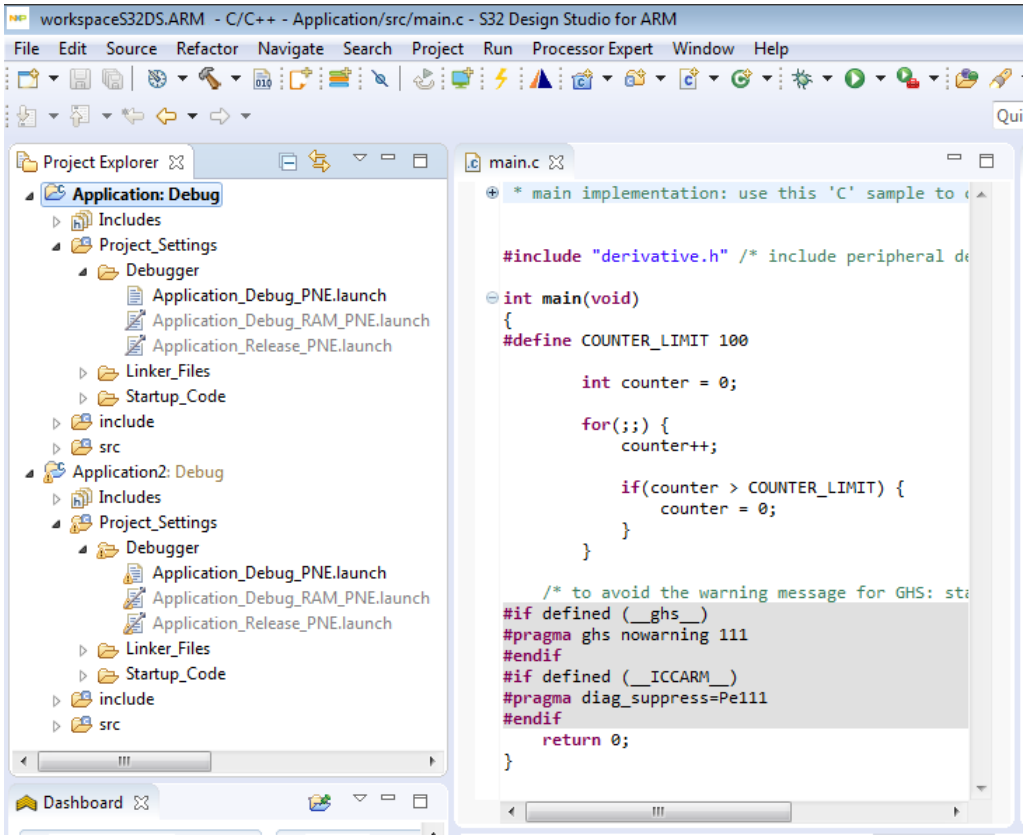
If you want to create a project based on an existing project, you can duplicate the project by using simple copy & paste operations right in the **Project Explorer** view. S32DS ARM v2018.R1 automatically handles renaming operations of the referenced resources such as launch configurations used in the project. This section explains how you can duplicate a project and debug the created duplicate.

To duplicate a project:

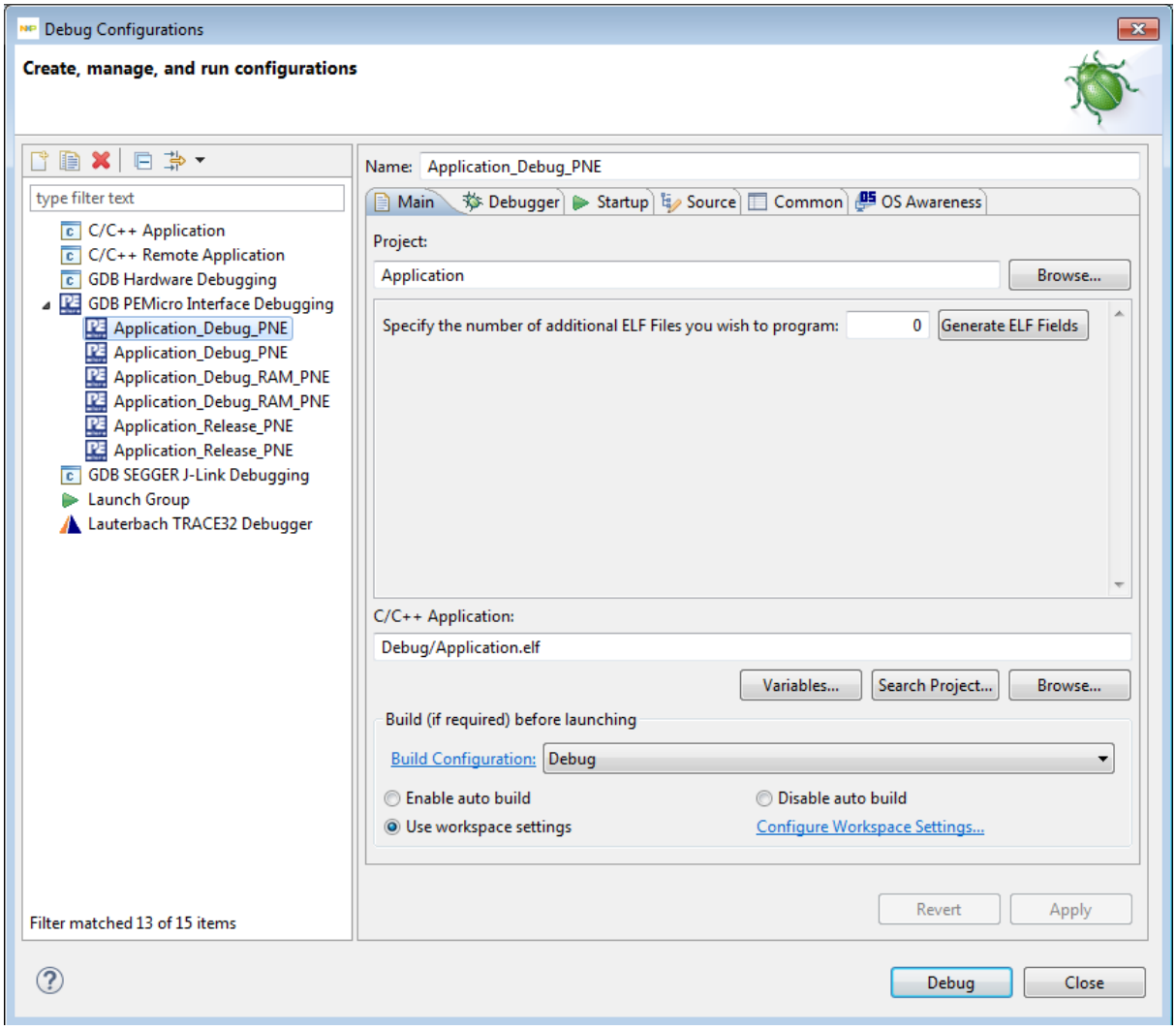
1. In the **Project Explorer** view, select the project that you want to duplicate and press **Ctrl+C**.
2. Press **Ctrl+V** to paste the copied project.
   S32DS ARM v2018.R1 automatically adds a postfix to the name for the copied project



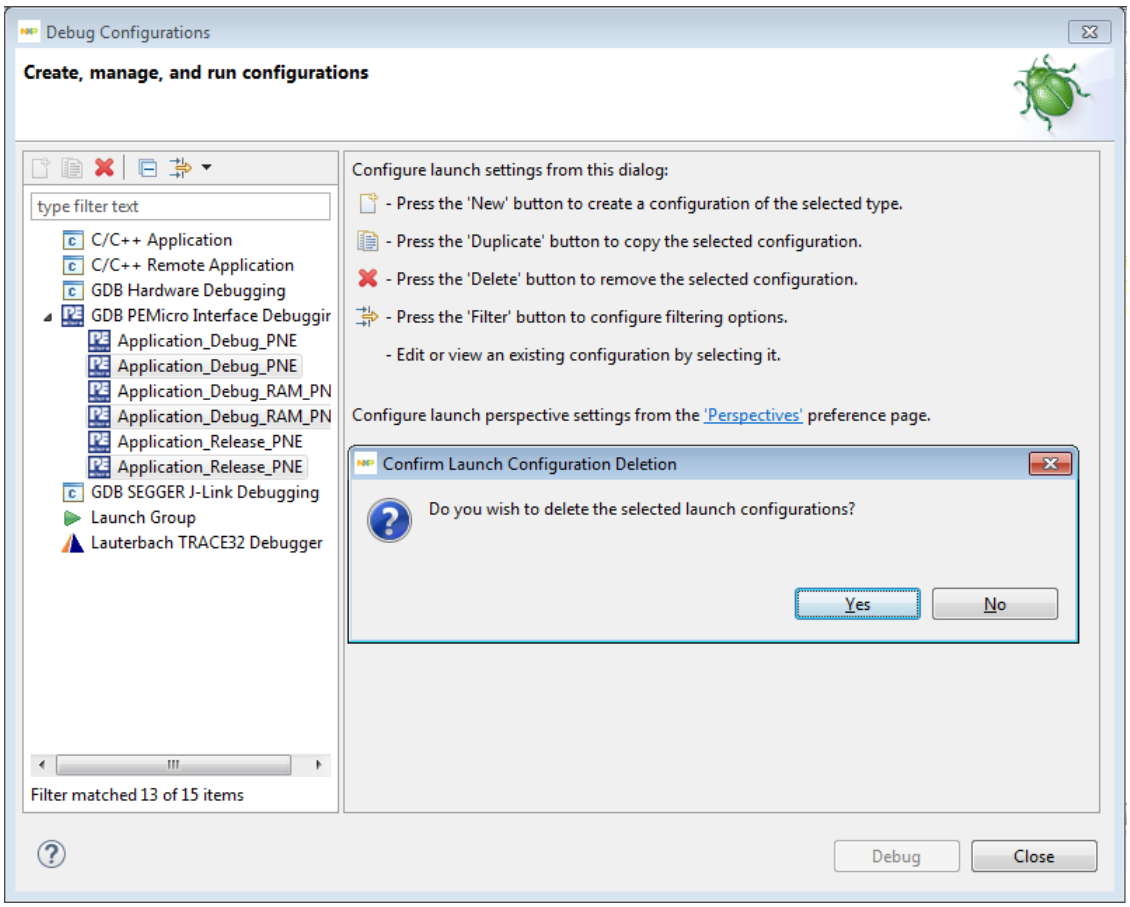
3. Confirm the new name by clicking **OK**.
   The copied project receives the warning sign over its folder indicating that you have to manually correct the launch configuration files.



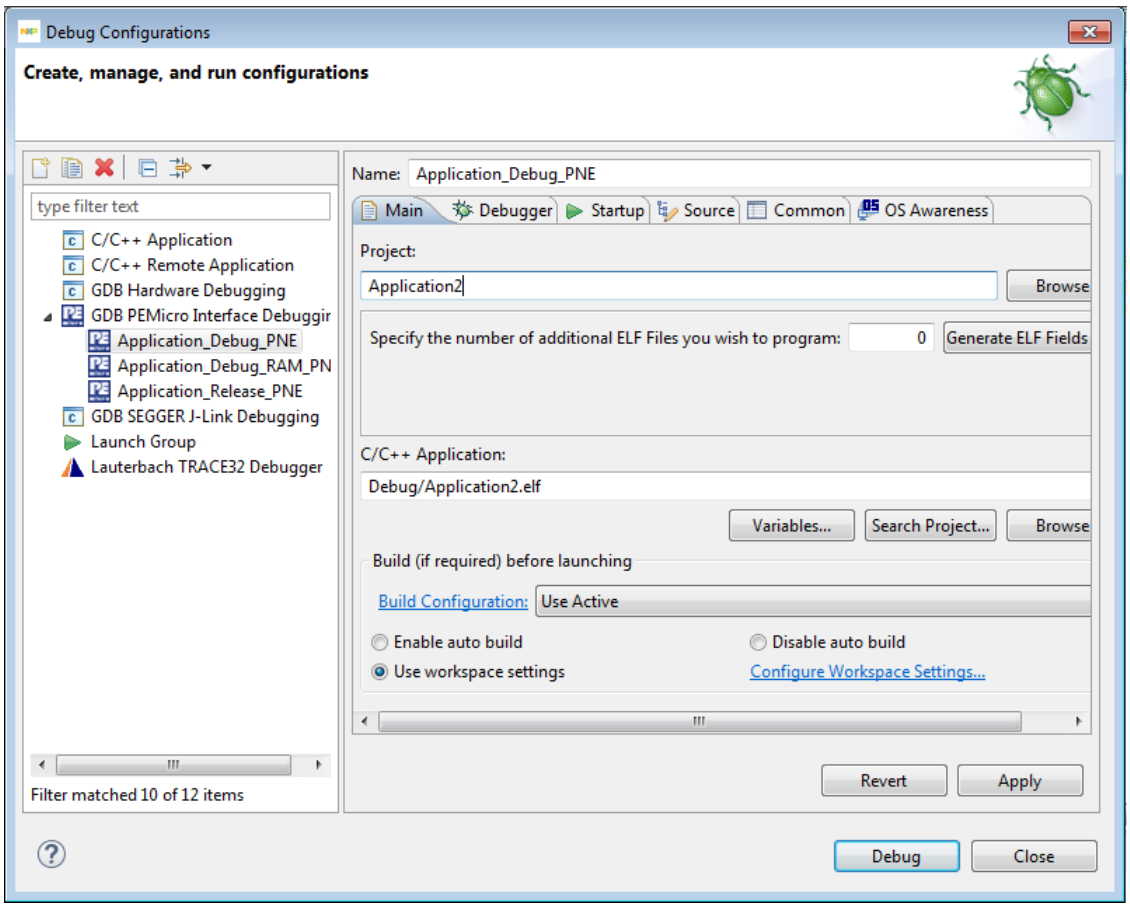
4. In the **Project Explorer** view, right-click the target project, and then, in the context menu, select **Debug As** > **Debug Configurations...**.
   Debug Configurations window opens, and configuration node for the debugger used in this project will have duplicate items of launch configurations.



5. Select each of the duplicate launch configurations by holding **Ctrl** as you click them, and then press **Delete**.
   The confirmation message box opens.



6. Click **Yes** to remove the duplicate launch configurations.
7. Click the **<project_name>_<build_configuration>_<debugger>** configuration.
8. On the **Main** tab, in the **Project** and **C/C++ Application** fields, update the name of the copied project.
   For example, if the source project has the name "Application", the default name automatically assigned to the copied project will be "Application2". Both fields will contain the name of the source project. The **Project** field will be set to `Application` and the **C/C++ Application** field will be set to `Debug/Application.elf`. In this case, type `Application2` in the **Project** field, and then type `Debug/Application2.elf` in the other field.



9. Click on the **Apply** button.
10. Repeat steps 7 though 9 for other launch configurations.

Now launch configurations in this debug configuration are ready, and you can start debugging the project.

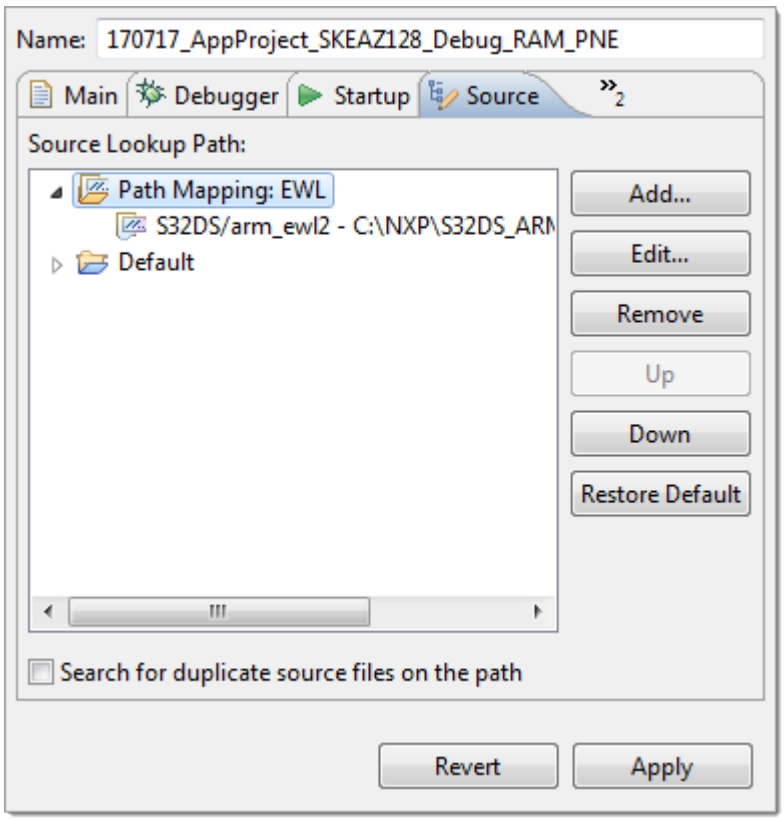
Specifying path mapping for a EWL project

Copied project may link to locations that only existed on the source computer. If you are reusing a previously created project that uses functions of EWL libraries, you may need to change the linked locations to the local ones.

For example, if you open a copied project that links to a EWL (Embedded Warrior Library), you have to update the path mapping for the EWL location. Doing this ensures that PnE and Segger debuggers work with the migrated project. Here is how.

To update a path mapping in a project:

1. Open the **Debug Configurations** of project.
2. On the **Source** tab, click the **Path Mapping: EWL** node, and then click **Edit...**



3. In the **Path Mappings** window click within the **Local file system path** column for the default EWL compilation path, and then, in the path, click the **(…)** button.
4. In the browse dialog specify the path to the EWL libraries, and then click **OK**.
5. Click **OK** to update the path, and then on the **Source** tab, click **Apply**.

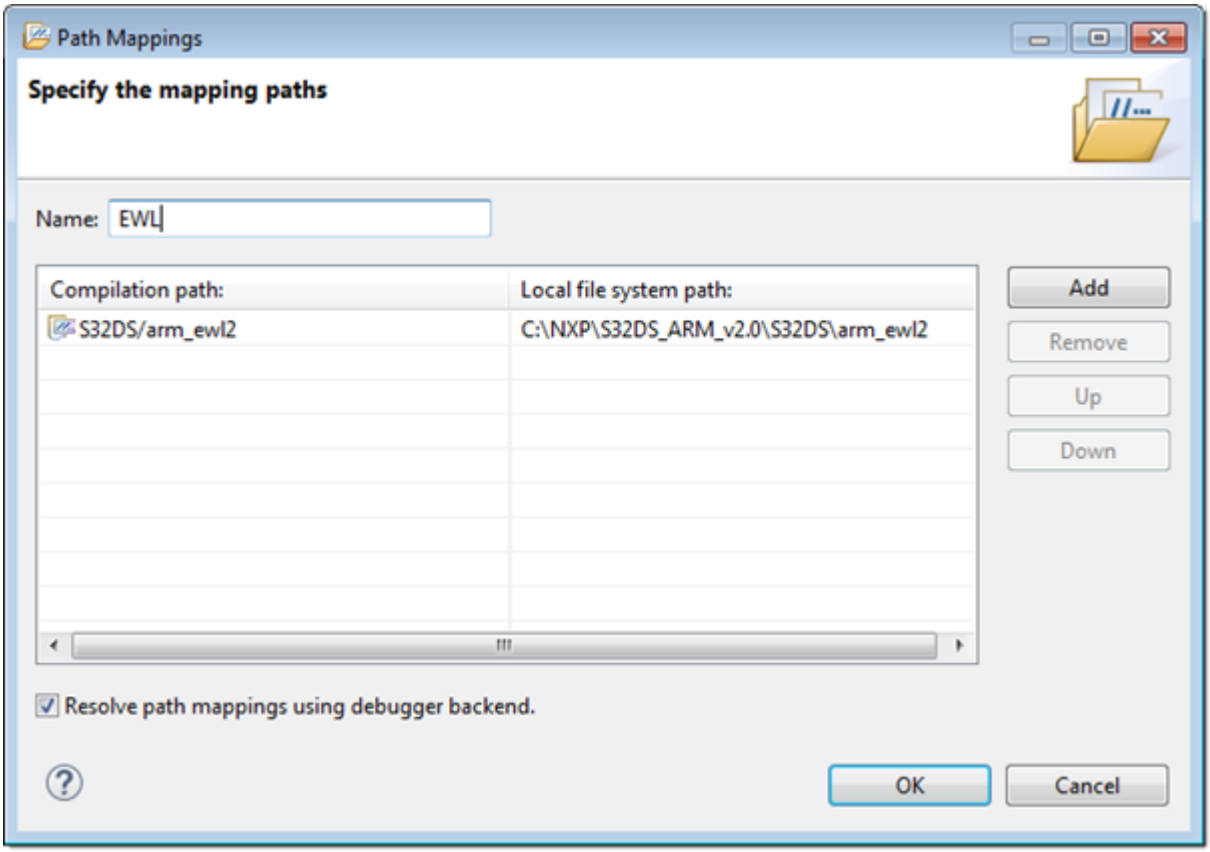
   Note: The IDE looks for the EWL libraries in the *S32DS/arm_ewl2* location. This is a virtual path, which is hardcoded in the IDE settings. By default, when you create an application project, the wizard maps this virtual path to the following location: *<S32DS_ProgramFolder>\S32DS\arm_ewl2* folder, where <S32DS_ProgramFolder> is the location where you have installed the S32DS ARM product. This is where source code for the EWL libraries and the libraries are stored by default. At the debug time, when the debugger attempts to load EWL by using *S32DS/arm_ewl2*, it will be redirected to the path specified in **Local file system path**.

   The **Resolve path mappings using debugger backend** check box is selected by default and allows you to specify that the path to EWL libraries has to be resolved by the debugger, and not the IDE. That is, the path handle will be created on the debugger side. For example, if you use macros to substitute for the compilation path, if this check box is selected, the macros will be resolved by the debugger.

   If you open an example file shipped with S32DS ARM, there are no existing mappings specified by default. You have to create a new mapping.

   To create a new path mapping for a project:

   a. Open the **Debug Configurations** of project.
   b. On the **Source** tab, click **Add…**, and then, in the **Add Source** window, double-click **Path Mapping**.
   c. In the **Path Mappings** window, specify the name of the new path mapping, for example, type *EWL*. Click **Add**. The insertion point jumps to the cell under the **Compilation path** column.



d. In the **Compilation path** column, type *S32DS/arm_ewl2*.
e. Click within the **Local file system path** column, and then, click the **(…)** button.
f. In the browse dialog specify the path to the EWL libraries, and then click **OK**.
g. Click **OK** to create the path, and then on the **Source** tab, click **Apply**.

Managing the search order for the debugger

You can control the order in which the debugger looks for the source code.

By default, the IDE searches for the source code locations in this order:

1. EWL path
2. Absolute files path
3. Path relative to the program.
4. Path relative to the application project.

Lookup locations are defined by search groups. The EWL location is configured by the **Path Mapping: EWL** group created by S32DS ARM. #ther locations are organized into the **Default** group and are defined by Eclipse.

You can manage the search order in which the IDE searches for the libraries and code at debug time by using the **Up** and **Down** buttons which become available when you click a search group.

Chapter 5

Multicore debugging

Topics:

S32DS ARM v2018.R1 allows to define multiple grouping of cores and perform multicore operations. Additionally, the chapter lists the steps to add multicore operations to S32DS ARM v2018.R1 through the UI.

The debugger in S32DS ARM v2018.R1 provides the facility to debug multiple processors using a single debug environment. The run control operations can be operated independently. A common debug kernel facilitates multicore, run control debug operations for examining and debugging the interaction of the software running on the different cores on the system.

Targeting core

The debugger connects to specific processor core through information provided in the **Debug Configurations** window. Specifically, the core index value on the **Main** tab of the **Debug Configurations** dialog box determines the core targeted for debug operations.

The core index value starts from “1”. That is, the first processor core has an index value of “1”, the second processor core has an index of “2”, and so on.

You can change this core index value on the **Main** tab > **Name** textbox, when you are modifying the settings of a Debug configuration.

Starting debugging session for core

To start debug for a specific core user need to connect the debugger to that core and start a debugging session. User can use the following methods:

- Start debug from **Debug Configurations** dialog box
- Start debug from **Run** menu
- Start debug from toolbar's **Debug** icon.

To start multicore debugging you should use the **Launch group** feature. First starts the main core, and then the sessions for the other cores start.

Debugging Specific Core

After you select the launch configuration and click **Debug**, the debugger downloads the program to the main core and the **Debug** perspective appears. Within the **Debug** view, the program’s thread appears. The thread is identified by its launch configuration name and the index value of the core that it executes on. If you are debugging source code, the program’s source appears in an Editor view.

To debug a specific core's program, click on its thread in the **Debug** view. The **Debug** perspective automatically displays the source, registers, and variables for this core. If you click on another thread, the **Debug** perspective updates all of the views to display that core's context.

Chapter 6

Connections

Topics:

This chapter describes the features and settings of the connections that interface the S32DS ARM v2018.R1 debuggers with the target board.

For the IDE to communicate with the target hardware, you must specify several key items:

- connection type
- port over which debug communications is conducted
- target device name being debugged
- connection parameters.

The connection type determines what debugger protocol the debugger uses to communicate with the target. You can select the connection type in the **New S32DS Project** wizard (**New S32DS Project for <processor name>** step > **Debugger** list).

The others connection settings configure options specific for the hardware probe. After you make the option for the connection type you can enter connection settings using options in the **Debugger** panel of the **Debug Configurations** dialog box .

GDB PnE Micro connection

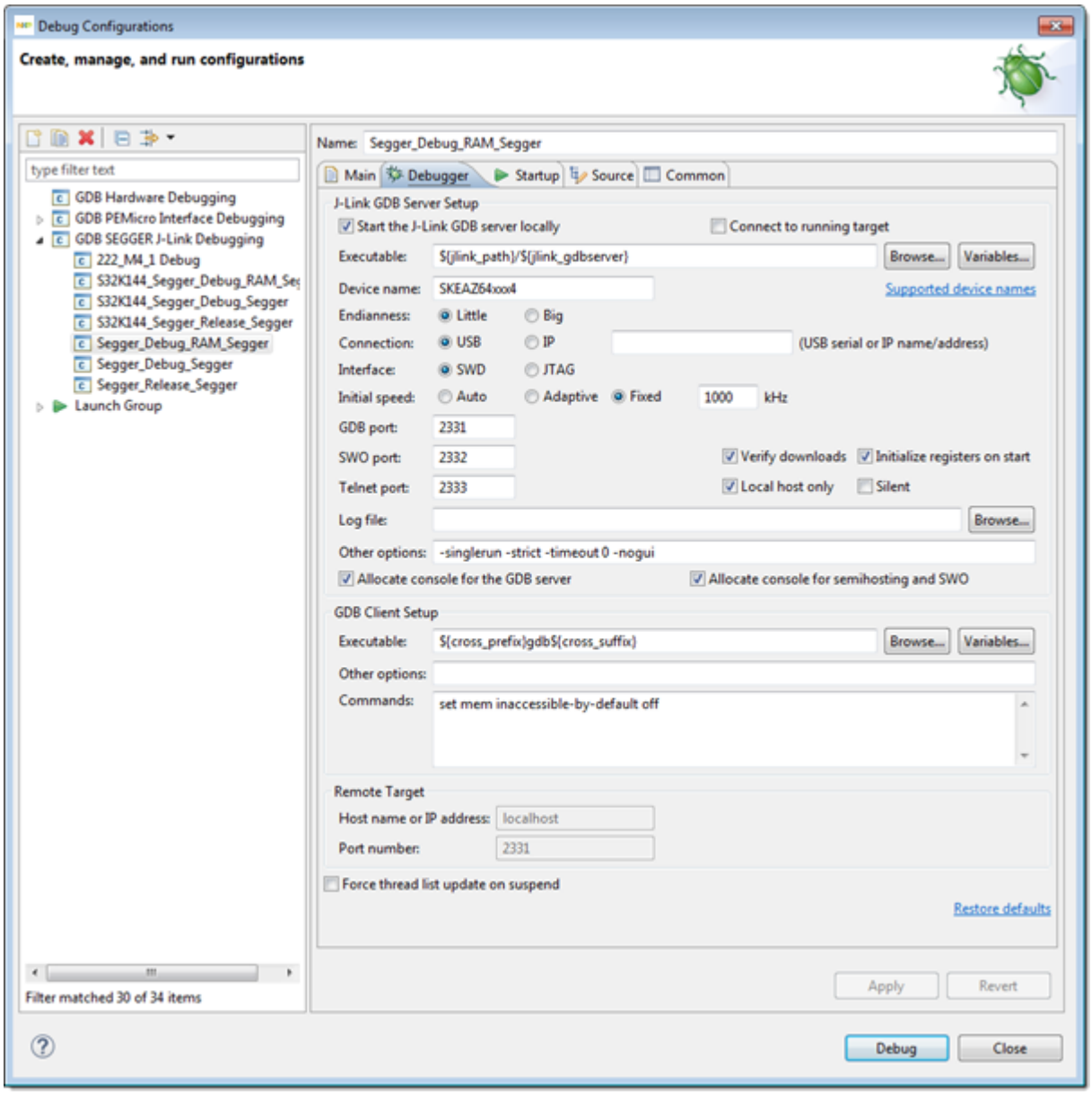
This chapter describes the features and settings of the PnE Micro connection that interfaces the debugger with the target board.

For the IDE to communicate with the target hardware, additionally you must select the interface type. The connection setting permits a connection to devices via following hardware interfaces:

- USB Multilink
- Embedded OSBDM/OSJTAG
- Cyclone
- TraceLink
- OpenSDA Embedded Debug

Note: More information about customizing connection parameters for GDB PnE Micro connection interface can be found in **PnE GDB Server Plug-In for ARM Devices. Debug Configuration User Guide**. The document can be found on the **Reference Manuals** page of S32DS ARM v2018.R1 Documentation Suite. The Documentation Suite is available when you open the **Documentation** link located under S32 Design Studio for ARM, Version 2018.R1 in the Start menu (on Windows platform) or on the desktop (on Linux platform).

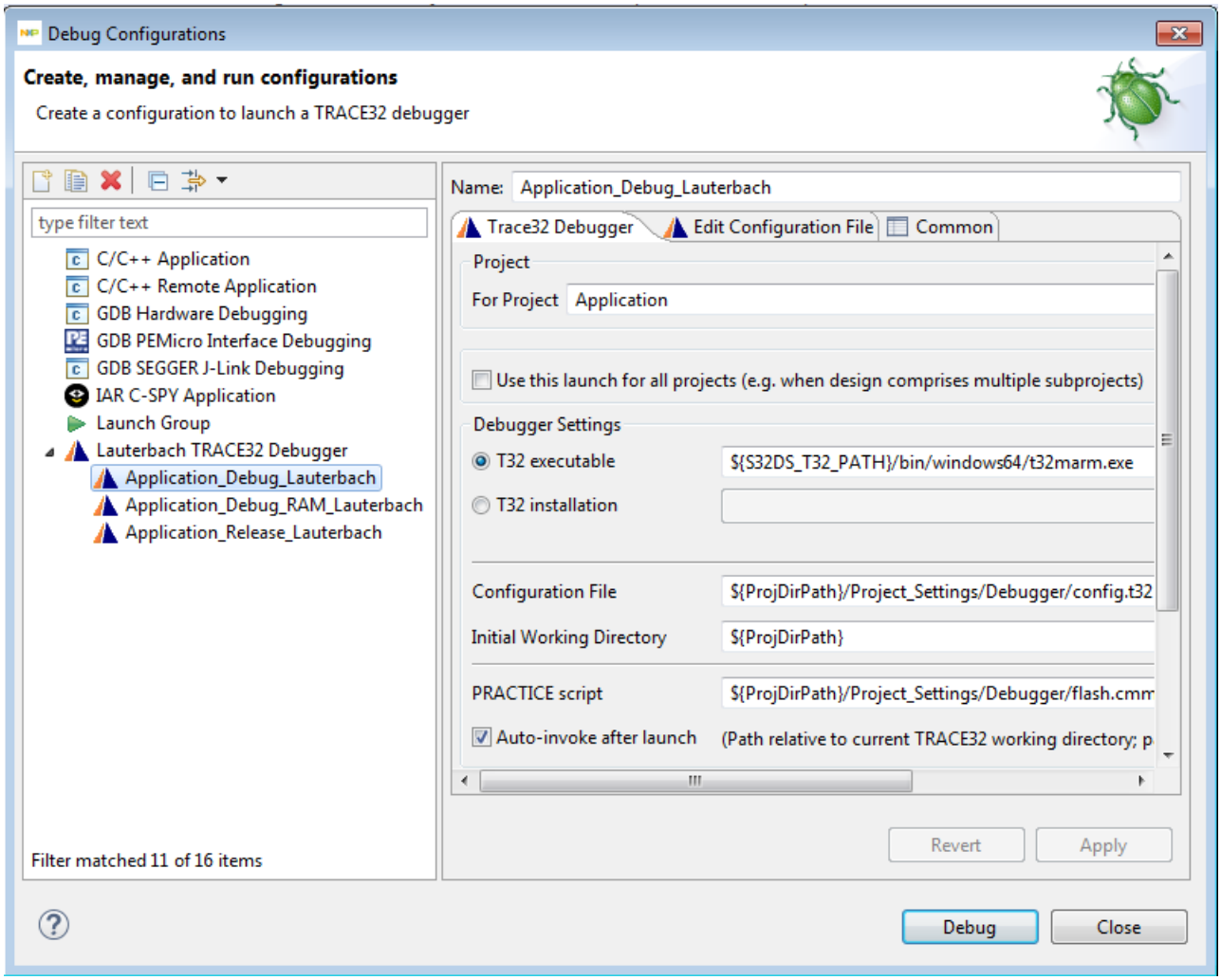
GDB SEGGER J-Link Connection



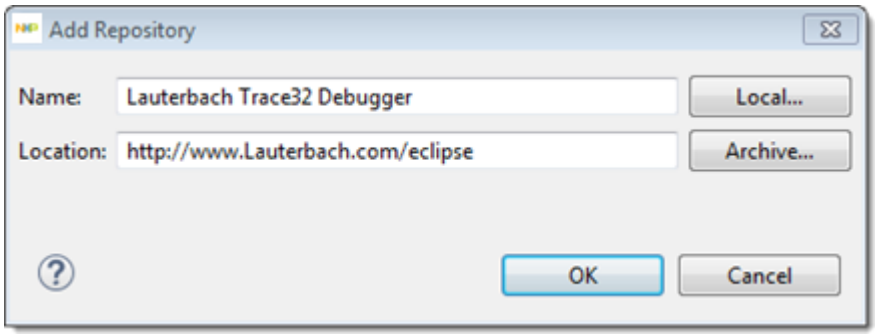
How to customize connection parameters for GDB SEGGER J-Link see **The J-Link debugging Eclipse plug-in** article.

Lauterbach connection

S32DS ARM v2018.R1 supports the Lauterbach connection.



The Lauterbach TRACE32 Eclipse plug-in should be installed in the S32DS ARM v2018.R1 to debug a project. Use the **Help** > **Install new software** menu to install it.



- How to install the Lauterbach Trace32 Eclipse plug-in see the article Installing the Lauterbach Trace32 Eclipse plug-in software:

  www.qnx.com/developers/docs/6.5.0/index.jsp?topic=%2Fcom.qnx.doc.ide.userguide%2Ftopic%2Fdebug_JTAGNeutInstTrace32pi_.html
- How to customize connection parameters for Lauterbach see Lauterbach web site:

  www.lauterbach.com

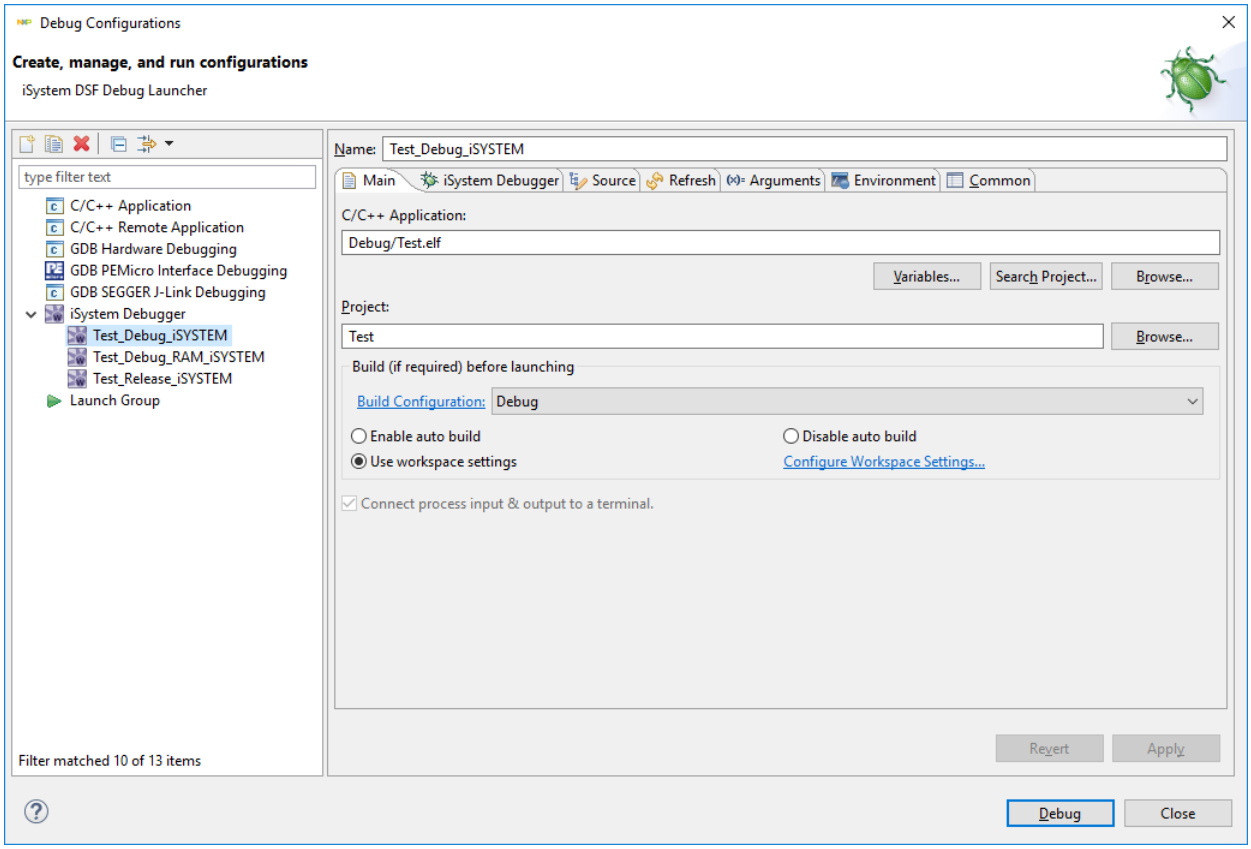
  and the article Coupling for Eclipse:

www2.lauterbach.com/pdf/int_eclipse.pdf

- How to start the debugging session for your project by using the Lauterbach TRACE32 debugger see Use the Lauterbach plugin for debugging.

iSystem connection

S32DS ARM v2018.R1 supports a target connection to iSystem.



Note: The iSystem Connection is not supported on Linux platform.

The following software should be installed in the S32DS ARM v2018.R1 to use the iSystem debugger:

- iSystem testIDEA version 9.12.273

  Download it from http://www.isystem.com/downloads/winIDEA/setup/winIDEA(x64)9_12_273.exe. Do not use the 9.12.256 version from the www.isystem.com site.
- iSystem Debugger Plug-in for Eclipse.

Use the **Help** > **Install new software** menu to install the plug-in. Download it from the software site: http://www.isystem.si/eclipseUpdate/debuggerJuno42/ , accept the license agreement and restart IDE.

- How to install the iSystem Debug Plug-in for Eclipse see the **Eclipse Plug-ins** web-page: http://www.isystem.com/download/eclipse
- For details on the usage read the **iSystem Debug Plug-in for Eclipse User's Guide**:http://www.isystem.com/downloads/SDK/eclipse/iSystem-EclipseDebugPlugin-UsersGuide.pdf.

Chapter 7

Working with SDKs

Topics:

This chapter describes usage of Software Development Kits (SDKs) in S32DS ARM v2018.R1.

SDK management

S32DS ARM v2018.R1 supports using the following types of SDK in the development of products:

- predefined SDK
- custom user-created SDK

The predefined SDKs are delivered as plugins. The following ways of delivery the sources of the SDKs are supported:

- packaged in the plugin
- provided as directory in the S32DS ARM v2018.R1
- link to GIT repository

SDKs delivered in the form of either source files or precompiled object files (libraries) can be integrated through the SDK manager. By using the SDK manager dialog you can define SDK parameters or the plugin (when using predefined SDKs) and add a user library. You can move a custom SDK between computers and reuse them in multiple installations of S32DS ARM v2018.R1or its workspaces.

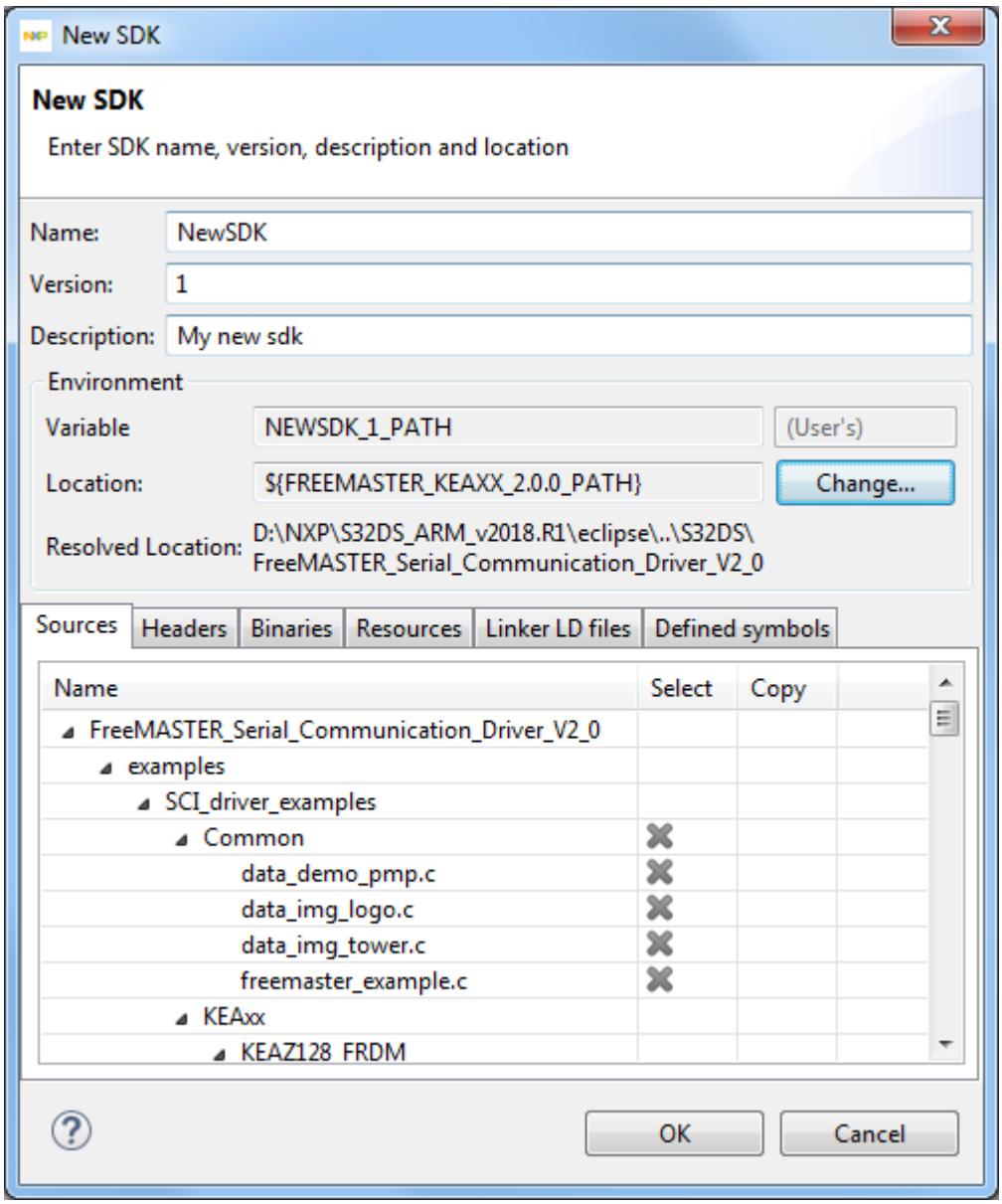
To use an SDK in your S32DS ARM v2018.R1 project, add the SDK to this project. You can do that when creating a new project by using the **New S32DS Application Project** wizard, or you can add the SDK to an existing project by editing project settings. When a SDK added to project the SDK Explorer displays information about modules added to the project.

S32DS ARM v2018.R1 performs the following actions when you add an SDK to the project:

- links configured source files to the project
- copies files to the project (if it is specified in configuration)
- specifies the location of include files in toolchain settings
- adds preprocessor symbols to toolchain settings
- adds reference to the library object file
- updates libraries (-L) paths in compiler and linker options.

If you detach SDK from project, all settings and linked files listed above are removed from the toolchain settings/ project. Only files that were specified as copied are stayed in project structure. You can manually remove the remnants from the project.

Following dialog window allows you to specify new or edit existing SDK settings:



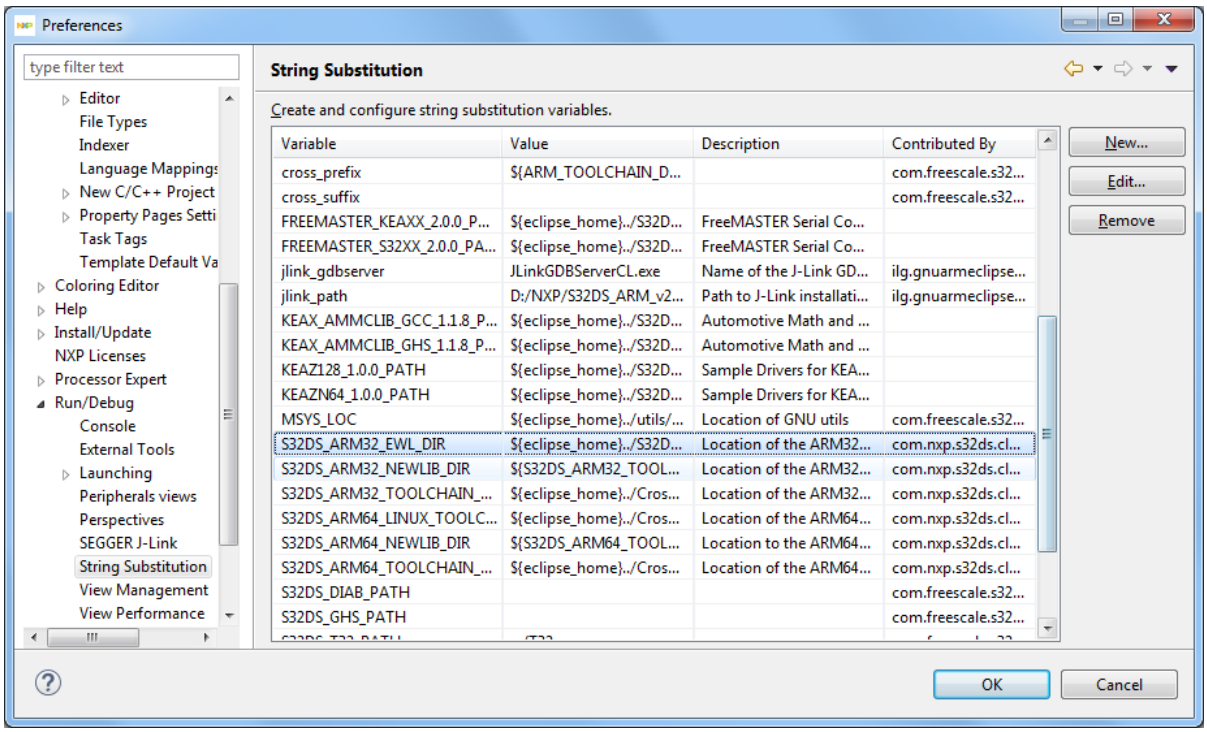
The SDK manager provides the user with the following information about SDK:

- name
- version
- description
- root directory
- list of header files
- list of source files
- list of library/object files
- list of other files
- information about symbols to be defined for preprocessor.

The information for the user defined SDKs can be edited, while for predefined (provided as plugins) –can be viewed only.

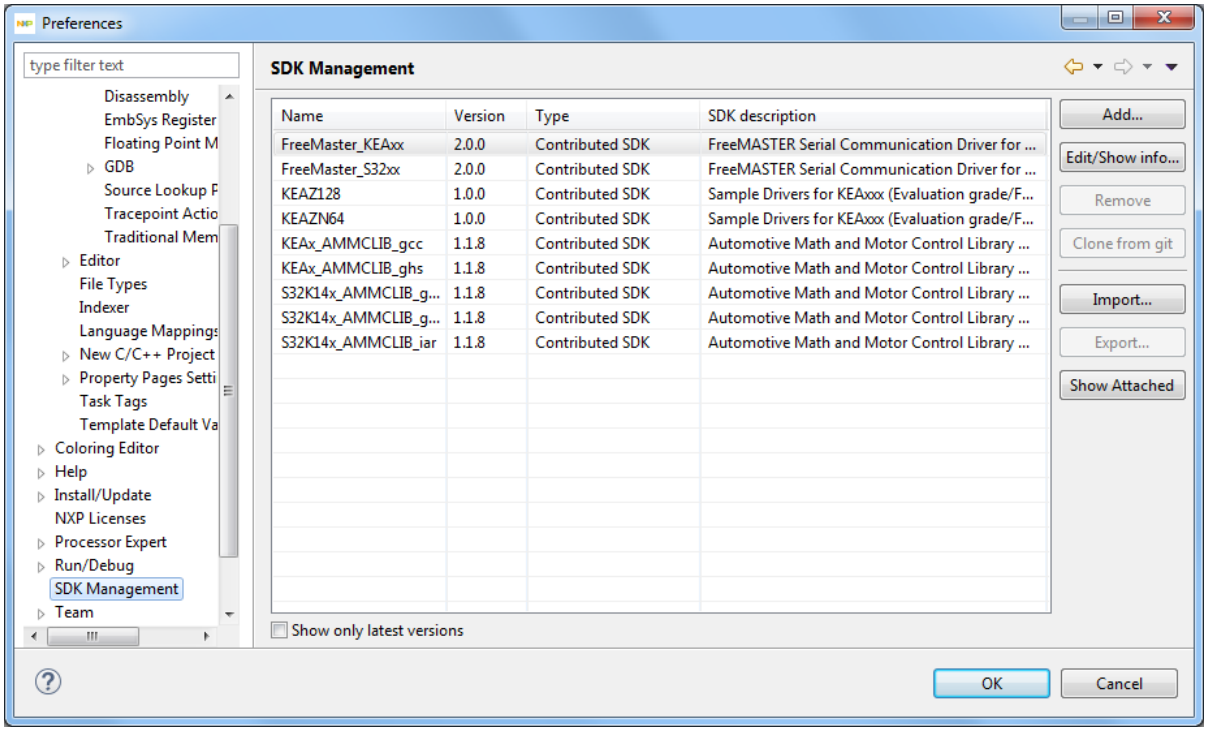
The list of files is created by scanning the root directory specified as location. The user can specify files to include to project by linking, or copying to the project.

SDK location can be edited using preferences page - **Run/Debug** - **String Substitution**:



SDK manager in workbench preferences

The **SDK management** page in global preferences allows to manage SDKs (add and edit or remove not built-in libraries) and import or export SDKs.

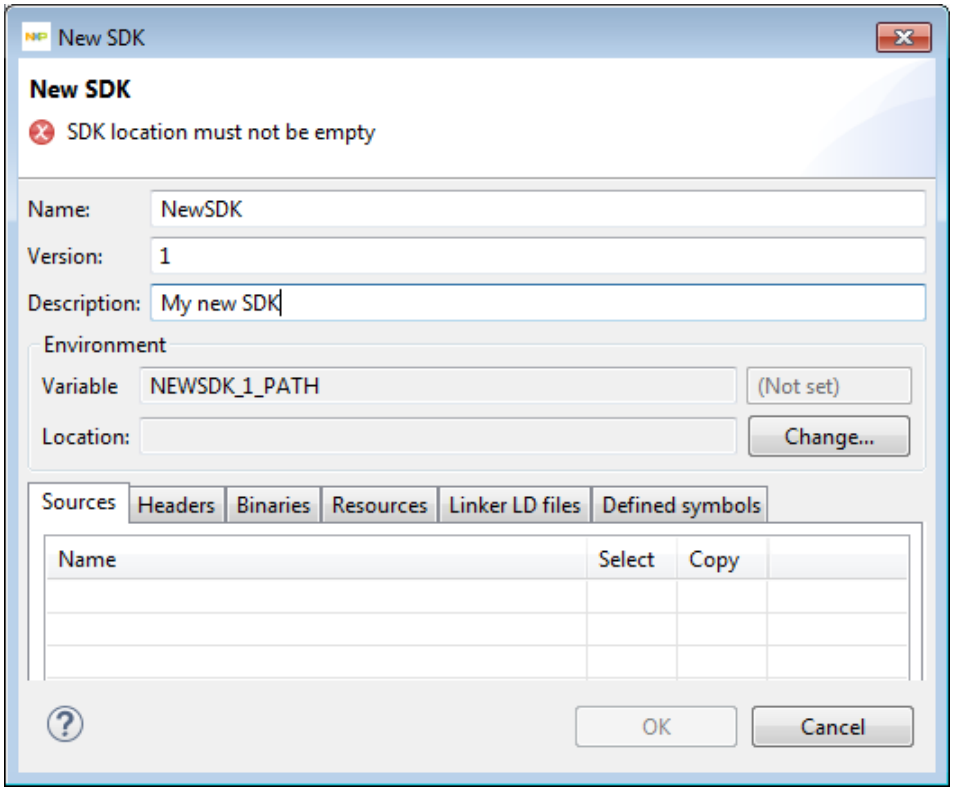


Adding custom SDK to Studio

When you create a new project, you can only choose from existing SDK shipped with S32DS ARM v2018.R1. If you want to add a custom SDK, you can edit S32DS ARM v2018.R1 preferences and add the SDK files there. Files added in preferences are visible to all projects in the **Project Explorer** view.

To add a custom SDK:

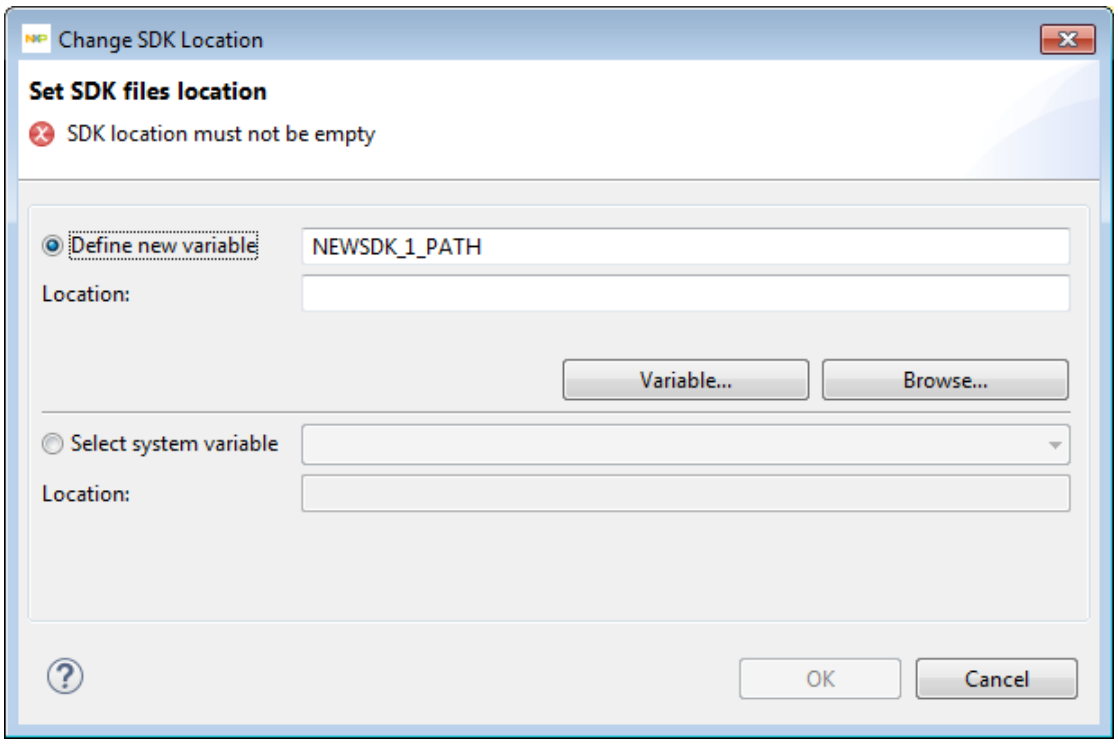
1. In the main window, select **Window** > **Preferences** > **SDK Management** in the menu bar.
2. Click **Add...**.
   The **New SDK** dialog window opens.
3. Define Name, Version and Description of your new SDK:



- **Name** - A valid C identifier. Must start with a letter. Allowed characters: letters, digits, and underscore.
- **Version** - Allowed characters: letters, digits, underscore, and period.
- **Description** (optional).

Note: Combination of name and version must be unique among other SDKs in the system because it is used for constructing Environment variable name.

4. To set location of environment variable click **Change...** button. You can either create new variable, use internal or system environment variable. Using internal variable allows you to share SDKs with other people or create SDKs to distribute widely.
5. In appeared Change SDK Location window you can Define new variable or Select system variable.

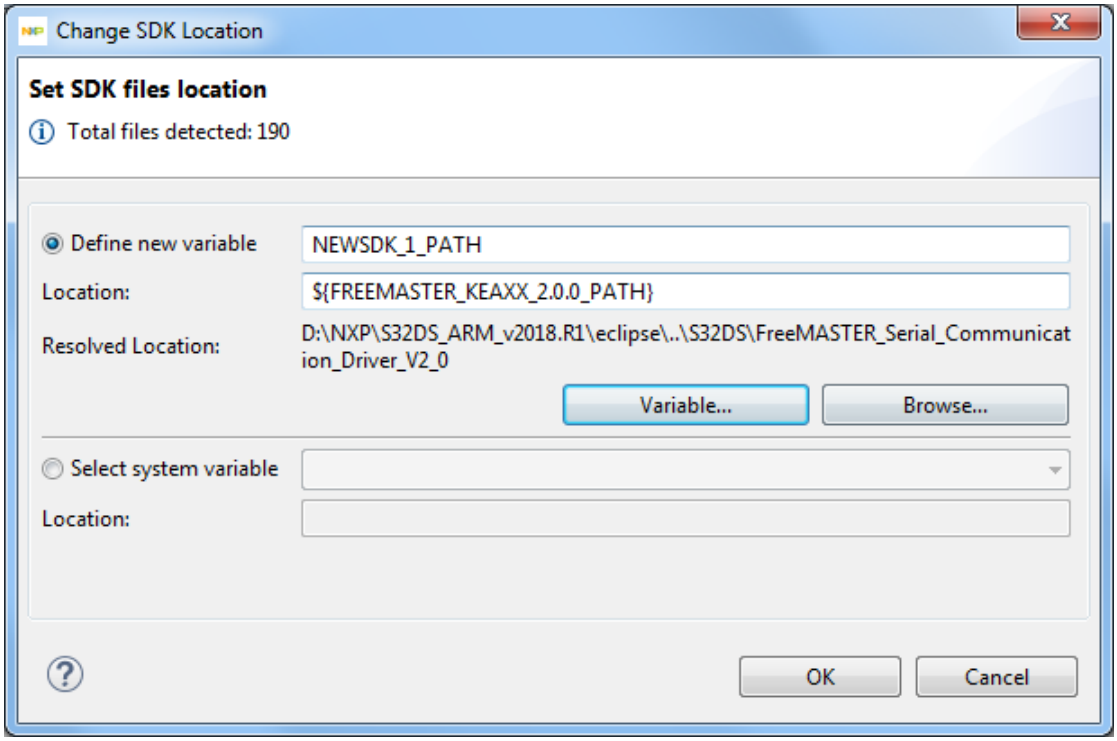


a) If you want to Define new variable, you should define the directory where SDK files are located by one of following ways:

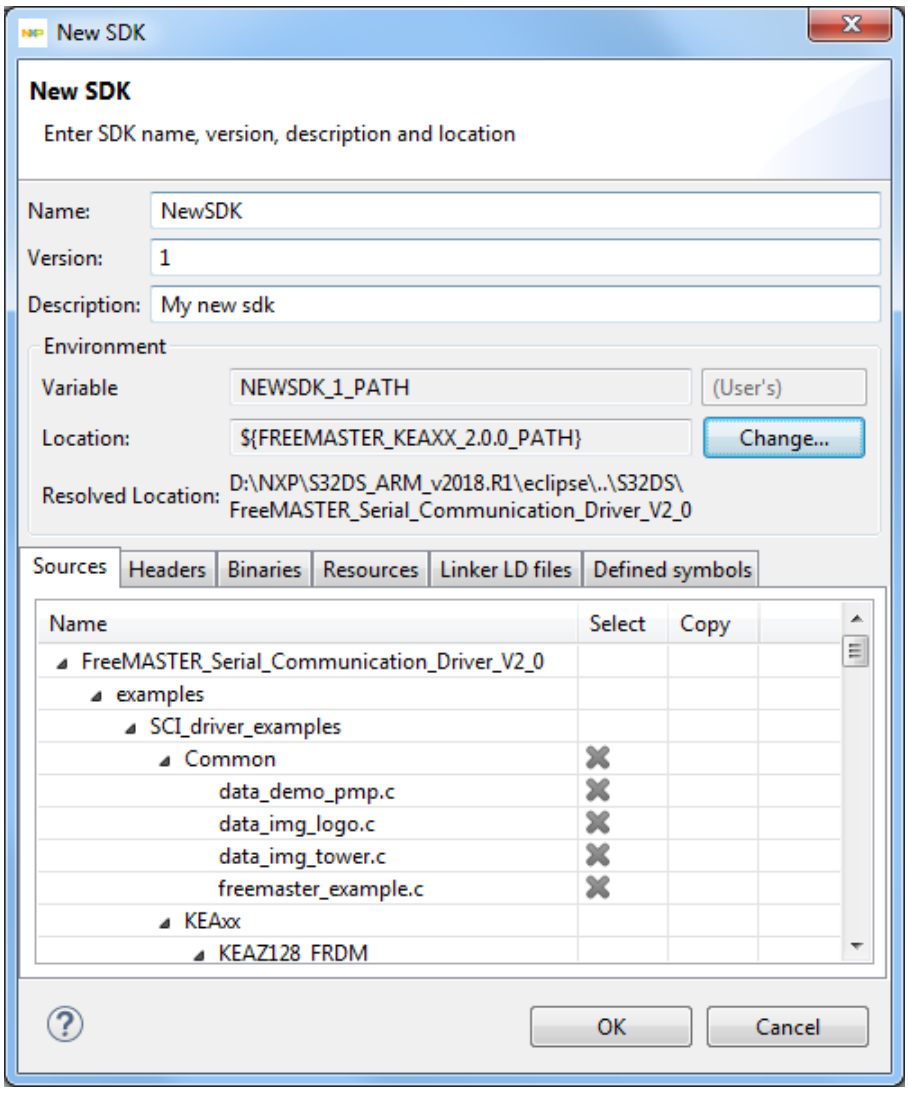
- Click **Browse...** and select folder where SDK files are located.
- Define path in Location field. Path may be defined as canonical file system path or as a reference to internal variable that points to SDK folder in file system.
- Click **Variable...** to use internal variable.

b) If you want to Select system variable, choose it from drop-down list.

After setting location of environment variable you can see Resolved location:



6. You can view and manage: Sources, Headers, Binaries, Resources and Linker LD files:



- Sources - usually **.c** and **.cpp** files.
- Headers - usually **.h** and **.hpp** files.
- Binaries
- Resources - any files (documents, images, etc.)
- Linker ID files - **.ld** files

In each files category you can:

- Mark with ✚ files in "Select" column. In this case files will be linked to the project during SDK attaching procedure.
- Mark with ✚ files in "Copy" column. In this case files will be copied to the project during SDK attaching procedure.

✖ - default mark.

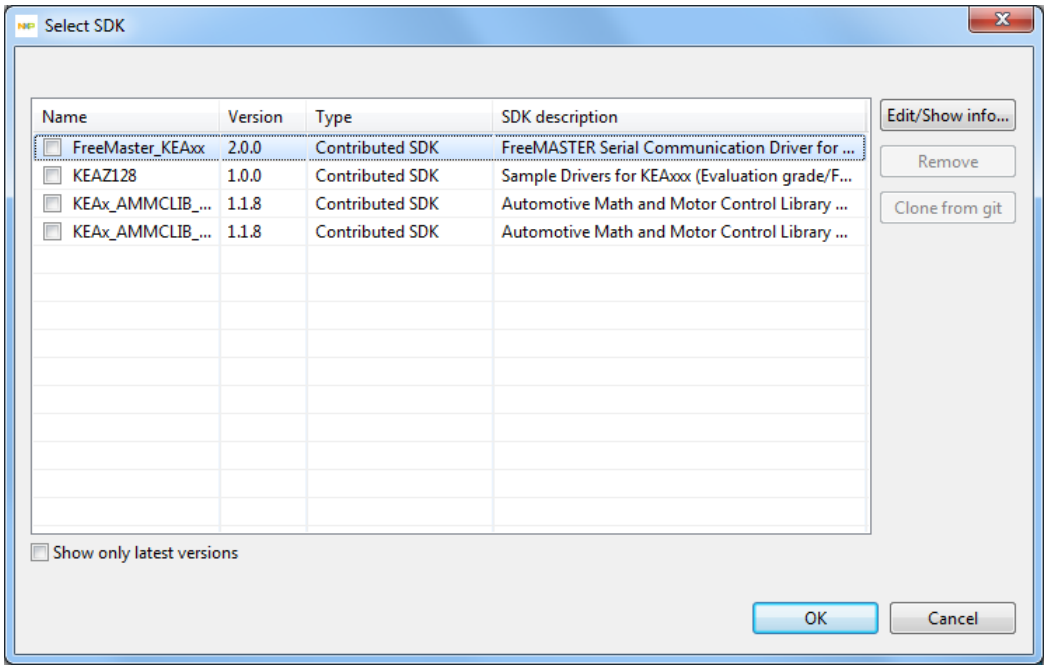
Note: When read-only (contributed/GIT) SDK is shown in this interface, its files are grouped not by extensions, but only according to SDK description. So, if “a.exe” is somehow defined as “<headerFile>”, it will be shown in Headers tab.

In Defined symbols tab you can view and manage information about symbols to be defined for preprocessor.

After SDK is created, descriptor of SDK is stored (in project properties for Project-local SDK or in Eclipse preferences for Global SDK) and SDK can be attached to project. New SDK is added to list of SDKs.

Selecting SDK in New S32DS Project wizard

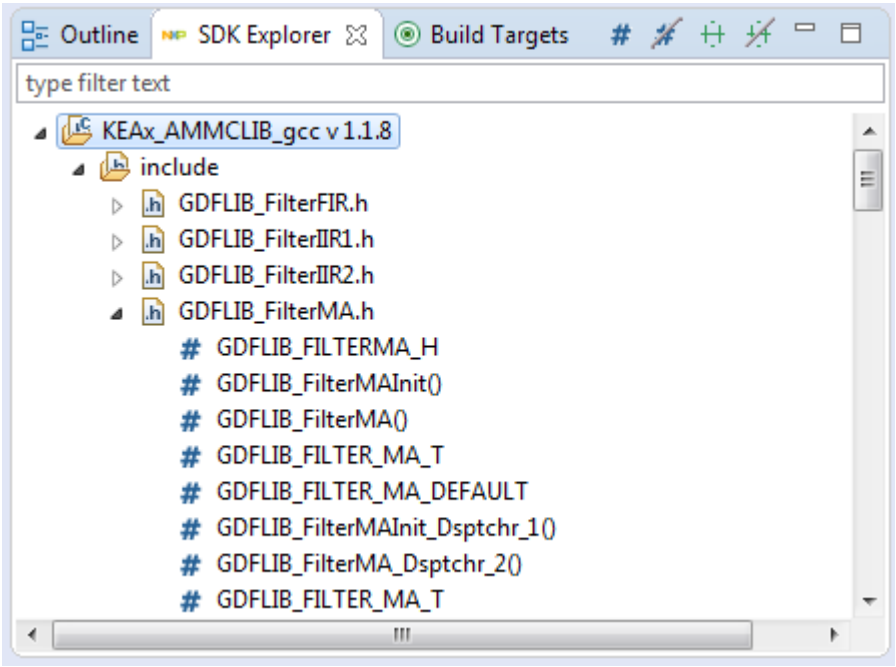
The second step of the **New S32DS Project** wizard contains a control **SDKs** to select SDKs for project. In the **Select SDK** dialog user can select SDKs to attach to newly created project.



The set of SDKs depends on the selected processor/ toolchain/ core, so only modules intended for the device should be available. The user can include any SDK – predefined (**Contributed SDK**) or user's created (**Local**) - into a project.

SDK Explorer

The **SDK Explorer** view shows SDKs attached to a project selected in the **Project Explorer**. **SDK Explorer** view displays information from the header files with defines and functions.



User can use the drag-and-drop possibility to drag a function from **SDK Explorer** into a source code in editor. When a function is dragged to a source file the **SDK Explorer** inserts the **include** statement for corresponding header file at the beginning of the file and adds the function call to the source with names at the place of parameters. If the function has return – it is written as **return_type = function_name (first_param_type, second_param_type)**;

For example for the function with prototype:

flexcan_status_t FLEXCAN_HAL_Enable(CAN_Type * base)

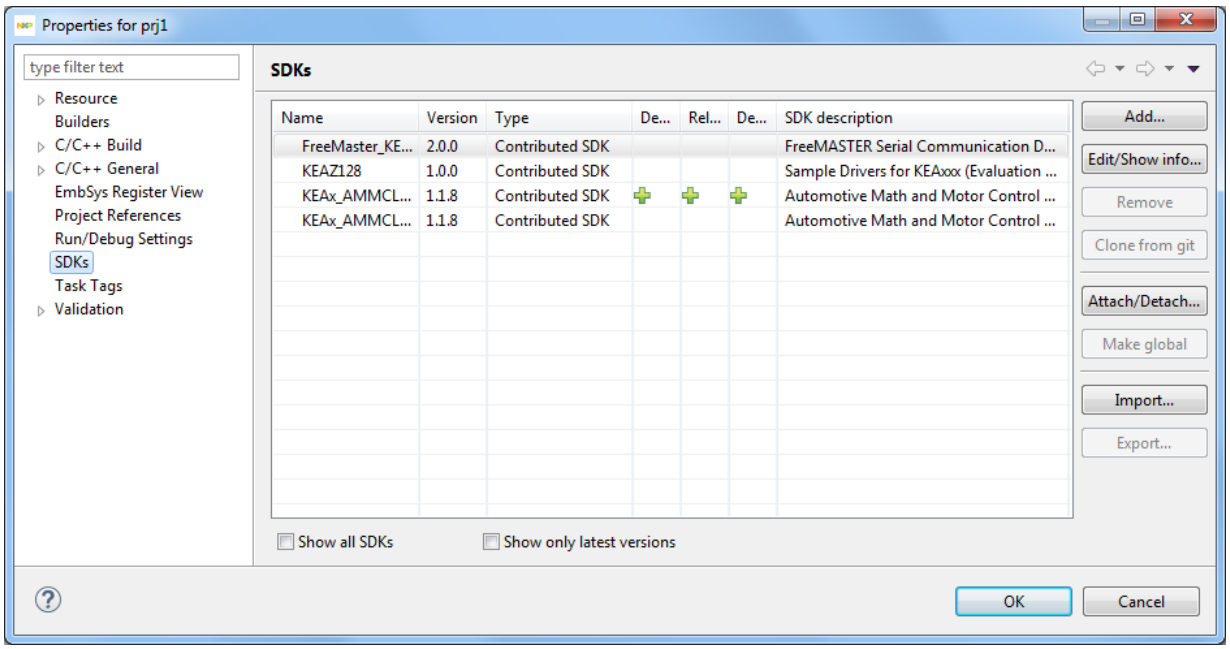
the following statement is added to source file (when it is dropped):

flexcan_status_t = FLEXCAN_HAL_Enable(CAN_Type *);

When a **"define"** is dragged to the source file – it is added as is, without closing semicolon.

SDKs property page

SDKs property page in a project properties allows to attach\detach SDKs to the project. Include (-I) and libraries (-L) paths are automatically updated in compiler and linker options according to currently attached libraries:



The SDKs project property page displays information about all compatible SDKs which are defined within S32DS ARM v2018.R1 – either installed with plugin (preconfigured) or user's defined.

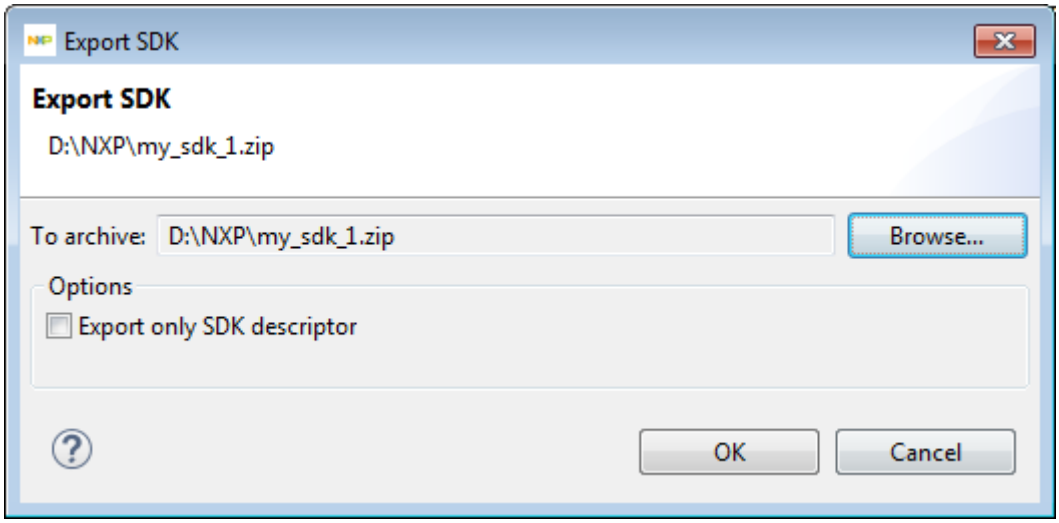
The SDKs list is filtered according the following criteria:

- supported compiler(s)
- supported language (C only or C/C++)
- supported architecture/core – if the SW module is independent from peripherals and depends only on core type
- supported device (core) – if SW module use some HW modules, then it could be used only for certain device (or core in case of multicores on this device).

Filtering can be switched off and user can select SDK by his choice.

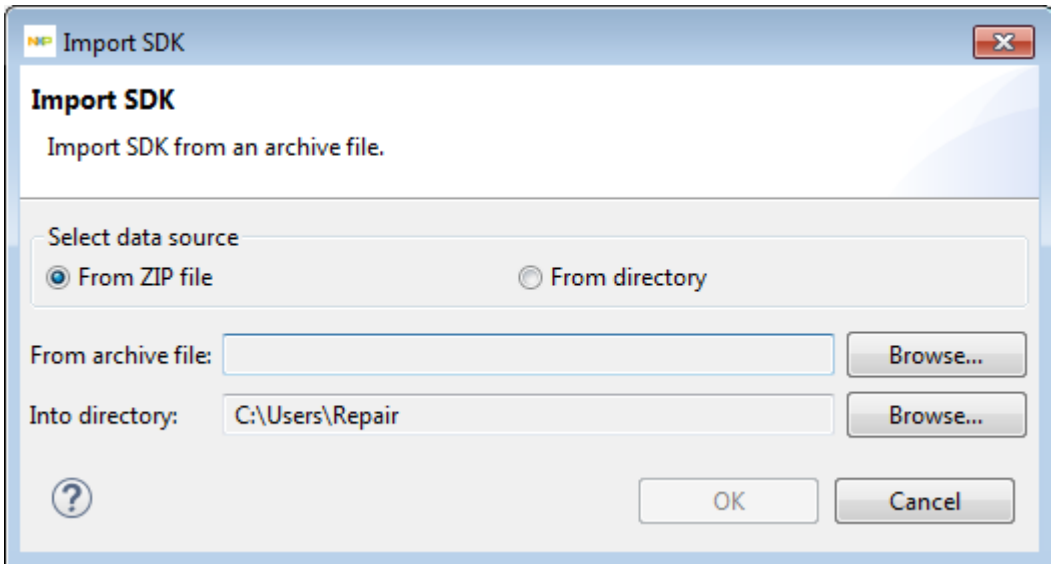
Import/export

User can export user-defined local SDK using the **Export…** button on the SDKs property page:



Enter or select the path to archive where to store SDK content or only xml-descriptor if the appropriate option is selected (the **Export only SDK descriptor** checkbox is selected).

Also SDK import is available from previously created archive file:



User defined SDK can be transferred. The predefined SDKs should be properly installed on different S32DS installation.

When SDK is exported the user might have possibility to include both SDK configuration file (XML-descriptor) as well as the header/source/library files, or only configuration file. The export creates zip-file.

When the SDK is imported from zip-file it requests location of header/source/library files:

- to link them (if only configuration was exported)
- to place them (if the whole SDK was included into export).

SDK delivered through GIT repository

The predefined SDK could have sources delivered through GIT repository, then instead of the files packaged in plugin or references in product layout a provided URL for GIT repository can be used as well as label. When the user start using this SDK he creates local copy of repository, defines location and provide credentials for GIT. The user cannot edit URL or label.

In case of GIT repository using the definition of file set and options is provided via manifest file. Manifest file is defined in the plugin configuration.

How to Reach Us:

Home Page:

nxp.com

Web Support:

nxp.com/support

Revision January, 2018