**TASKING**

# TASKING Embedded Debugger User Guide

# Table of Contents

# Chapter 1. Preparing for First Use

This chapter guides you through the installation process of the TASKING® Embedded Debugger. It also describes which documentation is available and how you best can use it.

In this manual, **TASKING Embedded Debugger** and **TASKING Debugger** are used as synonyms.

## 1.1. Installing the Software

This section describes the installation of the software for Windows. TASKING products are protected with TASKING license management software (TLM). To use a TASKING product, you must install that product and install a license.

### 1.1.1. Installation for Windows

#### System Requirements

Before installing, make sure the following minimum system requirements are met:

• Windows 7 or higher

• 2 GHz Pentium class processor

• 1 GB memory

• 3 GB free hard disk space

• Screen resolution: 1024 x 768 or higher

#### Installation

1. If you received a download link, download the software and extract its contents.

   - or -

   If you received an USB flash drive, insert it into a free USB port on your computer.

2. Run the installation program (**setup.exe**).

   *The TASKING Setup dialog box appears.*

3. Select a product and click on the **Install** button. If there is only one product, you can directly click on the **Install** button.

4. Follow the instructions that appear on your screen. During the installation you need to enter a license key, this is described in Section 1.1.2, *Licensing*.

5. When the installation asks to install the Eclipse IDE, you have two options:

   • Enable **Eclipse IDE** if you want to use the Eclipse Mars SR1 that is part of this installation.

- Disable **Eclipse IDE** if you intend to run the TASKING Embedded Debugger from within an existing Eclipse environment. See Section 1.1.3, *Installing the Software in an Existing Eclipse Environment*.

## 1.1.2. Licensing

TASKING products are protected with TASKING license management software (TLM). To use a TASKING product, you must install that product and install a license.

The following license types can be ordered from Altium.

### Node-locked license

A node-locked license locks the software to one specific computer so you can use the product on that particular computer only.

For information about installing a node-locked license see Section 1.1.2.3.2, *Installing Server Based Licenses (Floating or Node-Locked)* and Section 1.1.2.3.3, *Installing Client Based Licenses (Node-Locked)*.

### Floating license

A floating license is a license located on a license server and can be used by multiple users on the network. Floating licenses allow you to share licenses among a group of users up to the number of users (seats) specified in the license.

For example, suppose 50 developers may use a client but only ten clients are running at any given time. In this scenario, you only require a ten seats floating license. When all ten licenses are in use, no other client instance can be used.

For information about installing a floating license see Section 1.1.2.3.2, *Installing Server Based Licenses (Floating or Node-Locked)*.

### License service types

The license service type specifies the process used to validate the license. The following types are possible:

- **Client based** (also known as 'standalone'). The license is serviced by the client. All information necessary to service the license is available on the computer that executes the TASKING product. This license service type is available for node-locked licenses only.

- **Server based** (also known as 'network based'). The license is serviced by a separate license server program that runs either on your companies' network or runs in the cloud. This license service type is available for both node-locked licenses and floating licenses.

  Licenses can be serviced by a cloud based license server called "**Remote TASKING License Server**". This is a license server that is operated by TASKING. Alternatively, you can install a license server program on your local network. Such a server is called a "**Local TASKING License Server**". You have to configure such a license server yourself. The installation of a local TASKING license server is not part of this manual. You can order it as a separate product (SW000089).

The benefit of using the Remote TASKING License Server is that product installation and configuration is simplified.

Unless you have an IT department that is proficient with the setup and configuration of licensing systems we recommend to use the facilities offered by the Remote TASKING License Server.

### 1.1.2.1. Obtaining a License

You need a license key when you install a TASKING product on a computer. If you have not received such a license key follow the steps below to obtain one. Otherwise, you cannot install the software.

#### Obtaining a server based license (floating or node-locked)

• Order a TASKING product from Altium or one of its distributors.

  *A license key will be sent to you by email or on paper.*

If your node-locked server based license is not yet bound to a specific computer ID, the license server binds the license to the computer that first uses the license.

#### Obtaining a client based license (node-locked)

To use a TASKING product on one particular computer with a license file, Altium needs to know the computer ID that uniquely identifies your computer. You can do this with the **getcid** program that is available on the TASKING website. The detailed steps are explained below.

1.  Download the **getcid** program from http://www.tasking.com/support/tlm/download.shtml.

2.  Execute the **getcid** program on the computer on which you want to use a TASKING product. The tool has no options. For example,

    ```
    C:\Tasking\getcid
    Computer ID: 5Dzm-L9+Z-WFbO-aMkU-5Dzm-L9+Z-WFbO-aMkU-MDAy-Y2Zm
    ```

    *The computer ID is displayed on your screen.*

3.  Order a TASKING product from Altium or one of its distributors and supply the computer ID.

    *A license key and a license file will be sent to you by email or on paper.*

When you have received your TASKING product, you are now ready to install it.

### 1.1.2.2. Frequently Asked Questions (FAQ)

If you have questions or encounter problems you can check the support page on the TASKING website.

http://www.tasking.com/support/tlm/faq.shtml

This page contains answers to questions for the TASKING license management system TLM.

If your question is not there, please contact your nearest Altium Sales & Support Center or Value Added Reseller.

### 1.1.2.3. Installing a License

The license setup procedure is done by the installation program.

If the installation program can access the internet then you only need the licence key. Given the license key the installation program retrieves all required information from the remote TASKING license server. The install program sends the license key and the computer ID of the computer on which the installation program is running to the remote TASKING license server. No other data is transmitted.

If the installation program cannot access the internet the installation program asks you to enter the required information by hand. If you install a node-locked client based license you should have the license file at hand (see Section 1.1.2.1, *Obtaining a License*).

Floating licenses are always server based and node-locked licenses can be server based. All server based licenses are installed using the same procedure.

#### 1.1.2.3.1. Configure the Firewall in your Network

For using the TASKING license servers the TASKING license manager tries to connect to the Remote TASKING servers `lic1.tasking.com` .. `lic4.tasking.com` at the TCP ports 8080, 8936 or 80. Make sure that the firewall in your network has transparent access enabled for one of these ports.

#### 1.1.2.3.2. Installing Server Based Licenses (Floating or Node-Locked)

If you do not have received your license key, read Section 1.1.2.1, *Obtaining a License* before you continue.

1. If you want to use a local license server, first install and run the local license server before you continue with step 2. You can order a local license server as a separate product (product code SW000089).

2. Install the TASKING product and follow the instruction that appear on your screen.

   *The installation program asks you to enter the license information.*
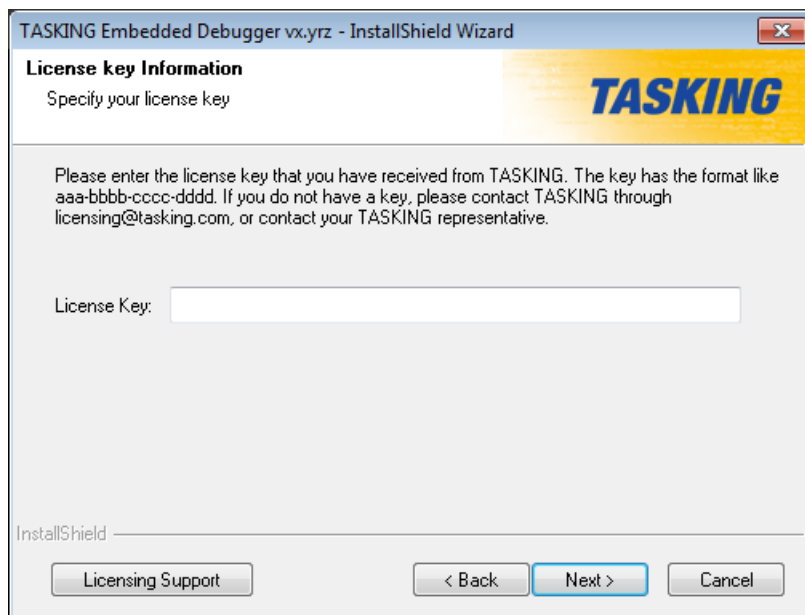
3.  In the **License key** field enter the license key you have received from Altium and click **Next** to continue.

    *The installation program tries to retrieve the license information from a remote TASKING license server. Wait until the license information is retrieved. If the license information is retrieved successfully subsequent dialogs are already filled-in and you only have to confirm the contents of the dialogs by clicking the **Next** button. If the license information is not retrieved successfully you have to enter the information by hand.*

4.  Select your **License type** and click **Next** to continue.

    *You can find the license type in the email or paper that contains the license key.*

5.  Select **Remote TASKING license server** to use one of the remote TASKING license servers, or select **Local TASKING license server** for a local license server. The latter requires optional software.

6.  (For local license server only) specify the **Server name** and **Port number** of the local license server.

7.  Click **Finish** to complete the installation.

### 1.1.2.3.3. Installing Client Based Licenses (Node-Locked)

If you do not have received your license key and license file, read Section 1.1.2.1, *Obtaining a License* before continuing.

1.  Install the TASKING product and follow the instruction that appear on your screen.

    *The installation program asks you to enter the license information.*
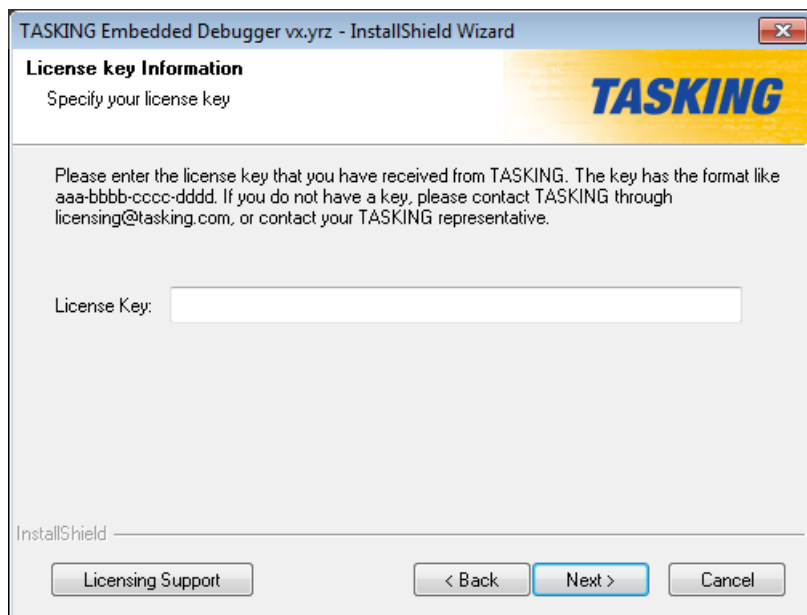
2.   In the **License key** field enter the license key you have received from Altium and click **Next** to continue.

   *The installation program tries to retrieve the license information from a remote TASKING license server. Wait until the license information is retrieved. If the license information is retrieved successfully subsequent dialogs are already filled-in and you only have to confirm the contents of the dialogs by clicking the **Next** button. If the license information is not retrieved successfully you have to enter the information by hand.*

3.   Select **Node-locked client based license** and click **Next** to continue.

4.   In the **License file content** field enter the contents of the license file you have received from Altium.

   *The license data is stored in the file licfile.txt in the etc directory of the product.*

5.   Click **Finish** to complete the installation.

## 1.1.3. Installing the Software in an Existing Eclipse Environment

If you have an existing Eclipse integrated development environment (IDE), you can add the TASKING Embedded Debugger software as plug-ins with the Install New Software feature.

### Requirements

• Eclipse Mars (SR1 or SR2) for Windows 32-bit

## Installation

1. First install the software as explained in Section 1.1.1, *Installation for Windows*. Since you want to install the software as plug-ins into an existing Eclipse environment, it is not necessary to select the Eclipse IDE feature. So, you can disable **Eclipse IDE** in step 5.

2. Start your existing Eclipse Mars (**eclipse.exe**).

   *The Workspace Launcher dialog appears.*

3. Enter the path to the workspace.

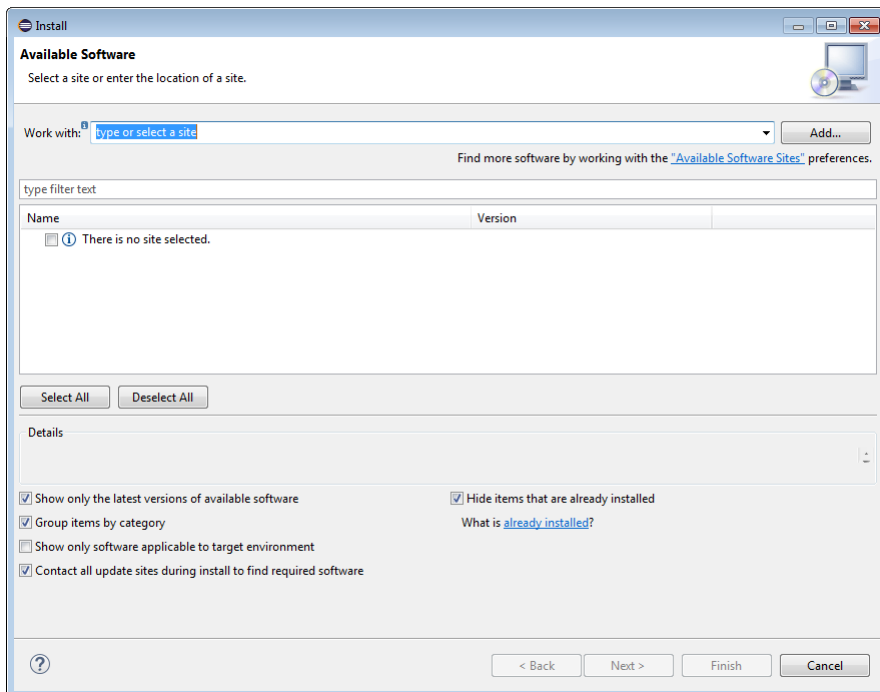   *In the remainder of this manual, we assume you use the default.*

4. Enable the option **Use this as the default and do not ask again**.

5. Click **OK** to proceed.

   *The Eclipse IDE opens.*

6. From the **Help** menu, select **Install New Software**.

   *The Available Software dialog appears.*



7. Click **Add**.

   *The Add Repository dialog appears.*
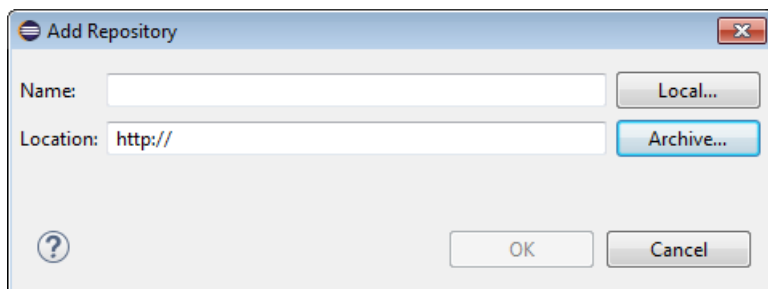
8. Click **Archive**.

   *The Repository archive dialog appears.*
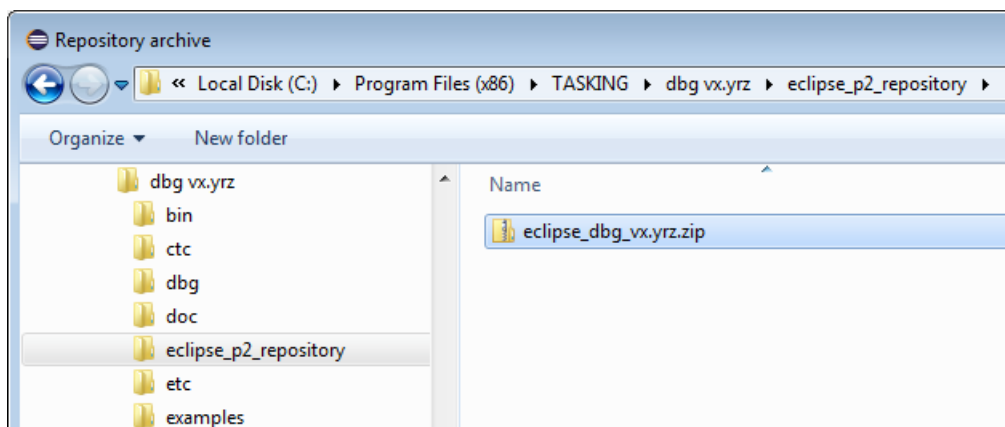


9. Browse to the directory where the TASKING Embedded Debugger was installed in Section 1.1.1, *Installation for Windows*, select the `eclipse_dbg_vx.yrz.zip` archive in the `eclipse_p2_repository` directory, where vx.yrz is your product version, and click **Open**.

10. Click **OK** in the Add Repository dialog.

    *The TASKING Tools software is available in the list of Available Software.*
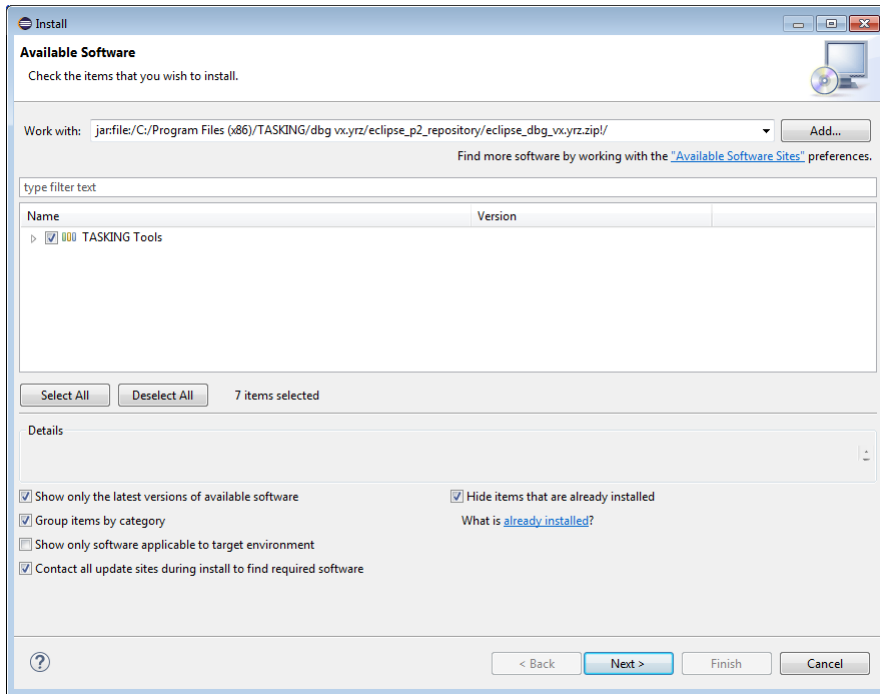
11. In the Available Software dialog, enable the **TASKING Tools** and click **Next**.

    *The Install Details dialog appears.*

12. Click **Next**.

    *The Review Licenses dialog appears.*

13. Read the `END-USER LICENSE AGREEMENT – TASKING`, select **I accept the terms of the license agreement** and click **Finish**.

14. Click **Yes** to restart Eclipse for the changes to take effect.

## 1.1.4. Bulk Installation into Existing Eclipse Environments

If you have to install the TASKING Embedded Debugger into existing Eclipse environments on a large number of computers you may want to distribute one and the same instance of the p2 repository including the license files (`licopt.txt`) to all computers involved without having to execute the install program on each computer. This functionality is called "bulk installation".

### To facilitate a bulk installation

1. Install the TASKING Embedded Debugger once in the regular way as described in Section 1.1.1, *Installation for Windows*. Since you want to install the software as plug-ins into an existing Eclipse environment, it is not necessary to select the Eclipse IDE feature. So, you can disable **Eclipse IDE** in step 5.

2.  Copy the `eclipse_p2_repository` directory and the licensing options file `licopt.txt` to a (temporary) directory that is accessible by all target computers.

3.  For each target computer, follow the procedure described in Section 1.1.3, *Installing the Software in an Existing Eclipse Environment* without the first step, and in step 9 instead of using the installation program to get the p2 repository, browse to the existing `eclipse_p2_repository` you copied at step 2.

4.  Copy the licence options file `licopt.txt` to all computers. It is also allowed to store this file on a network drive.

5.  On all computers where you copied `licopt.txt` to in step 4, set the environment variable `TSK_OPTIONS_FILE_`*Product-Code-version* to the location of `licopt.txt`. For example:

    `TSK_OPTIONS_FILE_SW000038v1_0r1 = `*install-dir*`\licopt.txt`

### Restrictions

Keep in mind that installing the TASKING Embedded Debugger this way has a few restrictions:

•  No Script Debugger will be installed

•  No menu items will be added to your Windows Start menu.

# 1.2. How to Use the Documentation

The documentation for the TASKING embedded debugger consists of:

•  online documentation for Eclipse

•  this user guide

It is strongly recommended to read the documentation in this order.

## Getting acquainted with Eclipse

If you are new to Eclipse, start familiarizing with Eclipse. Eclipse comes with several online documents. One document describes how Eclipse is organized as a Workbench, with Perspectives that contain Views; another document explains how to create a sample C/C++ project, build and debug it (CDT documentation).

To start with this documentation:

1.  Start Eclipse.

2.  From the **Help** menu, select **Help Contents**.

    *The help screen overlays the Eclipse Workbench.*

3.  In the left pane, select **Workbench User Guide** to learn more about working in Eclipse.

4. Continue with **C/C++ Development User Guide**, open the **Getting Started** entry and select **Debugging projects** to learn more about debugging.

   This Eclipse tutorial provides an overview of the debugging process. Be aware that the Eclipse example does not use the TASKING tools and TASKING debugger.

## TASKING Embedded Debugger User Guide (this manual)

The TASKING Embedded Debugger User Guide contains specific information for the TASKING embedded debugger. Its content overrides any information found in the Eclipse and CDT documentation.

The next chapters of this manual explain how to setup and use the TASKING embedded debugger.

# 1.3. Related Publications

## TriCore

- TriCore 1 32-bit Unified Processor Core, Volume 1 Core Architecture, V1.3 & V1.3.1 Architecture User's Manual, V1.3.8 [2007-11, Infineon]

- TriCore 1 32-bit Unified Processor Core, Volume 2 Instruction Set, V1.3 & V1.3.1 Architecture User's Manual, V1.3.8 [2007-11, Infineon]

- TC1xxx User's Manual, V2.0 [2007, Infineon]

- TriCore 1 32-bit Unified Processor Core, Embedded Applications Binary Interface (EABI), V1.3, V1.3.1 & V1.6 Architecture User's Manual, v2.5 [2008-01, Infineon]

## PCP

- PCP/DMA Architecture Specification, Rev. 1.13M [1998-09, Infineon]

- Peripheral Control Processor (PCP) User's Manual, V1.1.4 [1998-12, Infineon]

- PCP2 Target Specification, V1.0 [2000-06, Infineon]

## MCS

- AURIX TC27x 32-Bit Single-Chip Microcontroller Target Specification [V2.4, 2011-08, Infineon]

## 8051

- TASKING 8051 ELF/DWARF Application Binary Interface (EDABI) v1.1 [2009, Altium]

## ARM

- ARM Architecture Reference Manual - ARM DDI 0100I [2005, ARM Limited]

- ARMv7-M Architecture Reference Manual - ARM DDI 0403D [2010, ARM Limited]

- Cortex-M3 Technical Reference Manual [ARM Limited]

- Cortex Microcontroller Software Inferface Standard (CMSIS)

# Chapter 2. Setting up a Project

This tutorial shows how to create an embedded software project with the TASKING Embedded Debugger. The example assumes you have created an absolute ELF file with a TASKING VX-toolset for TriCore. The steps to create and debug projects for other TASKING toolsets such as, MCS, 8051 and ARM, are similar to that of the TriCore toolset.

By now you should be familiar with the Eclipse workbench, perspectives and views. If you are not, please read the Eclipse documentation as described in Section 1.2, *How to Use the Documentation*.

## 2.1. Create a Project

### Set the TASKING C/C++ perspective

Before creating a TASKING Embedded Debugger project, it is necessary to have the TASKING C/C++ perspective on the workbench. By default, this should be the case when you start Eclipse, but if it is not, do the following:

1.  Start Eclipse.

    *Eclipse starts with the last saved workbench layout.*

2.  To open the TASKING C/C++ perspective: from the **Window** menu, select **Perspective » Open Perspective » Other... » TASKING C/C++**.

    *The name of the perspective is displayed in the title bar of the workbench window.*

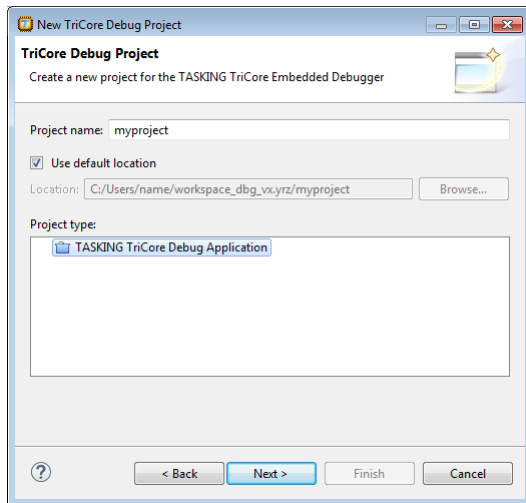### Create a TASKING Debug Project with the New Project wizard

1.  From the **File** menu, select **New » Project**

    *The New Project wizard appears.*
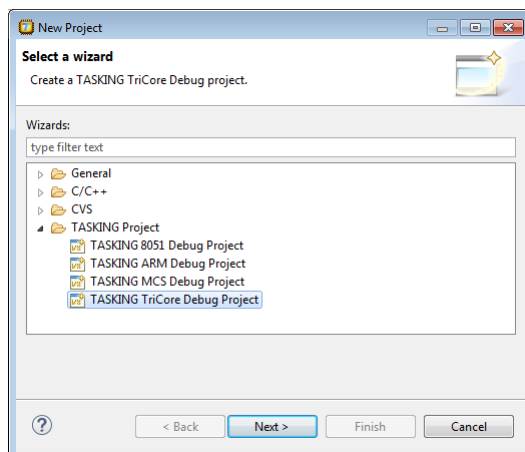
2.   Select **TASKING Project » TASKING TriCore Debug Project** and click **Next**.

   *The TriCore Debug Project page appears.*



3.   In the **Project name** field enter the name of the project (for example `myproject`).

   In the **Location** field you will see the location where the new project will be stored. To change the default location, you can uncheck the **Use default location** check box and browse for an alternative location. However, use the default location for now and click **Next**.

   *The Executable Files page appears.*
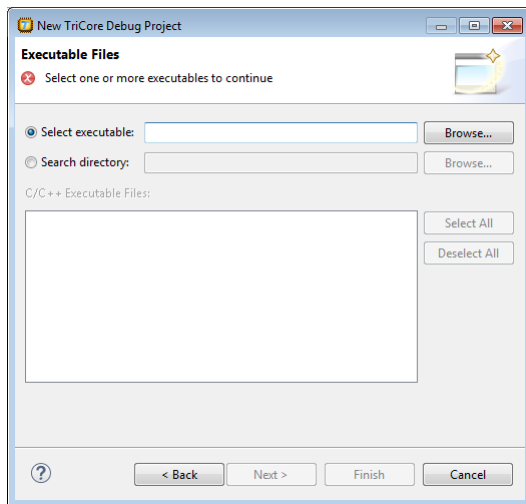
4. In the **Select executable** field, enter the full path to an absolute ELF file, or use the **Browse** button to choose an executable file. For example:

```
C:\Users\name\workspace_ctc_vx.yrz\myproject\Debug\myproject.elf
```

(Optional) If you choose **Select directory**, you can specify a directory containing the executable file(s). In that case select one or more executable files in the **C/C++ Executable Files** box.
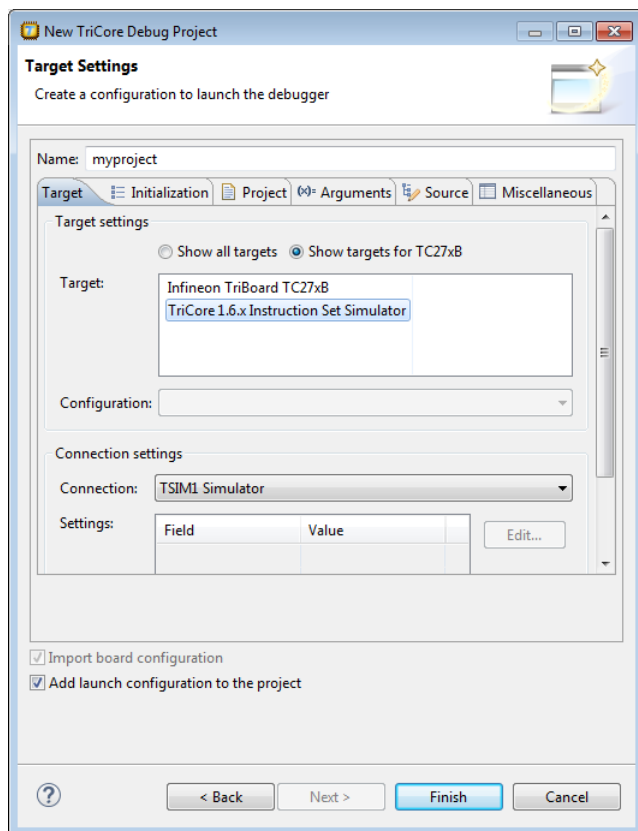
5. Click **Next**.

*The Processor page appears.*
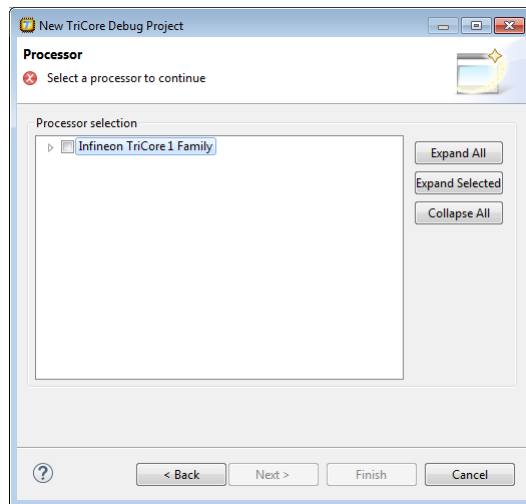


6. Select the same target processor that was used to build the application with, for example TC27xB (under AURIX Family). Afterwards you can always change the processor in the **Project » Properties for** dialog.

7. Click **Next**.

*The Target Settings page appears.*

8.  In order to debug your project you need to create a debug configuration.

    - Select a target. You can select a target board or a simulator. For this example we select the **TriCore 1.6.x Instruction Set Simulator**.

    - (Optional) If you selected a target board, specify the **Configuration** and **Connection settings**. For the simulator you can skip this.

    - (Optional) If you selected a target board, enable **Import board configuration**. Your project settings, such flash settings will be adjusted to the selected board configuration for you to build your application.

    - Enable **Add launch configuration to the project**. This allows you to debug your project.

    For details on all the tabs in this dialog, see Section 4.1, *Debug Configuration Dialog*.

9.  Open the **Source** tab and click **Add**.

    *The Add Source dialog appears.*

10. Select **File System Directory** and click **OK**.

*The Add File System Directory dialog appears.*

11. In the **Directory** field, enter the full path to the directory where the source files of your project are located, or use the **Browse** button to select the directory. For example:

   ```
   C:\Users\name\workspace_ctc_vx.yrz\myproject
   ```

12. Enable **Search subfolders**.

13. Click **OK**.

   *The directory is added to the Source Lookup Path. This way the debugger knows where to find the sources of your project.*

14. Click **Finish** to finish the wizard and to create the project.

   *The project has now been created and is the active project. The executable file is linked to the file you selected. The new project will let you debug but not build the executable.*
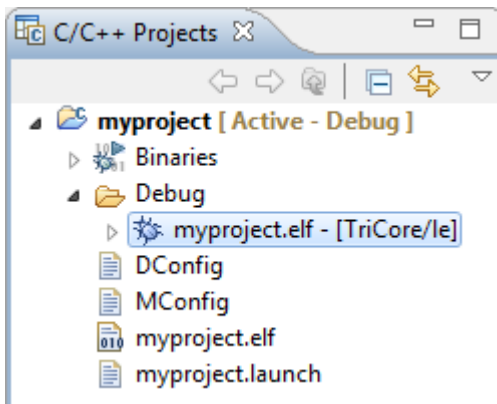


## 2.2. Configuring the Target

In order to debug your application, your project needs information about the target execution environment. The target can be a simulator or an evaluation board. If you are using the TASKING simulator you do not need to configure a target, you only need a launch configuration to start the debugger.

If you are using an evaluation board you have to create a launch configuration for your board in order to debug on a board. For all boards you also need to import a board configuration into your project. Based on your selections your project settings are adjusted, such as the processor, startup registers and flash settings.

> The steps below are only necessary if you have not configured the target when you created a project with the **New Project** wizard, or if you want to create another configuration.
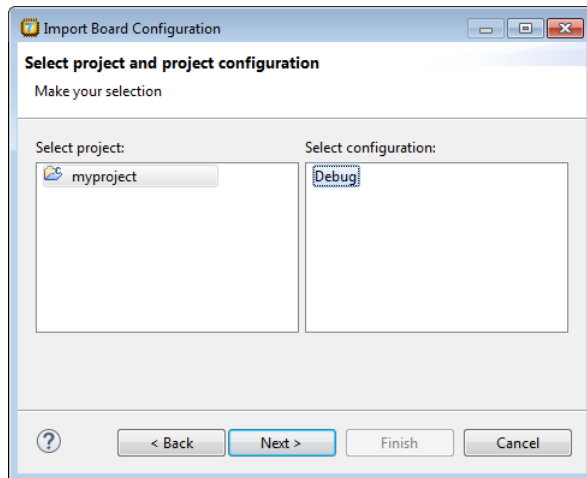
## Import TriCore board configuration

1.  From the **File** menu in Eclipse, select **Import**.

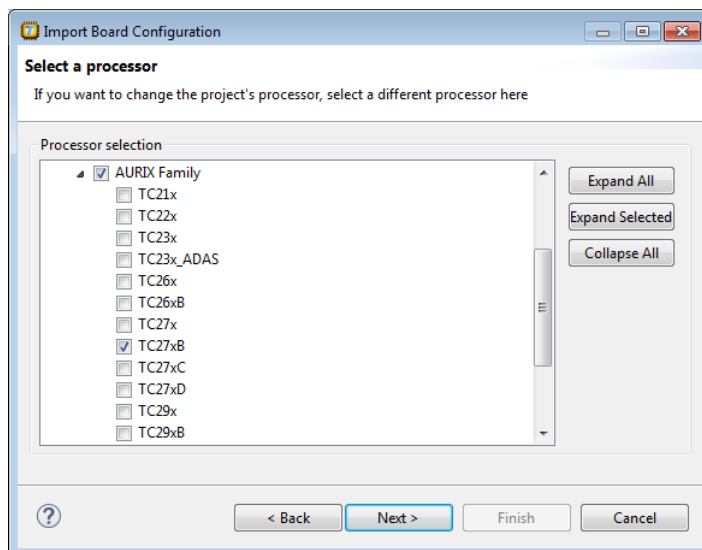    *The Import wizard appears.*

2.  Expand **TASKING C/C++**, select **Board Configuration** and click **Next**.

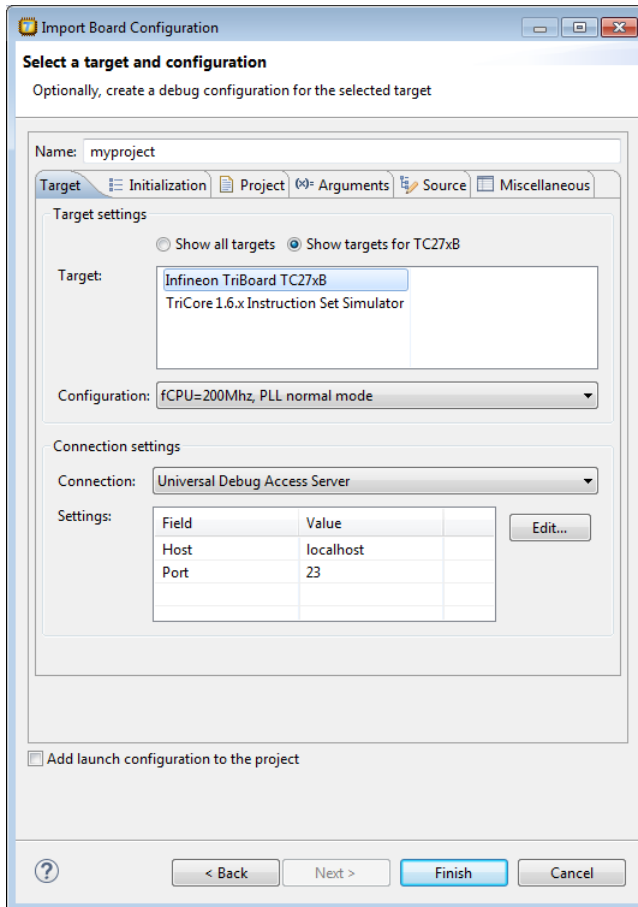    *The Import Board Configuration page appears.*



3.  Select your project (`myproject`) and project configuration (`Debug`) and click **Next**.

    *The Select a processor page appears.*

4.  If you want to change the project's processor, select a different processor.

5.  Click **Next**.

    *The Select a target and configuration page appears.*



6.  In the **Target** field, select the target evaluation board that you use to debug your application. By default only the boards are shown for the selected processor.

7.  In the **Configuration** field, select the configuration that matches the settings on your board.

8.  Enable **Add launch configuration to the project**. This allows you to debug your project.

    For details on all the tabs in this dialog, see Section 4.1, *Debug Configuration Dialog*.

9.  Open the **Source** tab and click **Add**.

    *The Add Source dialog appears.*

10. Select **File System Directory** and click **OK**.

    *The Add File System Directory dialog appears.*

11. In the **Directory** field, enter the full path to the directory where the source files of your project are located, or use the **Browse** button to select the directory. For example:

    ```
    C:\Users\name\workspace_ctc_vx.yrz\myproject
    ```

12. Enable **Search subfolders**.

13. Click **OK**.

    *The directory is added to the Source Lookup Path. This way the debugger knows where to find the sources of your project.*

14. Click the **Finish** button to import the configuration settings.

    *Your project settings, such as processor and flash settings are adjusted to the selected board configuration for you to debug your application. Note that only those registers are changed that are needed for the board to operate.*

The information in the Import Board Configuration wizard is based on Debug Target Configuration (DTC) files. DTC files define all possible configurations for a debug target. For more information on DTC files, see Chapter 5, *Debug Target Configuration Files*.

# 2.3. Project Properties

In the Properties dialog (**Project » Properties for *project***) you can change the target processor and change register values.

## To change the target processor (core)

1. From the **Project** menu, select **Properties for**.

    *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Processor**.

    *In the right pane the Processor page appears.*

3. From the **Processor selection** list, select a processor.

## To change the startup registers

The startup registers must be known to the debugger at the start of a debug session. When you create a new project and when you use the Import Board Configuration wizard, the registers are set as required for the specified board. In the Startup Registers page you can change the register values.

1. From the **Project** menu, select **Properties for**.

*The Properties dialog appears.*

2.  In the left pane, expand **C/C++ Build** and select **Startup Registers**.

    *In the right pane the Startup Registers page appears.*
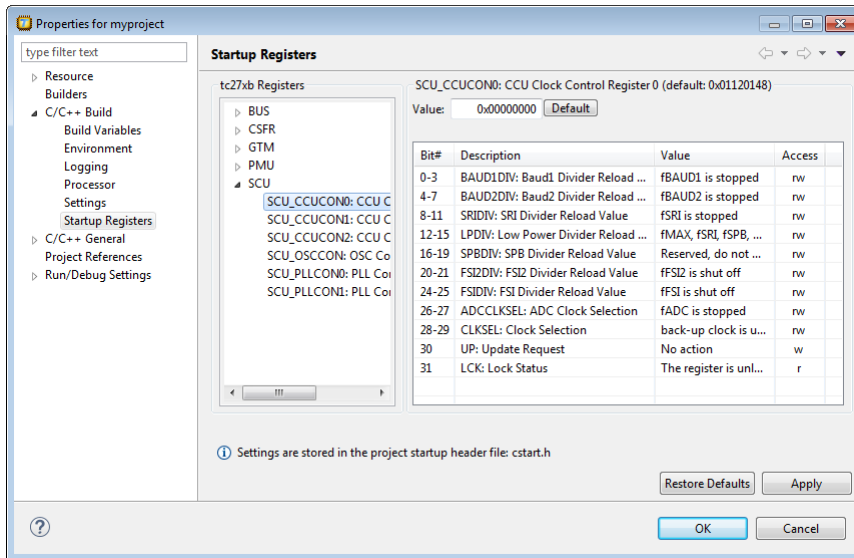


3.  Specify the registers and their settings that must be known to the debugger at the start of a debug session. If you click on the **Default** button, the register value is changed to the default as defined in the SFR files from the `include\sfr` directory of the product.

4.  Click **OK**.

    *The file cstart.h in your project is updated with the new values.*

> The values of the startup registers for a project are only set to their default values at project creation for the at that time selected processor.
>
> When you switch to a different processor afterwards, in the **Project » Properties for » C/C++ Build » Processor** property page, the registers are not set to their defaults again. The reason for that is that you may have set specific values in the startup registers that you want to keep.
>
> If you want to set all registers to their default values for the selected processor, you can do that any time by clicking on the **Restore Defaults** button on the **Project » Properties for » C/C++ Build » Startup Registers** property page.
>
> When you use the Import Board Configuration wizard to import (register) settings required for a certain board, only the registers needed to get the board going in the default situation are changed.

# 2.4. Using the Sample Projects

The TASKING Embedded Debugger comes with a number of examples. You can import the examples via the Welcome page. This is an alternative for importing existing projects via the **File » Import » TASKING C/C++ » TASKING** *target* **Embedded Debugger Example Projects** wizard.

## Import an existing project from the Welcome page

1.  From the **Help** menu, select **Welcome**.

    *The Welcome page appears.*

2.  Click the following button:
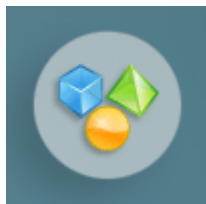
    

    *The Welcome Samples page appears.*

3.  Under **TASKING Embedded Debugger** click **TriCore examples** or **ARM examples**.

    *The Import examples dialog appears.*

4.  Select the example projects you want to import into the current workspace.

5.  Click **Finish**

    *The original examples are copied into the current workspace.*

    The project(s) should now be visible in the C/C++ Projects view.

# Chapter 3. Debugging your Application

This chapter guides you through a number of examples using the TASKING Embedded debugger.

The example used in this chapter assumes you have made a project named `myproject` with the following C source (`myproject.c`):

```c
#include <stdio.h>

int main( void )
{
    int i;
    for (i=1; i<=3; i++)
    {
        printf( "%d\n",i );
    }
    printf( "Hello world, " );
    printf( "this is \n" );
    printf( "a small %dst\n",i-3 );
    printf( "debugging example.\n" );
}
```

## 3.1. Create a Debug Configuration

Before you can debug a project, you need a Debug launch configuration. Such a configuration, identified by a name, contains all information about the debug project: which debugger is used, which project is used, which binary debug file is used, which perspective is used, ... and so forth.

You can create a launch configuration when you create a new project with the New Project wizard. In Section 2.1, *Create a Project* we created one for the TASKING simulator. At any time you can change this configuration. If you have not enabled option **Add launch configuration to the project** in the wizard, you have to create a custom debug configuration for your target board or the TASKING simulator. For a target board you can also create a launch configuration when you use the Import Board Configuration wizard. This was explained in Section 2.2, *Configuring the Target*.

To debug a project, you need at least one opened and active project in your workbench. In this chapter, it is assumed that the project `myproject` created in Section 2.1, *Create a Project* is opened and active in your workbench.

### Create or customize your debug configuration

To create or change a debug configuration follow the steps below.

1.  From the **Debug** menu, select **Debug Configurations...**

    *The Debug Configurations dialog appears.*

2.    Select **TASKING C/C++ Debugger** and click the **New launch configuration** button (
      ) to add a new configuration.
      Or: In the left pane, select the configuration you want to change, for example, **TASKING C/C++
      Debugger » myproject**.

3.    In the **Name** field enter the name of the configuration. By default, this is the name of the project, but
      you can give your configuration any name you want to distinguish it from the project name. For
      example enter `myproject.simulator` to identify the simulator debug configuration.

4.    On the **Target** tab, select the **TriCore 1.6.x Instruction Set Simulator** or any of the target boards.

5.    Open the **Source** tab and click **Add**.

      *The Add Source dialog appears.*

6.    Select **File System Directory** and click **OK**.

      *The Add File System Directory dialog appears.*

7.    In the **Directory** field, enter the full path to the directory where the source files of your project are
      located, or use the **Browse** button to select the directory. For example:

      `C:\Users\`*`name`*`\workspace_ctc_vx.`*`yrz`*`\myproject`

8.    Enable **Search subfolders**.

For details on all the tabs in the Debug Configurations dialog, see Section 4.1, *Debug Configuration
Dialog*.

## 3.2. Start a Debug Session

1.    From the **Debug** menu select **Debug** *project*.

      Alternatively you can click the 🔅 button in the main toolbar.

      *The TASKING Debug perspective is associated with the TASKING C/C++ Debugger. Because the
      TASKING C/C++ perspective is still active, Eclipse asks to open the TASKING Debug perspective.*

2.    Optionally, enable the option **Remember my decision** and click **Yes**.

      *The debug session is launched. This may take a few seconds.*
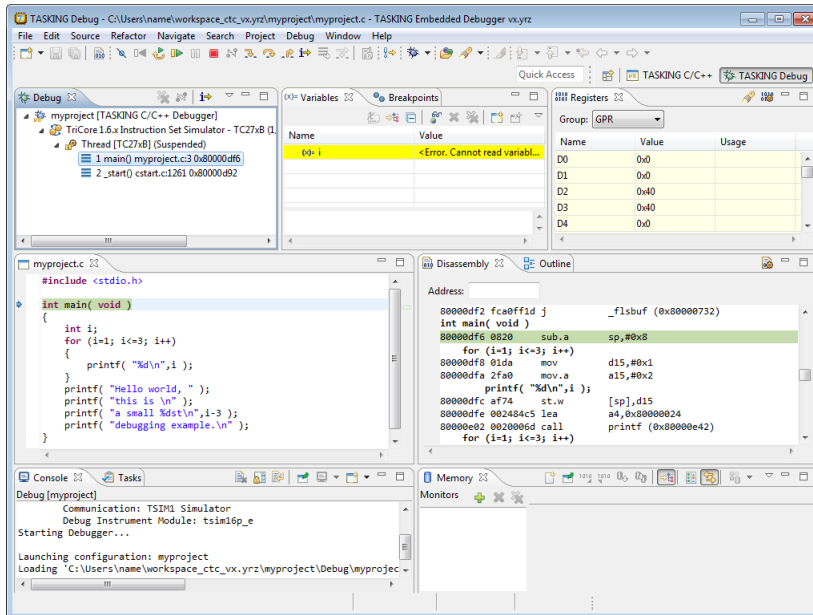
- The Debug view shows your running application. Because of the settings in the debug configuration, execution has suspended at the first instruction in the function `main()`.

- The Editor view shows the C source files of your application and shows the line where the execution has suspended.

- The Variables view shows the variables in your application.

For details on each view, see Section 4.3, *TASKING Debug Perspective*.

## Debugging a PCP project

When you use the simulator, the Debug view will show the TriCore core and the PCP as separate threads. When you select a thread this changes the context in the Disassembly view.
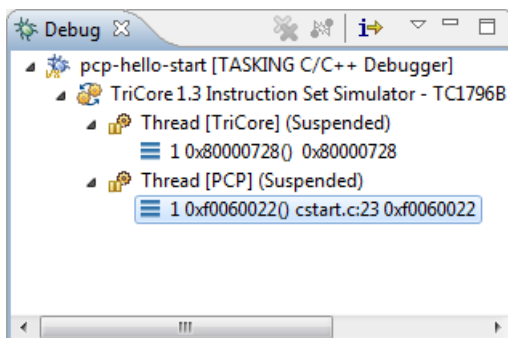
# 3.3. Stepping through the Application

At this moment your application is executing but suspended on the function `main()`. This means the C startup code has been executed already. From this point, you can step through your application while inspecting what happens.

1. From the **Debug** menu, select **Step Over**, or press **F6**, or click on the **Step Over** button (🔄) in the Debug view.

   *The highlight in the Edit view moves to the next statement.*

2. Press **F6** again.

   *The highlight in the Edit view moves to the next statement.*

   *In the Variables view, you can inspect the value of the variable `i`. It is now set to 1.*

3. Press **F6** again.

   *The `printf` statement has been executed now. The bottom area of your workbench now shows a new view: FSS # 1 - myproject.*

   FSS stands for *File System Simulation.* The FSS view simulates the input and output to and from the target board or simulator when you are debugging. The value of `int i` is printed and sent to the FSS view for output.

   > To clear the FSS view, right-click in the view and select **Clear**.
   >
   > To restart your application, from the **Debug** menu, select **Restart** (🔄).

4. Step further through your application.

   *Watch the value of `int i` in the Variables view and observe the output in the FSS view. The output is only flushed after a newline (`\n`)!*

## Interrupt aware stepping

When you debug your application in an interrupt enabled environment, it might be useful to enable **Interrupt aware stepping** (⟲➔). This prevents stepping into an interrupt handler when an interrupt occurs.

If an interrupt source continues generating interrupts while the target is stopped (either manually or by hitting a breakpoint), a following single step will always enter the Interrupt Service Routine (ISR). This can lead to some problems during single stepping.

With interrupt aware stepping enabled, interrupts are temporarily disabled after the target has stopped. When execution resumes the interrupts are restored.

# 3.4. Setting and Removing Breakpoints

Instead of stepping, you can set breakpoints to suspended the application at a certain point.

A breakpoint is set on an executable line of a program. If a breakpoint is enabled during debugging, the execution suspends *before* that line of code executes.

## Add breakpoints

To add a breakpoint:

• Double-click the marker bar located in the left margin of the C/C++ Editor next to the line of code where you want to add a breakpoint.

   *A dot* ● *is displayed in the marker bar and in the Breakpoints view, along with the name of the associated file. When the breakpoint is actually set, a check mark* ✔ *appears in front of the dot.*

## Disable breakpoints

You can disable a breakpoint or completely remove it. To disable a breakpoint, do one of the following:

• In the Breakpoints view, disable a breakpoint by clearing the check box.

• In the Editor view, right-click on a breakpoint dot in the margin and select **Disable Breakpoint**.

   *The blue breakpoint dot turns white.*

## Remove breakpoints

To completely remove the breakpoint, do one of the following:

• In the Breakpoints view, right-click on a breakpoint and select **Remove**.

• In the Editor view, right-click on a breakpoint dot in the margin and select **Toggle Breakpoint**.

• In the Editor view, double-click on a breakpoint.

   *The blue breakpoint dot disappears.*

## Example

With the techniques described above:

1.   Set a line breakpoint on the code line `printf( "a small %dst\n",i-3 );`.

2.   Clear the FSS view.

3.   Restart your application.

   *The application suspends when entering the main() function because this was defined in the Debug configuration.*

4.  To resume execution, from the **Debug** menu, select **Resume**, or press **F8**, or click on the **Resume** button ( ).

    *The application suspends execution, before this line is executed. The FSS view now shows:*

    ```
    1
    2
    3
    Hello world, this is
    ```

5.  Resume execution again to finish execution.

    *Note that though the application has finished execution, it has not been terminated yet. Your debug session is still active.*

## 3.5. Reload Current Application

When your application had changed, for example because you solved a bug, you can reload the application in the debugger without restarting it.

1.  Make the necessary changes in your source.

2.  Rebuild your application in your toolset product.

3.  Click on the **Reload current application** button ( ).

    *The new application is loaded in the debugger.*

## 3.6. End a Debug Session

To end the debug session:

1.  From the **Debug** menu select **Terminate** or click on the **Terminate** button ( ).

2.  To remove the debug session from the Debug view, right-click on the debug session and select **Remove All Terminated** or click on the **Remove All Terminated Launches** button ( ) in the Debug view.

## 3.7. Multiple Debug Sessions

It is possible to run multiple debug sessions. To do so, just repeat the steps for starting a debug session. First make sure that you have terminated all debug sessions.

1.  From the **Window** menu, select **Preferences**.

    *The Preferences dialog appears.*

2.  Select **TASKING » Debugger Start-up**.

3.  Enable the option **Allow multiple simultaneous debug sessions**.

4.  Select what you want to happen **When trying to start another debug session for the same configuration**. Select **Re-download** to download the absolute file again, or select **Start new session**, or select **Prompt** to get a question each time you try to start a new session.

5.  Click **OK**.

6.  From the **Debug** menu, select **Debug Configurations...**

    *The Debug Configurations dialog appears.*

7.  Select the debug configuration `myproject.simulator` and click on the **Debug** button.

    *The debug session launches.*

8.  Repeat steps 1 and 2, but in step 2 choose `myproject.board`.

There are now two debug sessions for the same application. In case you have multiple projects, you can make dedicated debug configurations for them. You can use these debug configurations to run multiple debug sessions at the same time.

Each session uses its own FSS view for output. In the Debug view you can select the debug session (or file in the debug session) for which you want to inspect, for example, the value of its variables in the Variables view.

# Chapter 4. Debugger Reference

This chapter describes the debugger and how you can run and debug a C or C++ application. This chapter only describes the TASKING specific parts.

## 4.1. Debug Configuration Dialog

This section describes the Debug Configurations dialog (**Debug » Debug Configurations**). The dialog shows several tabs.

### Target tab

On the **Target** tab you can select on which target the application should be debugged. An application can run on an external evaluation board, or on a simulator using your own PC. On this tab you can also select the connection settings. The information in this tab is based on the Debug Target Configuration (DTC) files as explained in Chapter 5, *Debug Target Configuration Files*.
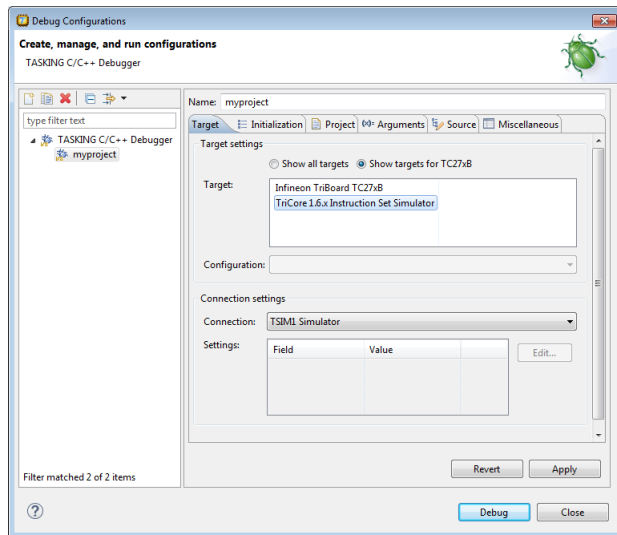


### Initialization tab

On the **Initialization** tab enable one or more of the following options:
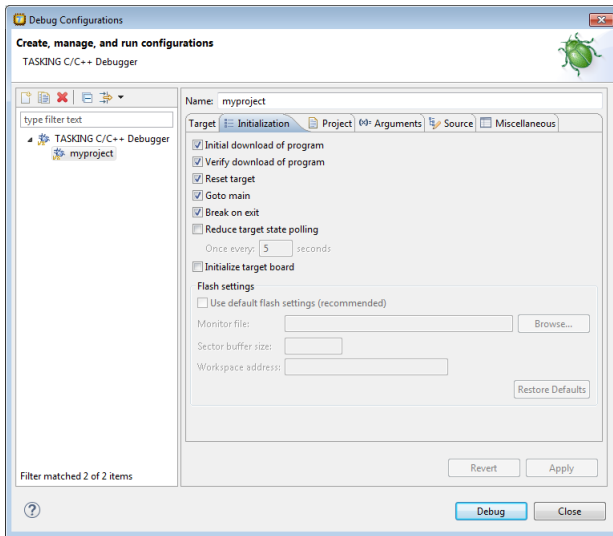
- **Initial download of program**

  If enabled, the target application is downloaded onto the target. If disabled, only the debug information in the file is loaded, which may be useful when the application has already been downloaded (or flashed) earlier. If downloading fails, the debugger will shut down.

- **Verify download of program**

  If enabled, the debugger verifies whether the code and data has been downloaded successfully. This takes some extra time but may be useful if the connection to the target is unreliable.

- **Reset target**

  If enabled, the target is immediately reset after downloading has completed.

- **Goto main**

  If enabled, only the C startup code is processed when the debugger is launched. The application stops executing when it reaches the first C instruction in the function `main()`. Usually you enable this option in combination with the option **Reset Target**.

- **Break on exit**

  If enabled, the target halts automatically when the `exit()` function is called.

- **Reduce target state polling**

  If you have set a breakpoint, the debugger checks the status of the target every *number* of seconds to find out if the breakpoint is hit. In this field you can change the polling frequency.

- **Initialize Target Board**

Some target boards contain a power supply chip which needs to be initialized every time after power-on, before hardware debugging is started. If enabled, the target board is initialized automatically before the start of a debug session. If disabled, you need to initialize the target board manually via the Devices view. See Section 4.3.13, *Devices View*.

## Initialization tab: Flash settings

- **Use default flash settings (recommended)**

  By default, the flash settings are derived from the `.dtc` file for the chosen target processor. So, when you change processors the flash settings change automatically. If you do not want that, you can specify your own flash settings. You can click **Restore Defaults** to restore the default flash settings.

- **Monitor file**

  Filename of the monitor, usually an Intel Hex or S-Record file.
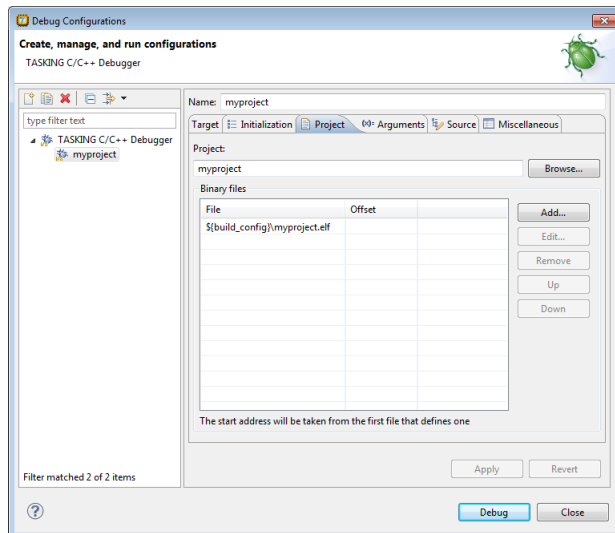
- **Sector buffer size**

  Specifies the buffer size for buffering a flash sector.

- **Workspace address**

  The address of the workspace of the flash programming monitor.

## Project tab

On the **Project** tab, you can set the properties for the debug configuration such as a name for the project and the application binary file(s) which are used when you choose this configuration.

- In the **Project** field, you can choose the project for which you want to make a debug configuration. Because the project `myproject` is the active project, this project is filled in automatically. Click the Browse... button to select a different project. Only the *opened* projects in your workbench are listed.

- In the **Binary files** group box, you can choose one or more binary files to debug. The file `myproject.elf` is automatically selected from the active project.

  The order of the binary files matters. Use the **Up** and **Down** buttons to change the order. If there are multiple files, the application start address is taken from the first file that defines one. An ELF file always defines one, whereas Hex files may not.

  Note that conflicts between symbols could arise, for example when you download two ELF files that both contain the function `main()`. When you download multiple files, we recommend that the first binary file is an ELF file that contains the startup code and `main()` and that the other files are auxiliary Hex files.

  **To add a binary file**

  1. Click **Add...** to add a binary file.

     *The Add Binary File dialog appears.*
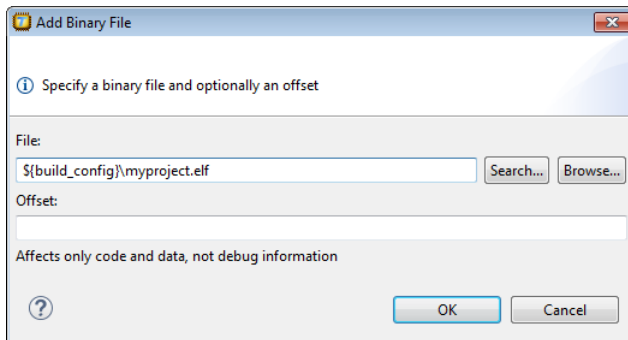


  2. Specify the binary file, use the **Search...** button to select one from the active project, or use the **Browse...** button to search the file system.

  3. Optionally, specify an address offset. The value will be added to all target addresses in the binary file.

     Note that the address offset will be applied only to code, data and the start address, not to debug information. Specifying a non-zero offset is not recommended for an ELF/DWARF file. If the offset causes an address to underflow or overflow an error occurs.

## Arguments tab

If your application's `main()` function takes arguments, you can pass them in this tab.
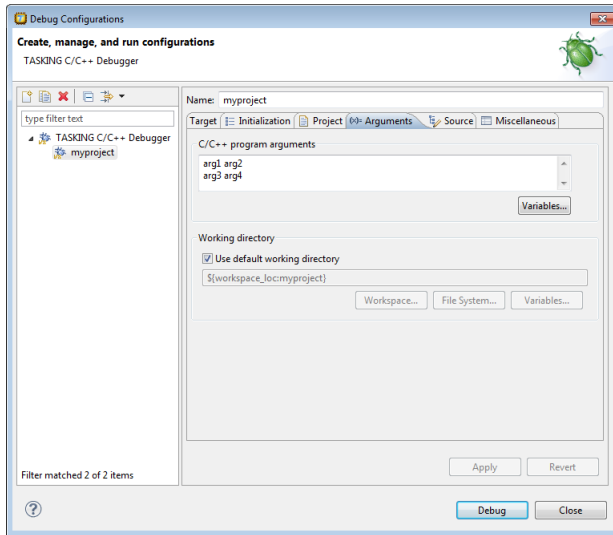
Arguments are conventionally passed in the argv[] array. Because this array is allocated in target memory, make sure you have allocated sufficient memory space for it.

When you made your project with a TASKING VX-toolset product, make sure you have set the following options before you build your project:

• In your TASKING VX-toolset product, in the C/C++ perspective select **Project » Properties for** to open the Properties dialog. Expand **C/C++ Build » Startup Configuration**. Enable the option **Enable passing argc/argv to main()** and specify a **Buffer size for argv**.
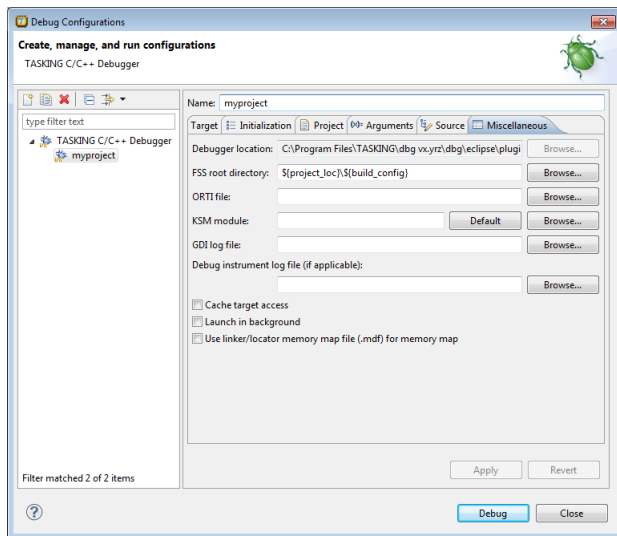
## Source tab

On the **Source** tab, you can add the directory where the source files of your project are located.
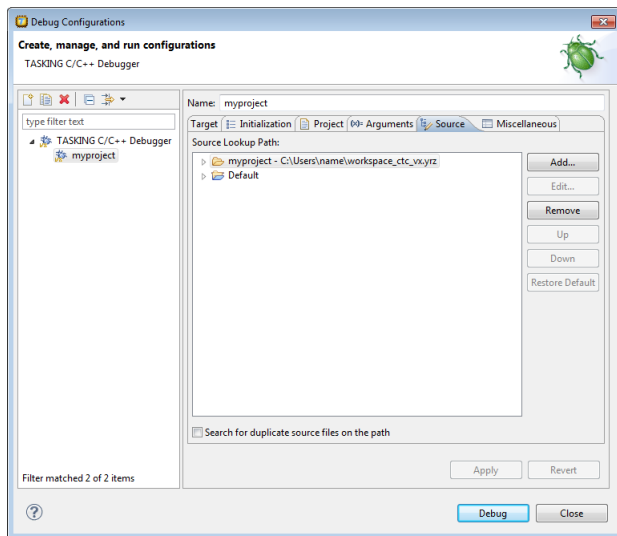
## Miscellaneous tab

On the **Miscellaneous** tab you can specify several file locations.



* **Debugger location**

  The location of the debugger itself. This should not be changed.

* **FSS root directory**

  The initial directory used by file system simulation (FSS) calls. See the description of the FSS view.

- **ORTI file** and **KSM module**

  If you wish to use the debugger's special facilities for kernel-aware debugging, specify the name of a Kernel Debug Interface (KDI) compatible KSM module (shared library) in the appropriate edit box. The TASKING Embedded Debugger comes with a KSM suitable for RTOS kernels. If you wish to use this, browse for the file `orti_radm.dll` (Windows) or `orti_radm.so` (UNIX) in the `ctc\bin` directory of the product. See also the description of the RTOS view.

- **GDI log file** and **Debug instrument log file**

  You can use the options GDI log file and Debug instrument log file (if applicable) to control the generation of internal log files. These are primarily intended for use by or at the request of Altium support personnel.

- **Cache target access**

  Except when using a simulator, the debugger's performance is generally strongly dependent on the throughput and latency of the connection to the target. Depending on the situation, enabling this option may result in a noticeable improvement, as the debugger will then avoid re-reading registers and memory while the target remains halted. However, be aware that this may cause the debugger to show the wrong data if tasks with a higher priority or external sources can influence the halted target's state.

- **Launch in background**

  When this option is disabled you will see a progress bar when the debugger starts. If you do not want to see the progress bar and want that the debugger launches in the background you can enable this option.

- **Use linker/locator memory map file (.mdf) for memory map**

  You can use this option to find errors in your application that cause access to non-existent memory or cause an attempt to write to read-only memory. When building your project, the linker/locator creates a memory description file (`.mdf`) file which describes the memory regions of the target you selected in your project properties. The debugger uses this file to initialize the debugging target.

  This option is only useful in combination with a simulator as debug target. The debugger may fail to start if you use this option in combination with other debugging targets than a simulator.

## 4.2. Pipeline and Cache During Debugging

The pipeline and the cache(s) of the TriCore architecture are implemented in such a way that there is no automatic coherency between the state as seen by the CPU itself and that seen by the debugger via OCDS. For example, if the target halts on a breakpoint, a memory value read via OCDS may not represent the "real" value as implied by the program logic if the value still has to be written back from the cache.

The TASKING debugger has a special "sync(hronize)-on-halt" facility to bring about this coherency. Every time the target halts, the debugger will execute a routine that flushes the pipeline and the caches insofar as necessary. This routine will be added to a new TriCore project when you use a TriCore VX-toolset product, unless you disable the option **Include debugger synchronization utility** in the New C/C++ Project wizard, which you may want to do if you do not intend to use the TASKING debugger. For example, for third-party debuggers this synchronization utility might not be necessary. In any case, by default the

code will be linked in only in the Debug configuration, not in the Release configuration (via the Exclude from build facility).

For more information see the *TASKING VX-toolset for TriCore User Guide*.

## 4.3. TASKING Debug Perspective

After you have launched the debugger, you are either asked if the TASKING Debug perspective should be opened or it is opened automatically. The Debug perspective consists of several views.

To open views in the Debug perspective:

1. Make sure the Debug perspective is opened

2. From the **Window** menu, select **Show View »**

3. Select a view from the menu or choose **Other...** for more views.

By default, the Debug perspective is opened with the following views:
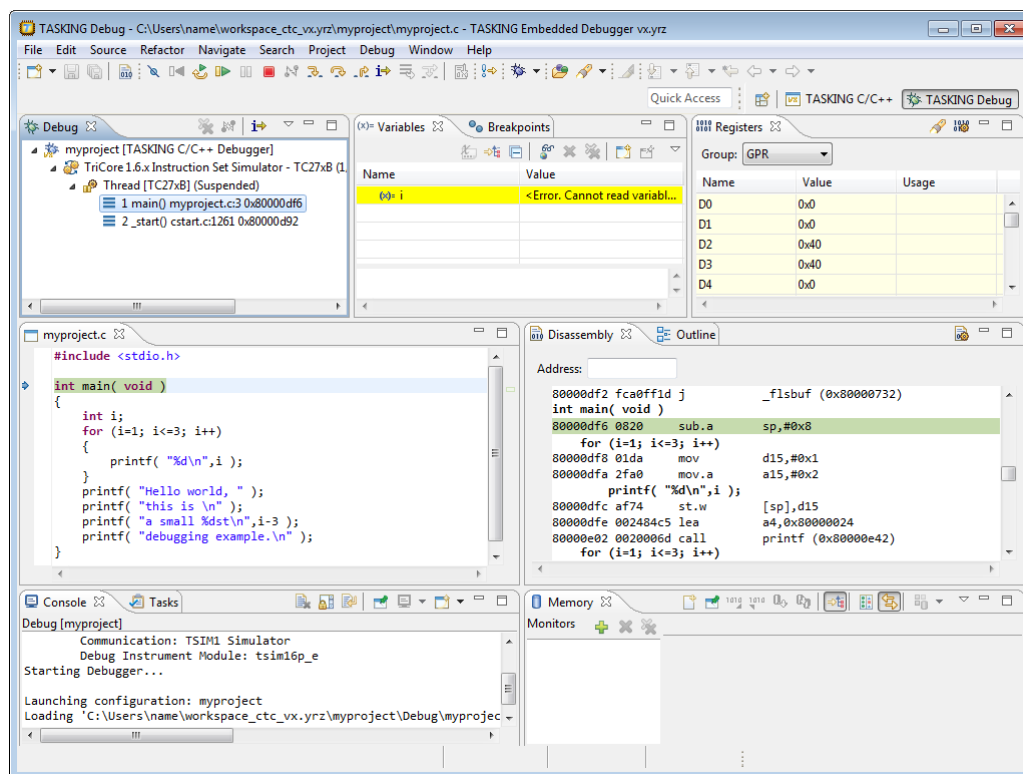
## 4.3.1. Debug View

The Debug view shows the target information in a tree hierarchy shown below with a sample of the possible icons:

| Icon | Session item | Description |
|------|--------------|-------------|
| ![icon] | Launch instance | Launch configuration name and launch type |
| ![icon] | Debugger instance | Debugger name and state |
| ![icon] | Thread instance | Thread number and state |
| ![icon] | Stack frame instance | Stack frame number, function, file name, and file line number |

### Stack display

During debugging (running) the actual stack is displayed as it increases or decreases during program execution. By default, all views present information that is related to the current stack item (variables, memory, source code etc.). To obtain the information from other stack items, click on the item you want.

The Debug view displays stack frames as child elements. It displays the reason for the suspension beside the thread, (such as end of stepping range, breakpoint hit, and signal received). When a program exits, the exit code is displayed.

The Debug view contains numerous functions for controlling the individual stepping of your programs and controlling the debug session. You can perform actions such as terminating the session and stopping the program. All functions are available from the right-click menu, though commonly used functions are also available from the toolbar.

### Controlling debug sessions

| Icon | Action | Description |
|------|--------|-------------|
| ![icon] | Remove all | Removes all terminated launches. |
| ![icon] | Reset target system | Resets the target system and restarts the application. |
| ![icon] | Restart | Restarts the application. The target system is *not* reset. |
| ![icon] | Resume | Resumes the application after it was suspended (manually, breakpoint, signal). |
| ![icon] | Suspend | Suspends the application (pause). Use the Resume button to continue. |
| ![icon] | Relaunch | Right-click menu. Restarts the selected debug session when it was terminated. If the debug session is still running, a new debug session is launched. |
| ![icon] | Reload current application | Reloads the current application without restarting the debug session. The application does restart of course. |

| Icon | Action | Description |
|---|---|---|
| ■ | Terminate | Ends the selected debug session and/or process. Use Relaunch to restart this debug session, or start another debug session. |
| ▣ | Terminate all | Right-click menu. As terminate. Ends *all* debug sessions. |
| ▣ | Terminate and remove | Right-click menu. Ends the debug session and removes it from the Debug view. |
| ▣ | Terminate and Relaunch | Right-click menu. Ends the debug session and relaunches it. This is the same as choosing Terminate and then Relaunch. |
| ⋈ | Disconnect | Detaches the debugger from the selected process (useful for debugging attached processes). |

### Stepping through the application

| Icon | Action | Description |
|---|---|---|
| ⤵ | Step into | Steps to the next source line or instruction. |
| ⤻ | Step over | Steps over a called function. The function is executed and the application suspends at the next instruction after the call. |
| ⤴ | Step return | Executes the current function. The application suspends at the next instruction after the return of the function. |
| i⤳ | Instruction stepping | Toggle. If enabled, the stepping functions are performed on instruction level instead of on C source line level. |
| ⟐⤳ | Interrupt aware stepping | Toggle. If an interrupt source continues generating interrupts while the target is stopped (either manually or by hitting a breakpoint), a following single step will always enter the Interrupt Service Routine (ISR). This can lead to some problems during single stepping. With interrupt aware stepping enabled, interrupts are temporarily disabled after the target has stopped. When execution resumes the interrupts are restored. |

### Miscellaneous

| Icon | Action | Description |
|---|---|---|
| 📄 | Copy Stack | Right-click menu. Copies the stack as text to the windows clipboard. You can paste the copied selection as text in, for example, a text editor. |
| 🔧 | Edit *project...* | Right-click menu. Opens the debug configuration dialog to let you edit the current debug configuration. |
| 📝 | Edit Source Lookup... | Right-click menu. Opens the Edit Source Lookup Path window to let you edit the search path for locating source files. |

## 4.3.2. Breakpoints View

You can add, disable and remove breakpoints by clicking in the marker bar (left margin) of the Editor view. This is explained in Section 3.4, *Setting and Removing Breakpoints*.

### Description

The Breakpoints view shows a list of breakpoints that are currently set. The button bar in the Breakpoints view gives access to several common functions. The right-most button ▽ opens the Breakpoints menu.

### Types of breakpoints

To access the breakpoints dialog, add a breakpoint as follows:

1. Click the **Add TASKING Breakpoint** button (🔵).

    *The Breakpoints dialog appears.*

Each tab lets you set a breakpoint of a special type. You can set the following types of breakpoints:

• **File breakpoint**

   The target halts when it reaches the specified line of the specified source file. Note that it is possible that a source line corresponds to multiple addresses, for example when a header file has been included into two different source files or when inlining has occurred. If so, the breakpoint will be associated with all those addresses.

• **Function**

   The target halts when it reaches the first line of the specified function. If no source file has been specified and there are multiple functions with the given name, the target halts on all of those. Note that function breakpoints generally will not work on inlined instances of a function.

• **Address**

   The target halts when it reaches the specified instruction address.

• **Stack**

   The target halts when it reaches the specified stack level.

• **Data**

   The target halts when the given variable or memory location (specified in terms of an absolute address) is read or written to, as specified.

• **Instruction**

   The target halts when the given number of instructions has been executed.

• **Cycle**

   The target halts when the given number of clock cycles has elapsed.

• **Timer**

   The target halts when the given amount of time elapsed.

In addition to the type of the breakpoint, you can specify the condition that must be met to halt the program.

In the **Condition** field, type a condition. The condition is an expression which evaluates to 'true' (non-zero) or 'false' (zero). The program only halts on the breakpoint if the condition evaluates to 'true'.

In the **Ignore count** field, you can specify the number of times the breakpoint is ignored before the program halts. For example, if you want the program to halt only in the fifth iteration of a while-loop, type '4': the first four iterations are ignored.

## 4.3.3. File System Simulation (FSS) View

### Description

The File System Simulation (FSS) view is automatically opened when the target requests FSS input or generates FSS output. The virtual terminal that the FSS view represents, follows the VT100 standard. If you right-click in the view area of the FSS view, a menu is presented which gives access to some self-explanatory functions.

### VT100 characteristics

The `queens` example demonstrates some of the VT100 features. (You can import it into your workspace via **File » Import » TASKING C/C++ » TASKING Embedded Debugger Example Projects**.) Per debugging session, you can have more than one FSS view, each of which is associated with a positive integer. By default, the view "FSS #1" is associated with the standard streams `stdin`, `stdout`, `stderr` and `stdaux`. Other views can be accessed by opening a file named "terminal window <number>", as shown in the example below.

```
FILE * f3 = fopen("terminal window 3", "rw");
fprintf(f3, "Hello, window 3.\n");
fclose(f3);
```

You can set the initial working directory of the target application in the Debug configuration dialog (see also Section 4.1, *Debug Configuration Dialog*):

1. On the **Debugger** tab, select the **Miscellaneous** sub-tab.

2. In the **FSS root directory** field, specify the FSS root directory.

The FSS implementation is designed to work without user intervention. Nevertheless, there are some aspects that you need to be aware of.

First, the interaction between the C library code and the debugger takes place via a breakpoint, which incidentally is not shown in the Breakpoints view. Depending on the situation this may be a hardware breakpoint, which may be in short supply.

Secondly, proper operation requires certain code in the C library to have debug information. This debug information should normally be present but might get lost when this information is stripped later in the development process.

When you use MIL linking/splitting the C library is translated along with your application. Therefore you need to build your application with debug information generation enabled when FSS support is needed.

### 4.3.4. Disassembly View

The Disassembly view shows target memory disassembled into instructions and / or data. If possible, the associated C / C++ source code is shown as well. If you are debugging a multi-core project, each thread has its own Disassembly view.

To open a thread specific Disassembly view, select a stack frame in a thread in the Debug view and click the **Open thread specific Disassembly View** button (  ).

The **Address** field shows the address of the current selected line of code.

To view the contents of a specific memory location, type the address in the **Address** field. If the address is invalid, the field turns red.

### 4.3.5. Expressions View

The Expressions view allows you to evaluate and watch regular C expressions.

To add an expression:

Click **OK** to add the expression.

1. Right-click in the Expressions View and select **Add Watch Expression**.

   *The Add Watch Expression dialog appears.*

2. Enter an expression you want to watch during debugging, for example, the variable name "i"

If you have added one or more expressions to watch, the right-click menu provides options to **Remove** and **Edit** or **Enable** and **Disable** added expressions.

- You can access target registers directly using `#NAME`. For example "`arr[#R0 << 3]`" or "`#TIMER3 = m++`". If a register is memory-mapped, you can also take its address, for example, "`&#ADCIN`".

- Expressions may contain target function calls like for example "`g1 + invert(&g2)`". Be aware that this will not work if the compiler has optimized the code in such a way that the original function code does not actually exist anymore. This may be the case, for example, as a result of inlining. Also, be aware that the function and its callees use the same stack(s) as your application, which may cause problems if there is too little stack space. Finally, any breakpoints present affect the invoked code in the normal way.

### 4.3.6. Memory View

Use the Memory view to inspect and change process memory. The Memory view supports the same addressing as the C and C++ languages. You can address memory using expressions such as:

- `0x0847d3c`

- `(&y)+1024`

- `*ptr`

## Monitors

To monitor process memory, you need to add a *monitor*:

1.   In the Debug view, select a debug session. Selecting a thread or stack frame automatically selects the associated session.

2.   Click the **Add Memory Monitor** button in the Memory Monitors pane.

     *The Monitor Memory dialog appears.*

3.   Type the address or expression that specifies the memory section you want to monitor and click **OK**.

     *The monitor appears in the monitor list and the Memory Renderings pane displays the contents of memory locations beginning at the specified address.*

To remove a monitor:

1.   In the Monitors pane, right-click on a monitor.

2.   From the popup menu, select **Remove Memory Monitor**.

## Renderings

You can inspect the memory in so-called *renderings*. A rendering specifies how the output is displayed: hexadecimal, ASCII, signed integer, unsigned integer or traditional. You can add or remove renderings per monitor. Though you cannot change a rendering, you can add or remove them:

1.   Click the **New Renderings...** tab in the Memory Renderings pane.

     *The Add Memory Rendering dialog appears.*

2.   Select the rendering you want (**Traditional**, **Hex**, **ASCII**, **Signed Integer**, **Unsigned Integer** or **Hex Integer**) and click **Add Rendering(s)**.

To remove a rendering:

1.   Right-click on a memory address in the rendering.

2.   From the popup menu, select **Remove Rendering**.

## Changing memory contents

In a rendering you can change the memory contents. Simply type a new value.

> **Warning**: Changing process memory can cause a program to crash.

The right-click popup menu gives some more options for changing the memory contents or to change the layout of the memory representation.

### 4.3.7. Compare Application View

You can use the Compare Application view to check if the downloaded application matches the application in memory. Differences may occur, for example, if you changed memory addresses in the Memory view manually, or your application overwrote parts of the memory.

• To check for differences, click the **Compare** button.

### 4.3.8. Heap View

With the Heap view you can inspect the status of the heap memory. This can be illustrated with the following example:

```
string = (char *) malloc(100);
strcpy ( string, "abcdefgh" );
free (string);
```

If you step through these lines during debugging, the Heap view shows the situation after each line has been executed. Before any of these lines has been executed, there is no memory allocated and the Heap view is empty.

• After the first line the Heap view shows that memory is occupied, the description tells where the block starts, how large it is (`100 MAUs`) and what its content is (`0x0, 0x0, ...`).

• After the second line, `"abcdefgh"` has been copied to the allocated block of memory. The description field of the Heap view again shows the actual contents of the memory block (`0x61, 0x62,...`).

• The third line frees the memory. The Heap view is empty again because after this line no memory is allocated anymore.

### 4.3.9. Logging View

Use the Logging view to control the generation of internal log files. This view is intended mainly for use by or at the request of Altium support personnel.

### 4.3.10. RTOS View

The debugger has special support for debugging real-time operating systems (RTOSs). This support is implemented in an RTOS-specific shared library called a *kernel support module* (KSM) or *RTOS-aware debugging module* (RADM). Specifically, the TASKING Embedded Debugger ships with a KSM supporting the ISO 17356 standard. You have to create your own Run Time Interface (ORTI) and specify this file on the **Miscellaneous** tab while configuring a customized debug configuration (see also Section 4.1, *Debug Configuration Dialog*):

1. From the **Debug** menu, select **Debug Configurations...**

   *The Debug Configurations dialog appears.*

2. In the left pane, select the configuration you want to change, for example, **TASKING C/C++ Debugger » myproject**.

   Or: click the **New launch configuration** button (🗋) to add a new configuration.

3.  Open the **Miscellaneous** tab.

4.  In the **ORTI file** field, specify the name of your own ORTI file.

5.  If you want to use the supplied KSM suitable for RTOS kernels, in the **KSM module** field browse for the file `orti_radm.dll` (Windows) or `orti_radm.so` (UNIX) in the `ctc\bin` directory of the product.
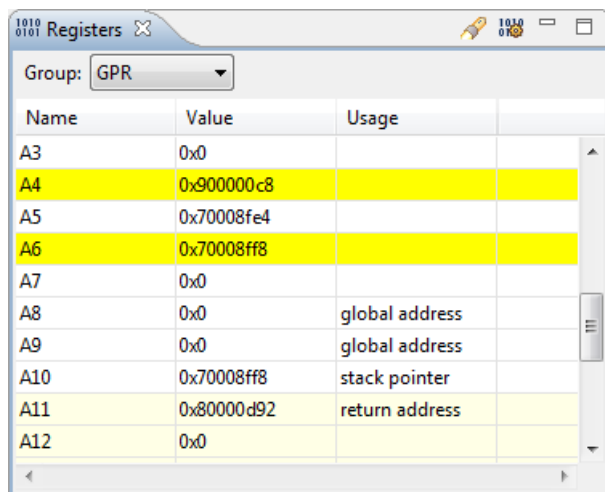
The debugger supports ORTI specifications v2.0 and v2.1.

## 4.3.11. Registers View

In the Registers view you can examine the value of registers while stepping through your application. If you are debugging a multi-core project, each thread has its own Registers view. The registers are organized in a number of *register groups*, which together contain all known registers. You can select a group to see which registers it contains. This view has a number of features:

• While you step through the application, the registers involved in the step turn yellow. If you scroll in the view or switch groups, some registers may appear on a lighter yellow background, indicating that the debugger does not know whether the registers have changed because the debugger did not read the registers before the step began.
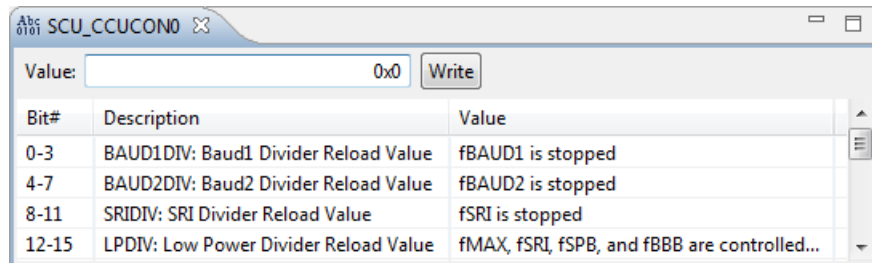
Registers view:

Thread specific Registers view:
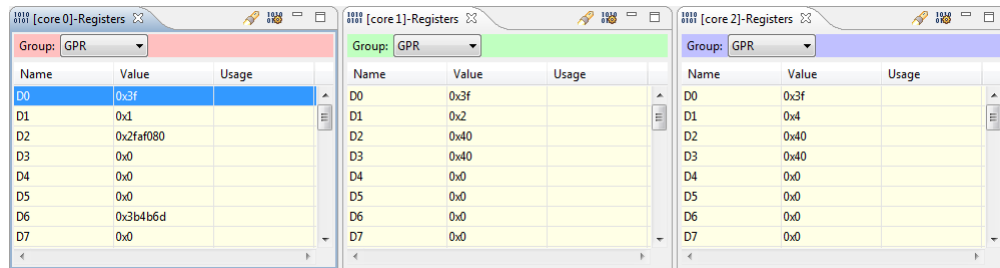


- To open a thread specific Registers view, select a thread in the Debug view and click the **Open thread specific Registers View** button (⚙️).

- You can change each register's value.

- For some registers the menu entry **Symbolic Representation** is available in their right-click popup menu. This opens a new view which shows the internal fields of the register. (Alternatively, you can double-click on a register). For example, the SCU_CCUCON0 register from the SCU group may be shown as follows:



In this view you can set the individual values in the register, either by selecting a value from a drop-down box or by simply entering a value depending on the chosen field. To update the register with the new values, click the **Write** button.

- You can search for a specific register: right-click on a register and from the popup menu select **Find Register...**. Enter a group or register name filter, click the register you want to see and click **OK**. The register of your interest will be shown in the view.

## 4.3.12. Trace View

If tracing is enabled, the Trace view shows the code was most recently executed. For example, while you step through the application, the Trace view shows the executed code of each step. To enable tracing:

- Right-click in the Trace view and select **Trace**.

  *A check mark appears when tracing is enabled.*

The view has three tabs, **Source**, **Instruction** and **Raw**, each of which represents the trace in a different way. However, not all target environments will support all three of these. The view is updated automatically each time the target halts.

## 4.3.13. Devices View

The Devices view shows which target boards are connected to your PC. The list of attached target boards has no specific relation with your Embedded Debugger project.
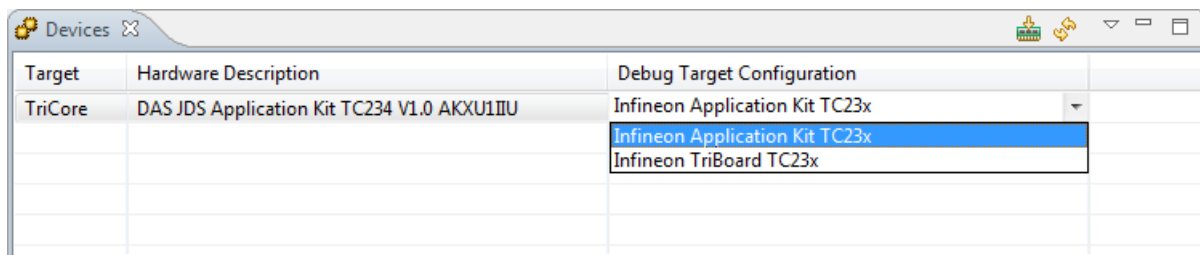


| Icon | Action | Description |
|------|--------|-------------|
| ⛏ | Initialize Target Board | Right-click menu and toolbar. Initializes the selected target board. |
| 🔧 | Refresh | Right-click menu and toolbar. When you have connected or disconnected a board and in the Preferences dialog polling is disabled, you can manually refresh the display of this view. |
| ▽ | Menu | Drop-down menu. Open the Preferences dialog. |

Some target boards contain a power supply chip which needs to be initialized every time after power-on, before hardware debugging is started. This initialization is necessary before starting a debug session, otherwise communication is not possible. You can do this initialization manually or automatically. Any of the listed target boards can be initialized, you need to take care that the target board is initialized when needed. Initialization is never harmful.

### Initialize target board manually

1.  Connect the target board. If the board does not appear in the Devices view automatically, click the **Refresh** button (🔧).

    *The board is visible in the Devices view.*

2.  Click in the **Debug Target Configuration** field and from the drop-down menu select the configuration that matches your target board.

3.  Right-click on the target board and select **Initialize Target Board**, or click the **Initialize Target Board** button (⛏).

### Initialize target board automatically

1.  Open the Debug Configurations dialog (**Debug » Debug Configurations**).

2.   On the **Initialization** tab, enable **Initialize target board**.

*The target board will be initialized automatically before the debug session starts.*

### Polling of target boards

At startup of the debugger the Devices view shows the list of attached target boards. By default this list is not refreshed. You can set a polling interval for automatic refreshing.

1.   From the drop-down menu or right-click menu in the Devices view, select **Preferences**.

*The Preferences (Filtered) dialog appears. This is a filtered version from the standard Preferences dialog (Window » Preferences » TASKING » Devices).*

2.   Enable **Poll connected target boards** and fill in a **Polling interval** in seconds. The default is 10 seconds.

If polling is disabled, you have to refresh the Devices view manually after you connected or disconnected a board.

# 4.4. Multi-core Hardware Debugging

The TASKING debugger supports multi-core hardware debugging. When you start the debugger for a multi-core device (AURIX), you can debug all TriCore cores of the selected device.

The following picture shows the example `tc_cpp-multi-core-test`, which is a C++ multi-core application and is debugged on an Infineon TriBoard TC29x.
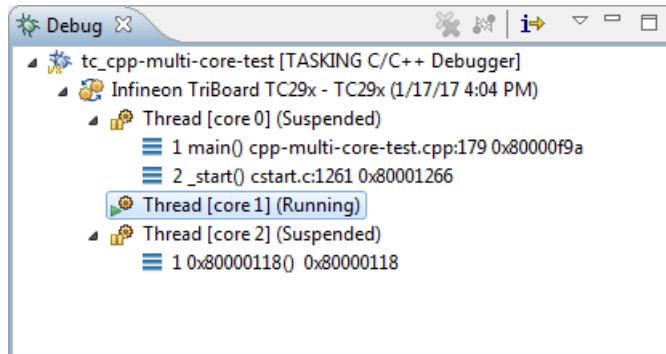


For each of the three TriCore cores of the TC29x a separate thread is started. The example shows the situation where `Thread [core 0]` is suspended, `Thread [core 1]` is running and `Thread [core 2]` is suspended.

## Suspend or resume a thread/core

You can suspend or resume each of the threads independent from the other threads.

1.   Select the thread you want to suspend or resume.

2.   Click the **Suspend** (⏸) or **Resume** (▶) button.

In the example above `Thread [core 1]` has been selected (highlighted color). When you click the **Suspend** button or **Resume** button this only effects `Thread [core 1]`.

## Update views

When you select a function in a thread, the Source view, the Register view and the Disassembly view are updated to the contents of the thread (either running or suspended - with the latter situation up-to-date information is shown).

## Suspend or resume all threads/cores simultaneously

When you select the board, in the picture above `Infineon TriBoard TC29x - TC29x`, you can suspend or resume all cores simultaneously. However, when one of the threads is running and the others are suspended, you must suspend the running thread before you can resume all threads at once via the board selection method. The same applies for suspending all threads at once.

# 4.5. Programming a Flash Device

With the TASKING debugger you can download an application file to flash memory. Before you download the file, you must specify the type of flash devices you use in your system and the address range(s) used by these devices.

To program a flash device the debugger needs to download a flash programming monitor to the target to execute the flash programming algorithm (target-target communication). This method uses temporary target memory to store the flash programming monitor and you have to specify a temporary data workspace for interaction between the debugger and the flash programming monitor.

Two types of flash devices can exist: on-chip flash devices and external flash devices.

## Setup an on-chip flash device

When you specified a target configuration board using the New Project wizard or the Import Board Configuration wizard, as explained in Section 2.1, *Create a Project* and Section 2.2, *Configuring the Target*, any on-chip flash devices are setup automatically.
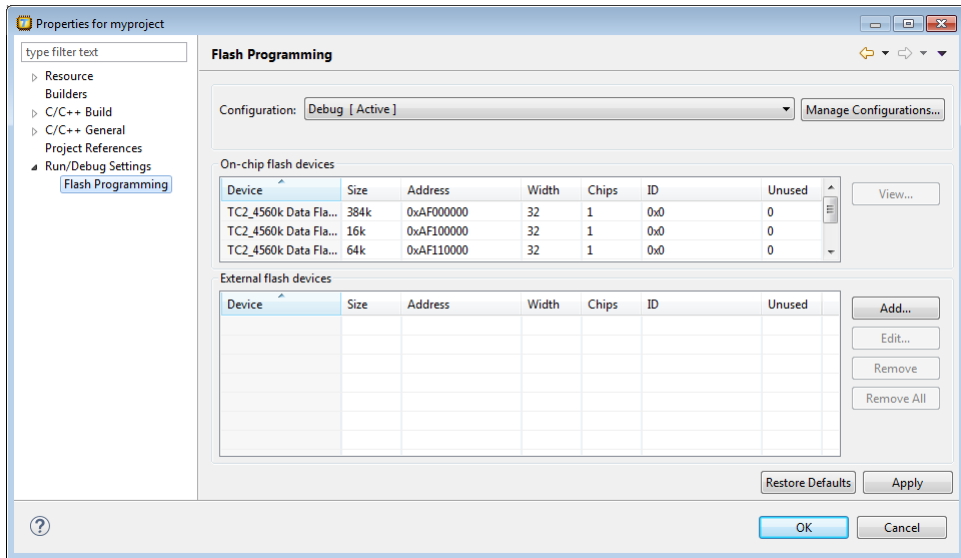
## Setup an external flash device

1.   From the **Project** menu, select **Properties for**

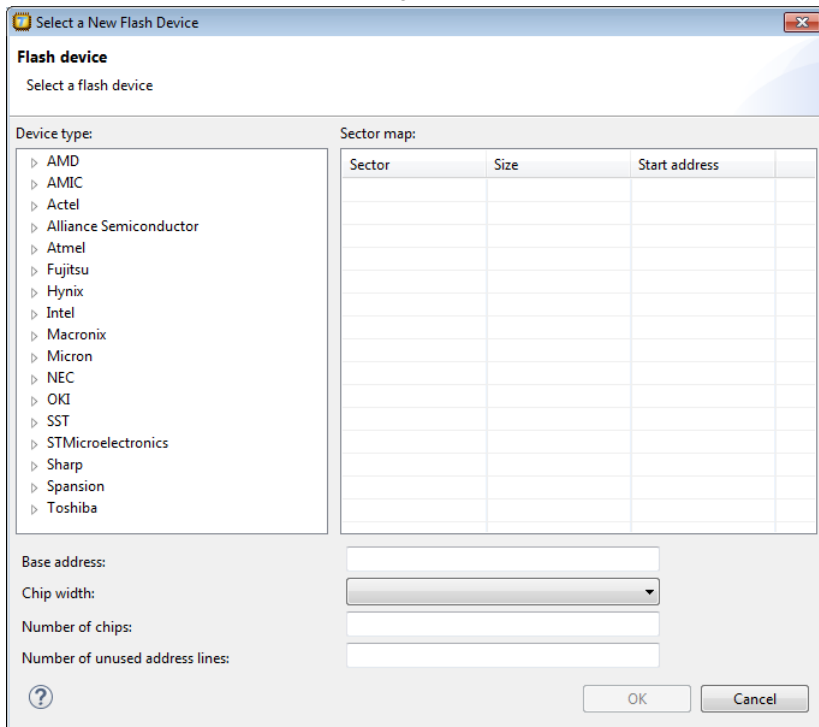     *The Properties for project dialog appears.*

2.   In the left pane, expand **Run/Debug Settings** and select **Flash Programming**.

     *The Flash Programming pane appears.*

3. Click **Add...** to specify an external flash device.

   *The Select a New Flash Device dialog appears.*

4.  In the **Device type** box, expand the name of the manufacturer of the device and select a device.

    *The Sector map displays the memory layout of the flash device(s). Each sector has a size and*

5.  In the **Base address** field enter the start address of the memory range that will be covered by the flash device. Any following addresses separated by commas are considered mirror addresses. This allows the flash device to be programmed through its mirror address before switching the flash to its base address.

6.  In the **Chip width** field select the width of the flash device.

7.  In the **Number of chips** field, enter the number of flash devices that are located in parallel. For example, if you have two 8-bit devices in parallel attached to a 16-bit data bus, enter 2.

8.  Fill in the **Number of unused address lines** field, if necessary.

The flash memory is added to the linker script file automatically with the tag "`flash=`*flash-id*".

## To program a flash device

1.  From the **Debug** menu, select **Debug Configurations...**

    *The Debug Configurations dialog appears.*

2.  In the left pane, select the configuration you want to change, for example, **TASKING C/C++ Debugger » myproject.board**.

3.  Open the **Initialization** tab

    *The Flash settings group box should be active.*

4.  Enable the option **Use default flash settings (recommended)**

    By default, the flash settings are derived from the `.dtc` file for the chosen target processor. So, when you change processors the flash settings change automatically. If you do not want that, you can specify your own flash settings. In that case perform steps 5-7, otherwise skip to step 8. You can click **Restore Defaults** to restore the default flash settings.

5.  In the **Monitor file** field, specify the filename of the flash programming monitor, usually an Intel Hex or S-Record file.

6.  In the **Sector buffer size** field, specify the buffer size for buffering a flash sector.

7.  Specify the data **Workspace address** used by the flash programming monitor. This address may not conflict with the addresses of the flash devices.

8.  Click **Debug** to program the flash device and start debugging.

## 4.5.1. Boot Mode Headers

Newer TriCore devices have (typically four) Boot Mode Headers (BMHs), which lie in flash memory. An individual BMH can be either valid or invalid. For certain devices, if all BMHs are invalid, the device is

normally inaccessible to the debugger. This more or less "bricks" the device because reprogramming the flash to revalidate one of the BMHs requires the debugger, or a similar program, to use the now inaccessible debug port. Recovery is possible via, for example, CAN, but this is cumbersome.

Therefore, the debugger has a special functionality to prevent all BMHs from being invalidated. The debugger only allows downloading if the target application adheres to one of the following restrictions.

• It contains at least one valid *non-ranged* BMH. "non-ranged" means that fields `ChkStart` and `ChkEnd` must be identical. This is a restriction needed for implementation reasons. You may use ranged BMHs, but they do not count as valid in this context.

• It contains no code or data in at least one of the 32-byte areas covered by a BMH. If necessary, you can instruct the linker to do this by explicitly reserving one of these ranges. The default LSL files provided with the product already do this.

### Remarks

• If your application does not contain any valid BMHs itself and would overwrite the last currently valid one, the debugger will silently validate one of the other BMHs. This means that the debugger may program more flash sectors than you might expect.

• Of course, your target application has the ability to reprogram flash memory for its own purpose. This too can cause bricking, but obviously the debugger cannot prevent this.

# Chapter 5. Debug Target Configuration Files

DTC files (Debug Target Configuration files) define all possible configurations for a debug target. A debug target can be target hardware such as an evaluation board or a simulator. The DTC files are used by Eclipse to configure the project and the debugger. The information is used by the Import Board Configuration wizard and the debug configuration.

## 5.1. Custom Board Support

When you need support for a custom board and the board requires a different configuration than those that are in the product, it is necessary to create a dedicated DTC file.

For details about the layout of DTC files, refer to the user guide of a TASKING VX-toolset product.

### To add a custom board

1.  Make a copy of a `.dtc` file and put it in your project directory (in the current workspace).

    *In Eclipse, the DTC file should now be visible as part of your project.*

2.  Edit the file and give it a name that reflects the custom board.

The Import Board Configuration wizard in Eclipse adds DTC files that are present in your current project to the list of available target boards.