
STM32CubeIDE user guide

Introduction

STM32CubeIDE is an all-in-one multi-OS development tool, and is part of the STM32Cube software ecosystem. It contains an advanced C/C++ development platform supporting software development of STM32-based products.

This document details the STM32CubeIDE features and usage, including how to get started, create and build projects, debug with standard and advanced techniques, and many other software analysis solutions. STM32CubeIDE is based on the Eclipse C/C++ Development Tools™ (CDT™) and GCC toolchain, which cannot be entirely described in this user manual. Additional information on Eclipse® is available from the STM32CubeIDE embedded help system. Special documents covering the details of the toolchain and GDB servers are included within the product.



1 Getting started

STM32CubeIDE supports STM32 products based on the Arm® Cortex® processor. Refer to STMicroelectronics documents listed in [Section 11 References](#) for details.

Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.



1.1 Product information

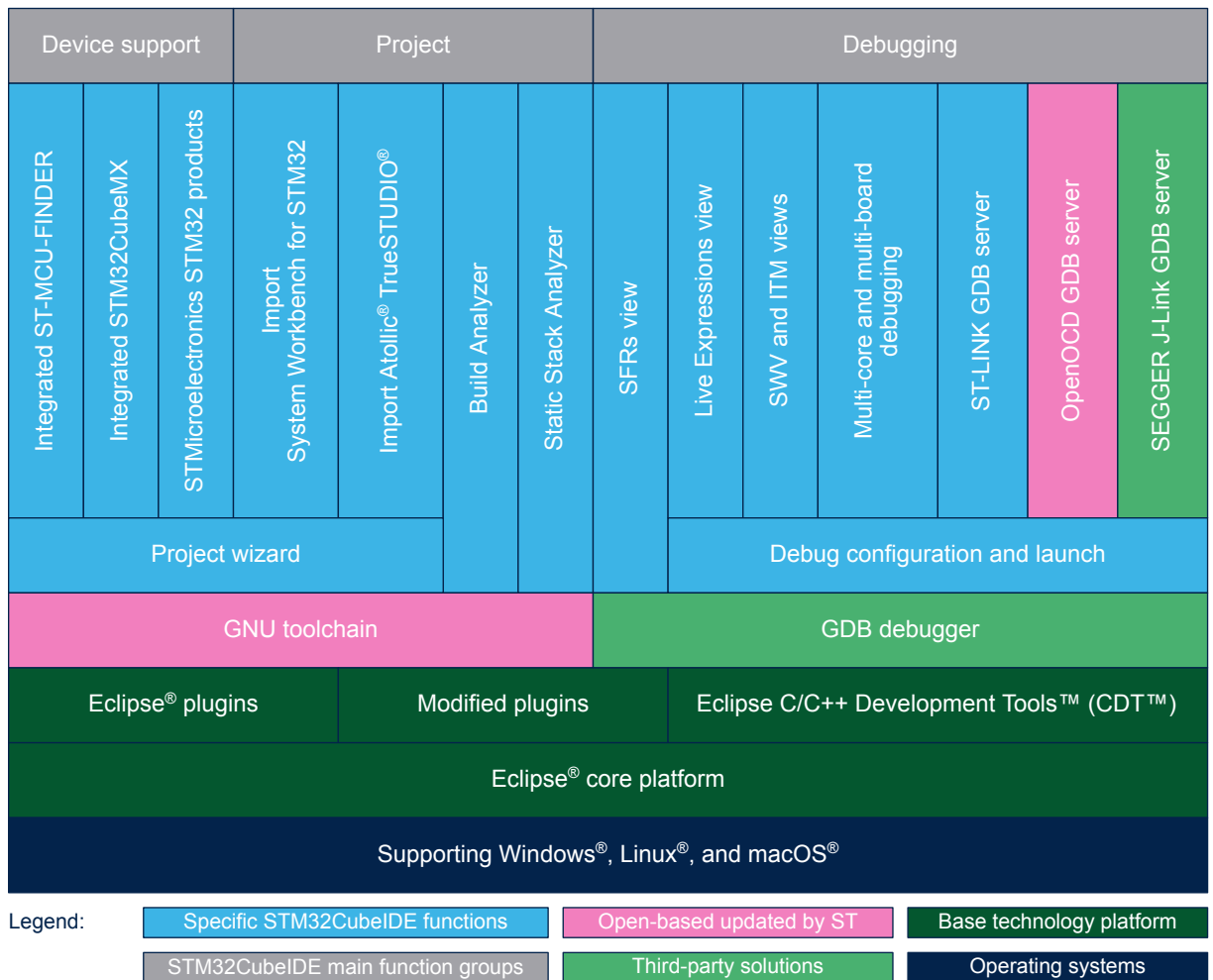
STM32CubeIDE is an advanced C/C++ development platform with peripheral configuration, code generation, code compilation, linking, and debug features. It is based on the Eclipse®/CDT™ framework and GCC toolchain for the development, and GDB for the debugging. It allows the integration of the hundreds of existing plugins that complete the features of the Eclipse® IDE.

STM32CubeIDE integrates ST MCUFinder (ST-MCU-FINDER-PC) and STM32CubeMX functionalities to offer all-in-one tool experience. It makes it easy to create new STM32 MCU or board projects and build them using the included GCC toolchain.

STM32CubeIDE includes a build analyzer and a static stack analyzer that provide the user with useful information about project status and memory requirements.

STM32CubeIDE also includes standard and advanced debugging features including views of CPU core registers, memories, and peripheral registers, as well as live variable watch, and serial wire viewer interface. A fault analyzer displays error information if an error is triggered by the STM32 processor during a debug session.

Figure 1. STM32CubeIDE key features



1.1.1 System requirements

STM32CubeIDE is tested and verified on the Microsoft® Windows®, Linux®, and macOS® operating systems.

Important:

STM32CubeIDE supports only 64-bit OS versions. For more details about supported versions of operating systems, refer to [ST-02].

Note:

macOS® is a trademark of Apple Inc. registered in the U.S. and other countries.

Linux® is a registered trademark of Linus Torvalds.

1.1.2 Downloading the latest STM32CubeIDE version

The latest version of STM32CubeIDE is available for free download from the www.st.com/stm32softwaretools website.

1.1.3 Installing STM32CubeIDE

The STM32CubeIDE installation guide [ST-04] gives directions on how to install on supported versions of Windows®, Linux® and macOS®. It is possible to have several versions of STM32CubeIDE installed in parallel. Read the installation guide if STM32CubeIDE is not already installed or if a new version must be installed. [Installing updates and additional Eclipse plugins](#) in this manual also provides information on how to install updates.

1.1.4 License

STM32CubeIDE is delivered under the *Mix Ultimate Liberty+OSS+3rd-party V1* software license agreement (SLA0048).

For more details about the license agreement of each component, refer to [ST-02].

1.1.5 Support

There are several different support options provided by STMicroelectronics. For instance, the ST Community is offering places to meet people with similar mind-set all over the world at any time. Choose the support option by visiting my.st.com/content/my_st_com/en/support/support-home.html.

1.2 Using STM32CubeIDE

1.2.1 Basic concept and terminology

The basic concept using STM32CubeIDE and Eclipse® terminology is outlined in this section.

Workspaces

When starting STM32CubeIDE, a workspace is selected. The workspace contains the development environment to be used. Technically, the workspace is a directory that may hold projects. The user may access any project within the active workspace.

A project contains files, which may be organized into sub-directories. Files existing somewhere else on the computer can also be linked to the project.

A single computer may hold several workspaces at various locations in the file system. The user may switch between workspaces, but only one workspace can be active at a time. Switching workspace is a quick way of switching from one set of projects to another.

In practice, the workspace and project model facilitate a well-structured hierarchy of workspaces, containing projects, which in turn contain files.

Information center

The first time STM32CubeIDE is started and a workspace is selected, the *Information Center* is opened. The *Information Center* provides quick access to start a new project, read STM32CubeIDE documentation, or get access to ST support & community. The *Information Center* can be easily accessed at any time via the *Information Center* toolbar button or from the *Help* menu.

Perspectives, menu bar, toolbar

When the *Information Center* is closed, STM32CubeIDE displays a perspective, which contains a menu bar, toolbar, views and editors. Each perspective is optimized for a special type of work. For instance, the *C/C++ perspective* is meant for creating, editing and building projects. The *Debug perspective* is intended to be used when debugging code on hardware.

Each perspective can be customized according to the user's need. It is possible to reset the perspective at any time if, for instance, too many views are opened or if the views are reordered. It is also possible to create new perspectives.

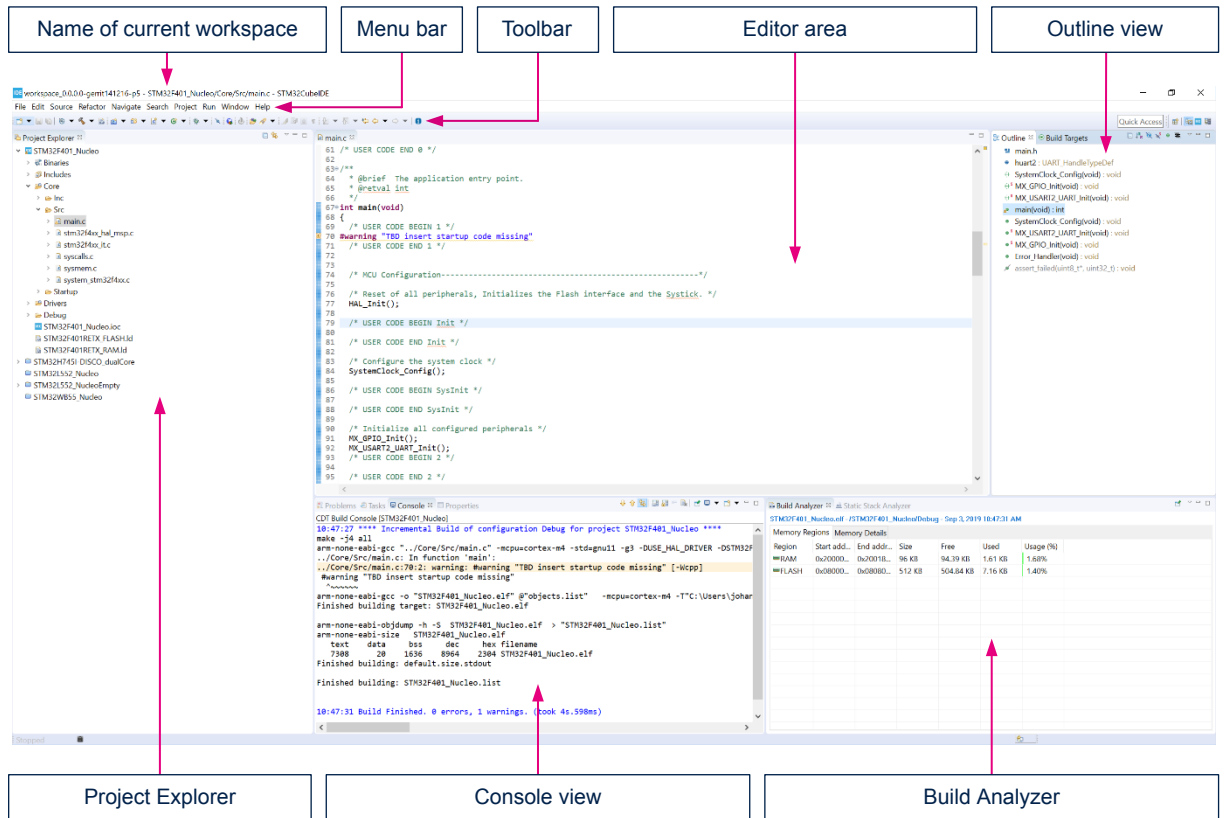
Views and editors

A perspective normally displays many views. Each view is developed to present specific information, which for instance can be collected from the project or from an embedded system under debug.

A perspective has one editor area. The editor can be used to edit project files. Many files can be edited in different tabs in the editor.

STM32CubeIDE window

Figure 2. STM32CubeIDE window



1.2.2

Starting STM32CubeIDE

Start STM32CubeIDE by performing the following steps depending on the operating system used.

Windows®

If a desktop shortcut is created during the installation of the product, the shortcut can be used to start STM32CubeIDE. The product can also be started from the Windows® start menu under STMicroelectronics programs.

Otherwise:

1. Locate where STM32CubeIDE is installed, for instance in C : \ST\STM32CubeIDE_1.0.2
2. Open the STM32CubeIDE folder
3. Start the stm32cubeide.exe program

Linux® or macOS®

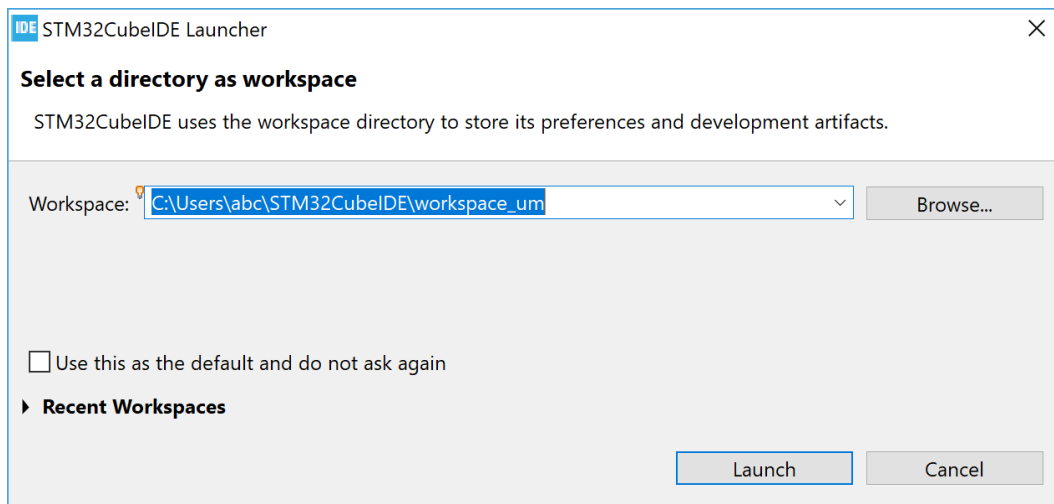
When using Linux® or macOS®, the program can be started in a similar way by opening the STM32CubeIDE folder where the product is installed.

STM32CubeIDE Launcher

When the product is started, it displays the STM32CubeIDE Launcher dialog with workspace selection. The first time the product is started, it presents a default location and workspace name. The dialog enables the user to select the name and location of the active workspace for holding all the projects currently accessible by the user. Any newly created project is stored in this workspace. The workspace is created if it does not yet exist.

Note: If Windows® is used, avoid locating the workspace folder too many levels below the file system root to avoid exceeding the Windows® path length character limitations. Build errors occur if the file paths become longer than what Windows® can handle.

Figure 3. STM32CubeIDE Launcher – Workspace selection

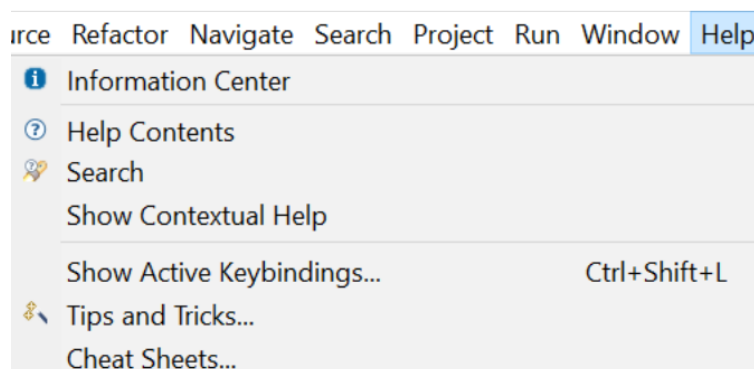


Click on the **[Launch]** button to launch STM32CubeIDE. The first time, it opens the *Information Center*, which is described in [Section 1.3 Information Center](#).

1.2.3 Help system

The *Help* menu provides several different help systems as seen in [Figure 4](#). The *Information Center* contains links to all available STM32CubeIDE documentation. It is also recommended for new users to try different Eclipse® built-in help systems to get an understanding of Eclipse® basics.

Figure 4. Help menu



1.3 Information Center

The *Information Center* provides quick access to:

1. Start a new project
2. Import an existing project
3. Read STM32CubeIDE documentation
4. Get access to *Getting Started with STM32CubeIDE (STM32CubeIDE quick start guide [ST-03])*
5. Explore the STM32 MPU and MCU wikis
6. Explore *What's New in STM32CubeIDE (STM32CubeIDE release note [ST-02])*

7. Get access to STMicroelectronics support and community on Twitter™, Facebook™, YouTube™, or ST community at community.st.com
8. Explore the STMicroelectronics application tools

It is not required to read all material before using the product for the first time. Rather, it is recommended to consider the *Information Center* as a collection of reference information to return to, whenever required.

1.3.1 Accessing the *Information Center*


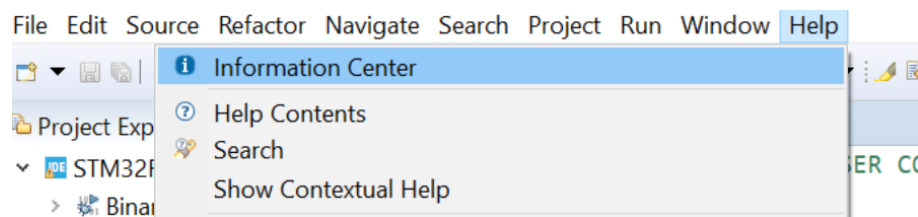
The *Information Center* can easily be accessed at any time, from any perspective, using the [Information Center] toolbar button . This icon is located at the right of the toolbar. It is also possible to open the *Information Center* from the [Help]>[Information Center] menu command.

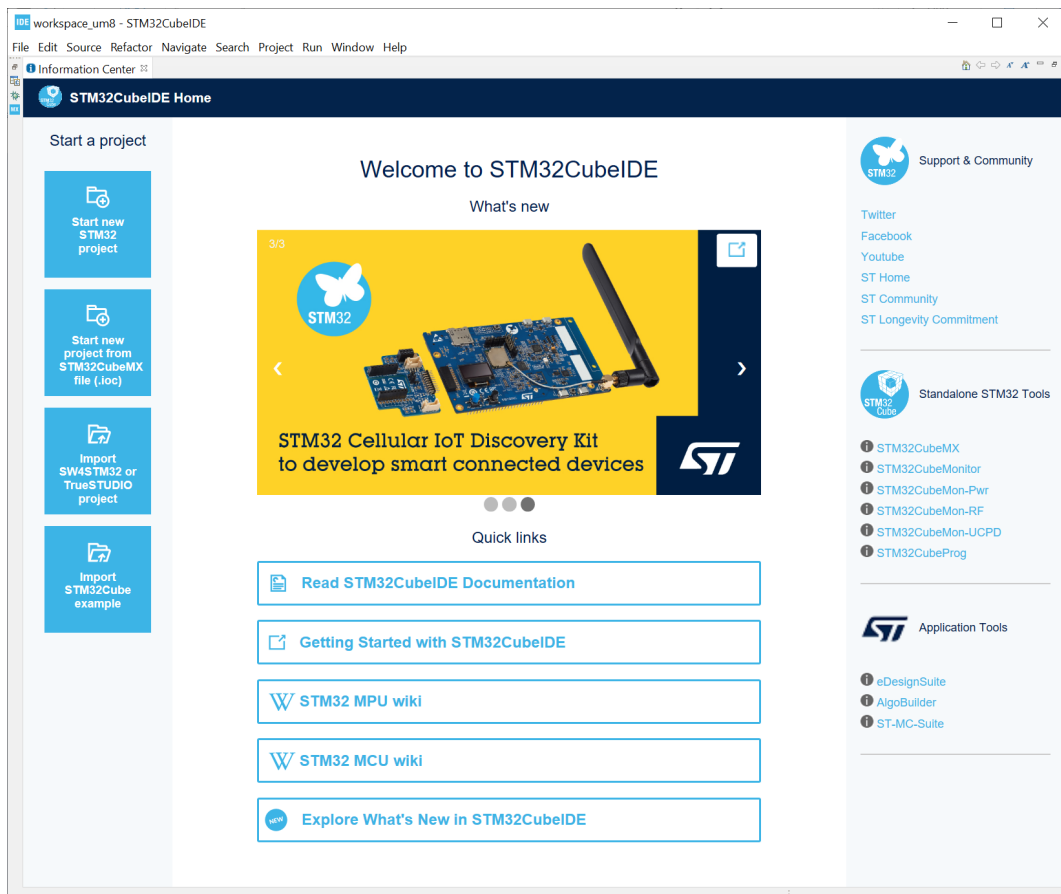
Figure 5. Help - Information Center menu




1.3.2 Home page

When the *Information Center* is opened, the *Home* page is displayed. It contains links to start a new project, import projects, read documentation and access STMicroelectronics support and community.

Figure 6. Information Center – Home page

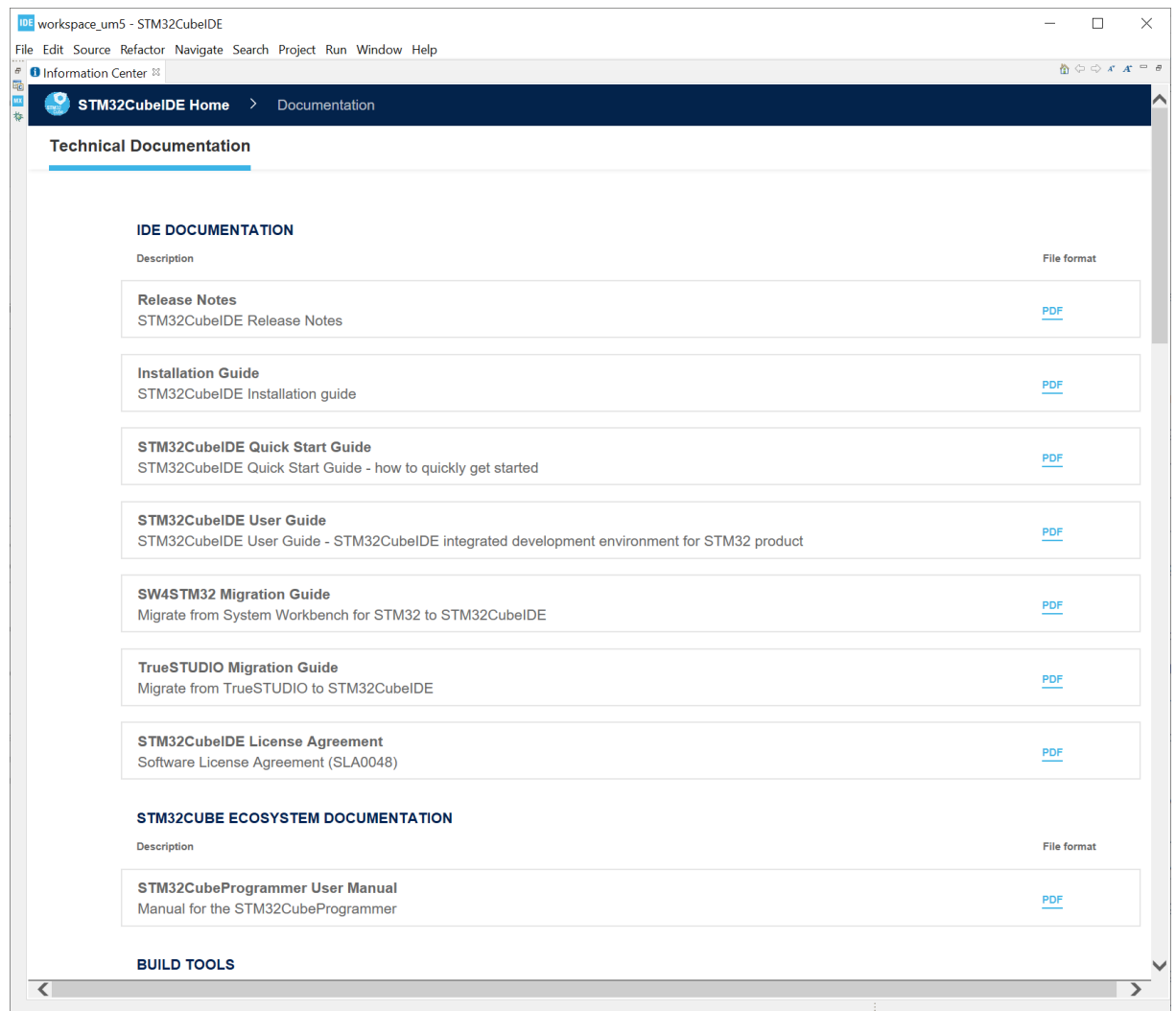


When using an old workspace, the *Information Center* may not display valid information, showing “*This page can't be displayed*” or opening old manuals when accessing documents. In such case, reload the page by clicking on the [Home] button  at the top right corner of the *Information Center* window.

1.3.3 Technical documentation

The *Information Center* also contains a *Technical Documentation* page, which is opened from the *Home* page when clicking on the *Read STM32CubeIDE Documentation* link.

Figure 7. Information Center – Technical documentation page



Scroll through the *Technical Documentation* page and click on a document in the list to open it. The documents are listed in groups:

- IDE documentation
- [STM32Cube](#) ecosystem documentation
- Build tools
- Debugger documentation
- Toolchain manuals (*GNU Tools for STM32*)

Note: *STM32CubeIDE is distributed with the “GNU Tools for STM32” toolchain. This toolchain is built from the “GNU ARM Embedded” toolchain with patched enhancements. The first versions of STM32CubeIDE also included the “GNU ARM Embedded” toolchain, which can be installed from the STM32CubeIDE update site. If additional toolchains are installed, their respective documents are also listed in the Technical Documentation page in the Information Center. Information about patches made in “GNU Tools for STM32” can be read in [EXT-12]. The document can be opened from the Technical Documentation page in the Information Center.*

To navigate back to the *Home* page, press *STM32CubeIDE Home* at the top left side of the *Information Center*.

1.3.4 Closing the *Information Center*

Close the *Information Center* tab to close the *Information Center*.

1.4 Perspectives, editors and views

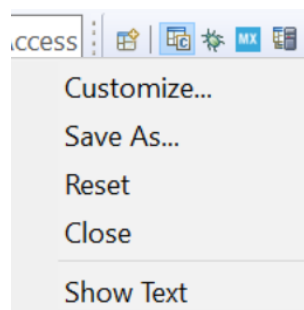
STM32CubeIDE is a powerful product with many views, loaded with various features. Displaying all views simultaneously would overload the user with information that may not be relevant to the task at hand.

To overcome such a situation, views can be organized in perspectives, where a perspective contains a number of predefined views and an editor area visible by default. A perspective typically handles one development task, such as *C/C++ Code Editing* or *Debugging*.

1.4.1 Perspectives

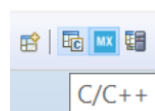
The perspectives can be customized according to the user's need; Views can be moved, resized and new views can be opened. It is possible to reset the perspective at any time if, for instance, too many views are opened or if the views are reordered. The perspective is reset by right-clicking the perspective icon in the toolbar and selecting **[Reset]** from the list. This resets the views; Added views in the perspective are closed and the default views are moved to their original location.

Figure 8. Reset perspective



As seen in [Figure 8](#), it is also possible to customize a perspective and save the perspective with a new name. Switching from one perspective to another is a quick way to hide some views and display others. To switch perspective, select the **[Open Perspective]** toolbar buttons at the right of the toolbar.

Figure 9. Toolbar buttons for switching perspective

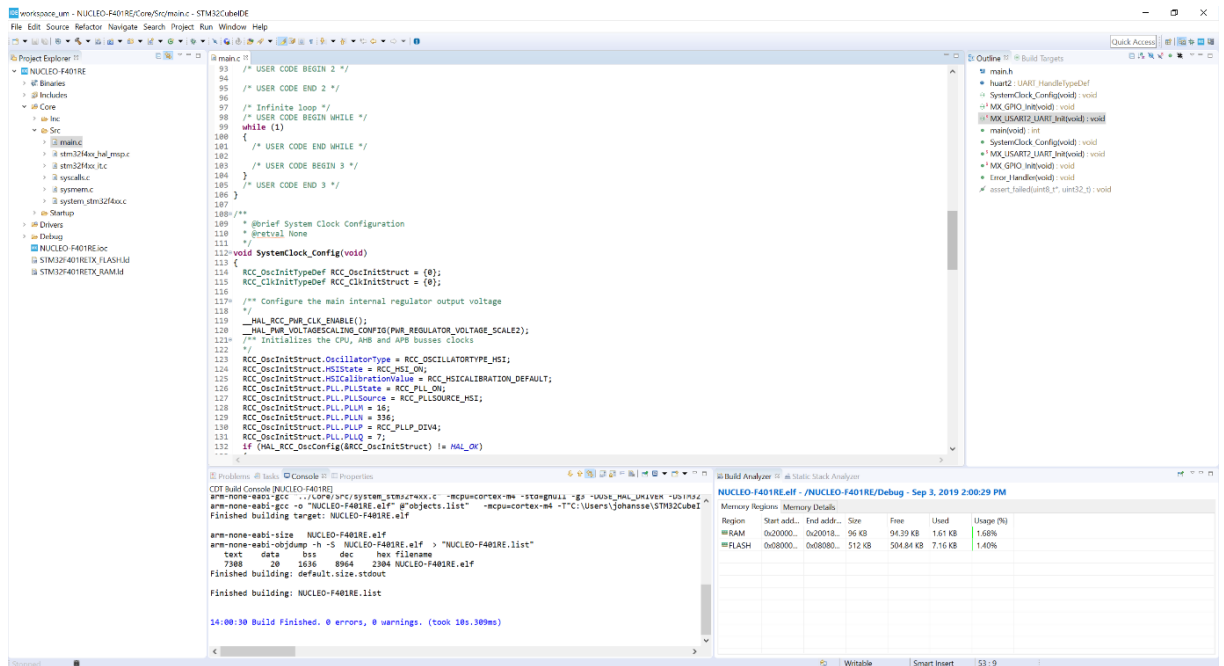


Another way to switch perspective is to use the menu command **[Window]>[Perspective]>[Open Perspective]>[Other...]** and select the perspective to use.

1.4.1.1 C/C++ perspective

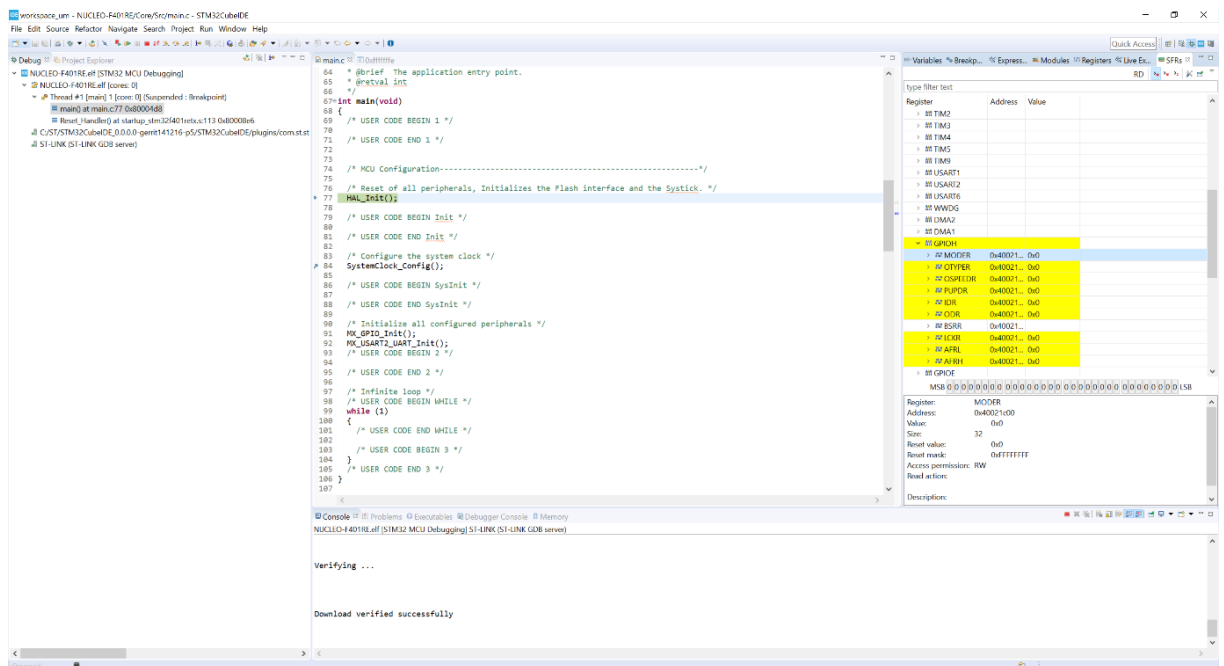
The C/C++ perspective is intended for creating new projects, editing files, and building the project. The left part of the perspective contains the *Project Explorer* view. The editor is located in the middle. The right part contains some views for the project (*Outline* and *Build Targets* views). At the bottom in the example illustrated in [Figure 10](#), there are the *Problems*, *Tasks*, *Console* and *Properties* views. At the lowest right, the *Build analyzer* and *Static stack analyzer* views are displayed.

Figure 10. C/C++ perspective



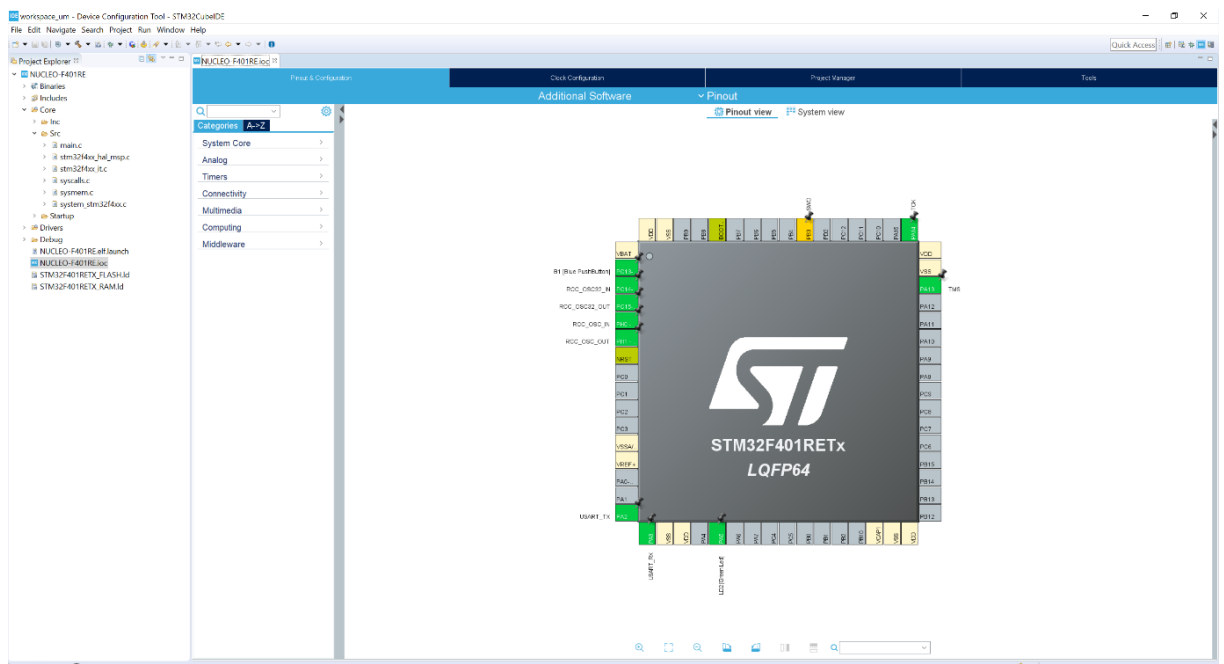
1.4.1.2 Debug perspective

The *Debug* perspective is intended for debugging the code. The *Debug* perspective is normally opened automatically when a new debug session is started. Later, when the debug session is closed, the perspective is switched back to the C/C++ perspective.

Figure 11. Debug perspective


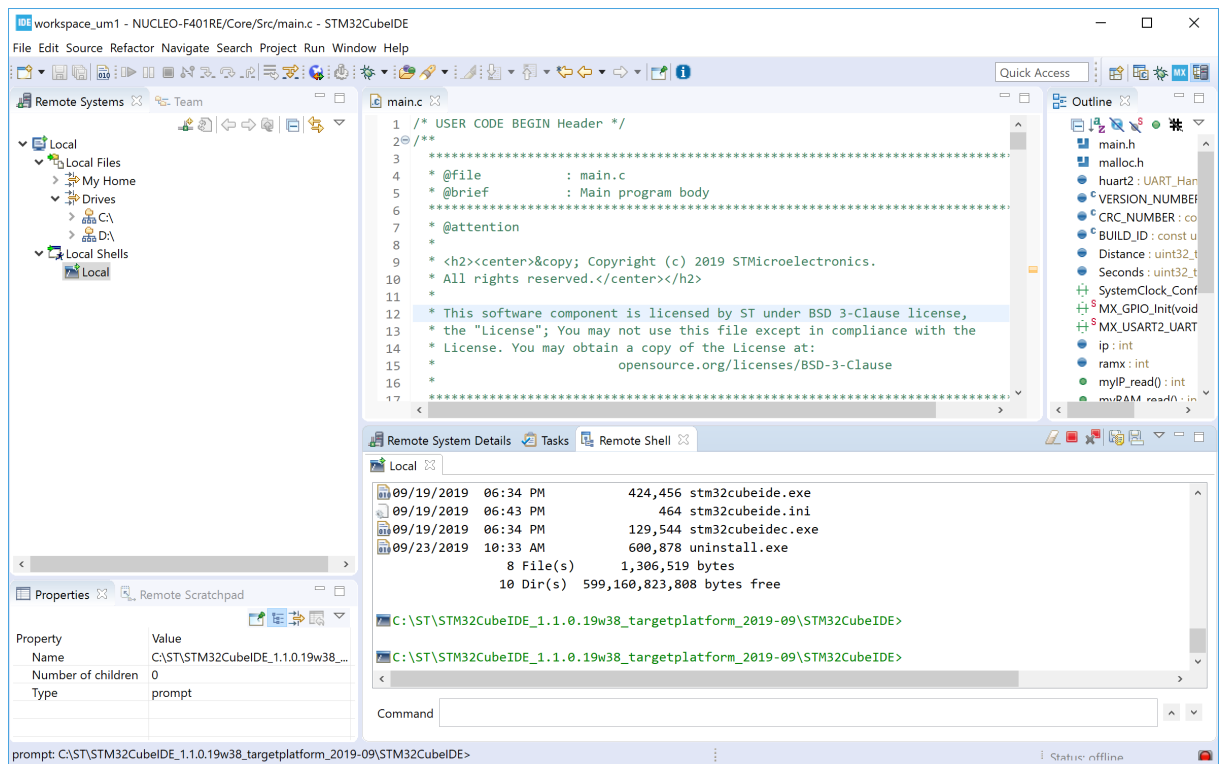
1.4.1.3 Device Configuration Tool perspective

The *Device Configuration Tool* perspective contains the STM32CubeMX device configuration tool integrated in STM32CubeIDE. This perspective is used for device configuration. When an *.ioc file is opened in an editor and the *Device Configuration Tool* perspective is used, the device can be configured in this perspective. How the device configuration is made is described in [ST-14].

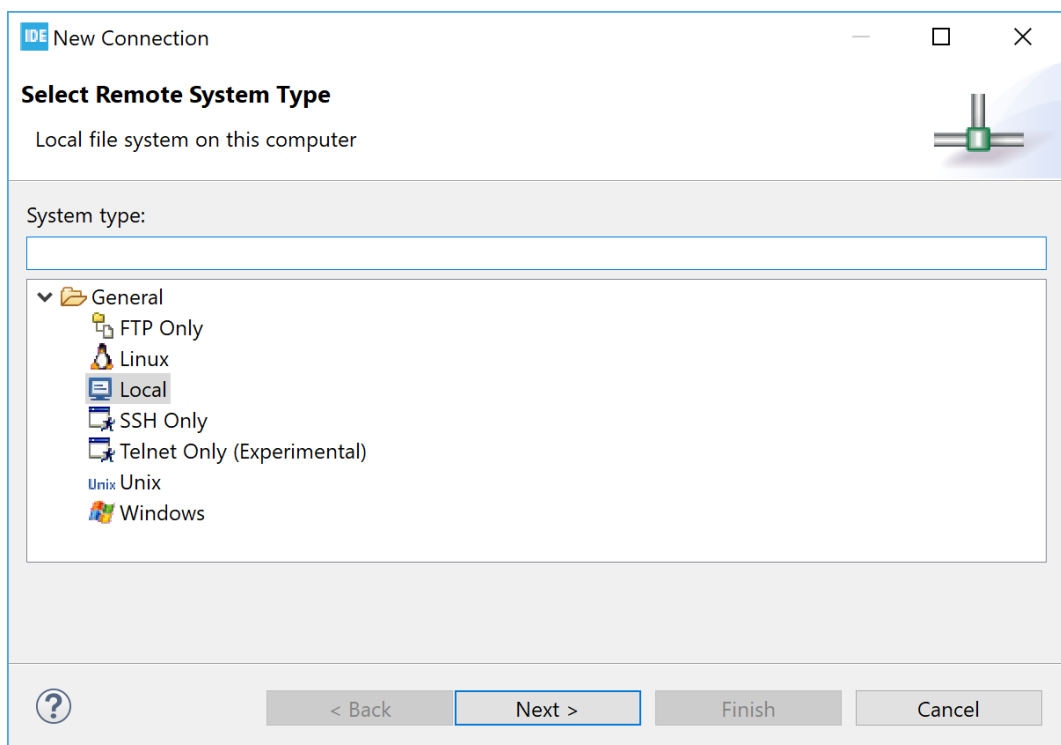
Figure 12. Device Configuration Tool perspective


1.4.1.4 Remote System Explorer perspective

The *Remote System Explorer* perspective is basically used when developing STM32 Arm® Cortex® MPU-based systems. The *Remote Systems* view is used to view files and the *Remote Shell* view is used to run commands.

Figure 13. Remote System Explorer perspective


The *Remote Systems* view contains buttons to open a new connection via FTP, Linux®, Local, SSH, Telnet and others.

Figure 14. New connection


1.4.2 Editors

The editor area in a perspective is used by editors. Any number of editors can be opened simultaneously but only one can be active at a time. Different editors can be associated with different file extensions. Example of editors are; c-editor, linker script editor, ioc-file editor for STM32CubeMX device configuration.

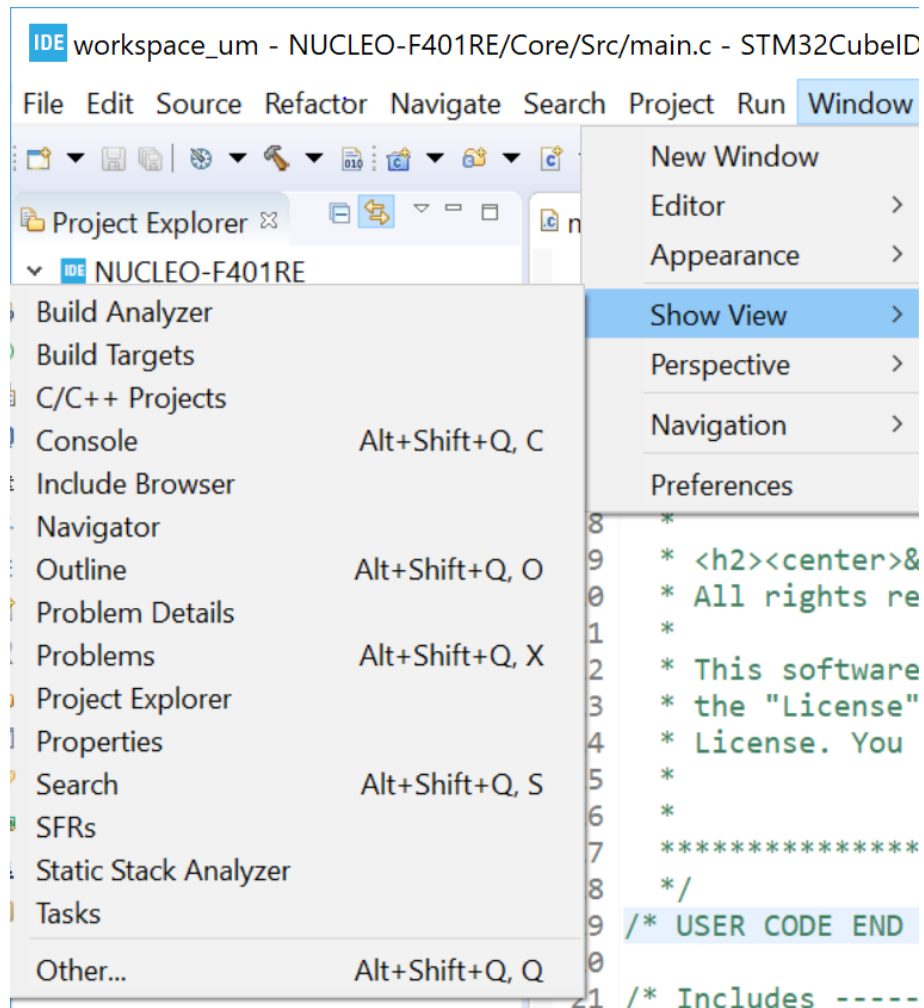
To open a file in the editor, double-click on the file in the *Project Explorer* view or open the file via the **[File]** menu. When a file is modified in the editor, it is displayed with an asterisk (*) indicating that the file has unsaved changes.

1.4.3 Views

Only the most common views associated with the perspective are displayed by default. There are many more views in the product supporting different features. Some of these views only provide valid data when a debug session is ongoing, while others always display data.

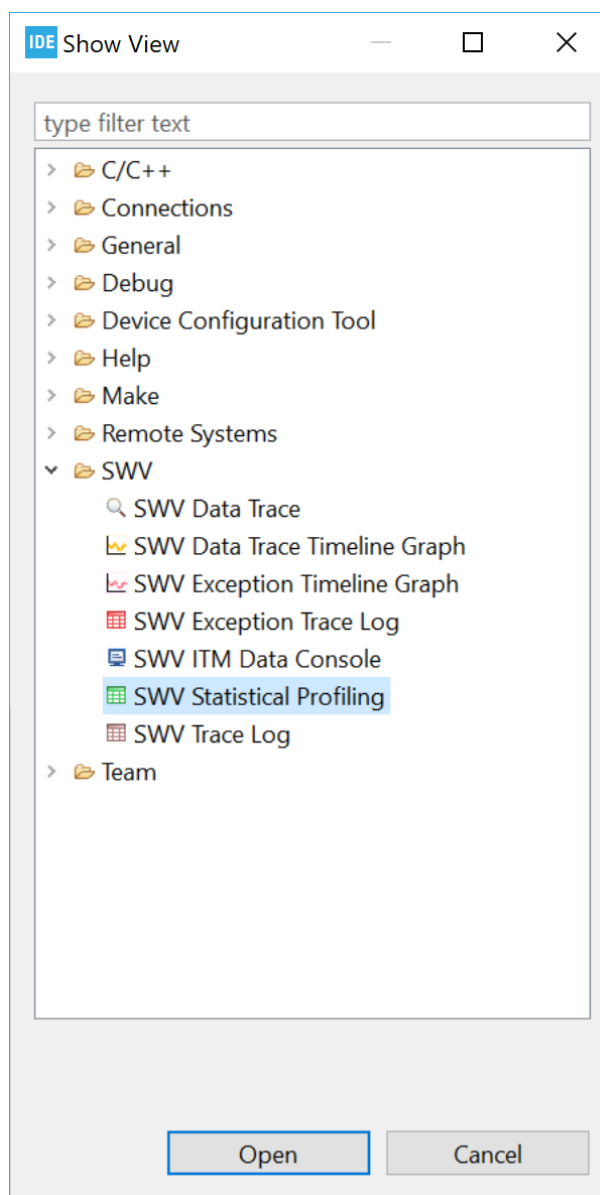
Views can be opened from the **[Window]>[Show View]** menu by selecting one of the views in the list.

Figure 15. [Show View] menu



The above list of views in Figure 15 is still not complete. It contains only the most common views for the work task related to the perspective currently selected. To access even more views, select [**Other...**] from the list. This opens the *Show View* dialog box. Double-click on any view to open it and access its additional features.

Figure 16. Show View dialog



The views can be resized and their positions can be changed: Simply drag the view to a new place in STM32CubeIDE. The view can also be dragged outside the STM32CubeIDE window on the screen. Such detached views are shown in separate windows. Detached views works like the other views but are always shown in front of the workbench. Detached views can be attached again by dragging the tab in the detached view into the STM32CubeIDE window.

To restore the perspective to original state, right-click the perspective icon in the toolbar and select [**Reset**] from the list. Another way to reset the perspective is to use the menu [**Window**]>[**Perspective**]>[**Reset Perspective**].

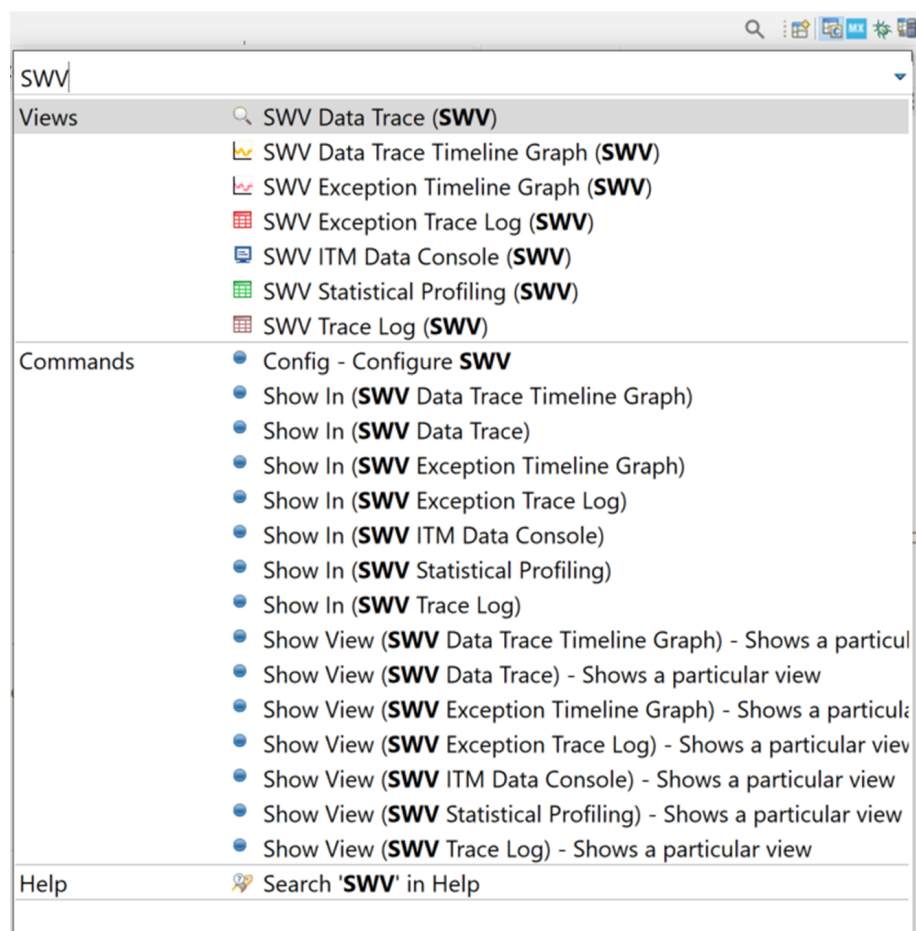
1.4.4 Quick Access edit field

The magnifying glass in the toolbar opens the *Quick Access* text box, where any search phrase or keyword can be entered. GUI objects like menu commands, toolbar buttons, preference settings or views can be found using the text box. As any search string is typed, the *Quick Access* shows all the GUI objects that match the criteria, in real time. Type a couple of characters or more and see how the list of results is refined correspondingly on-the-fly.

The *Quick Access* is a time saver when looking for a specific GUI object that cannot be found quickly otherwise, such as a preference setting deeply buried in the configuration dialogs. It is also convenient to retrieve a menu command or toolbar button hidden in the currently active perspective.

For example, in [Figure 17](#), the search string “SWV” entered in the *Quick Access* provides immediately the list of matching views, GUI commands and preference settings. To open the view or preference setting, click on the GUI object in the search result list.

Figure 17. Quick access

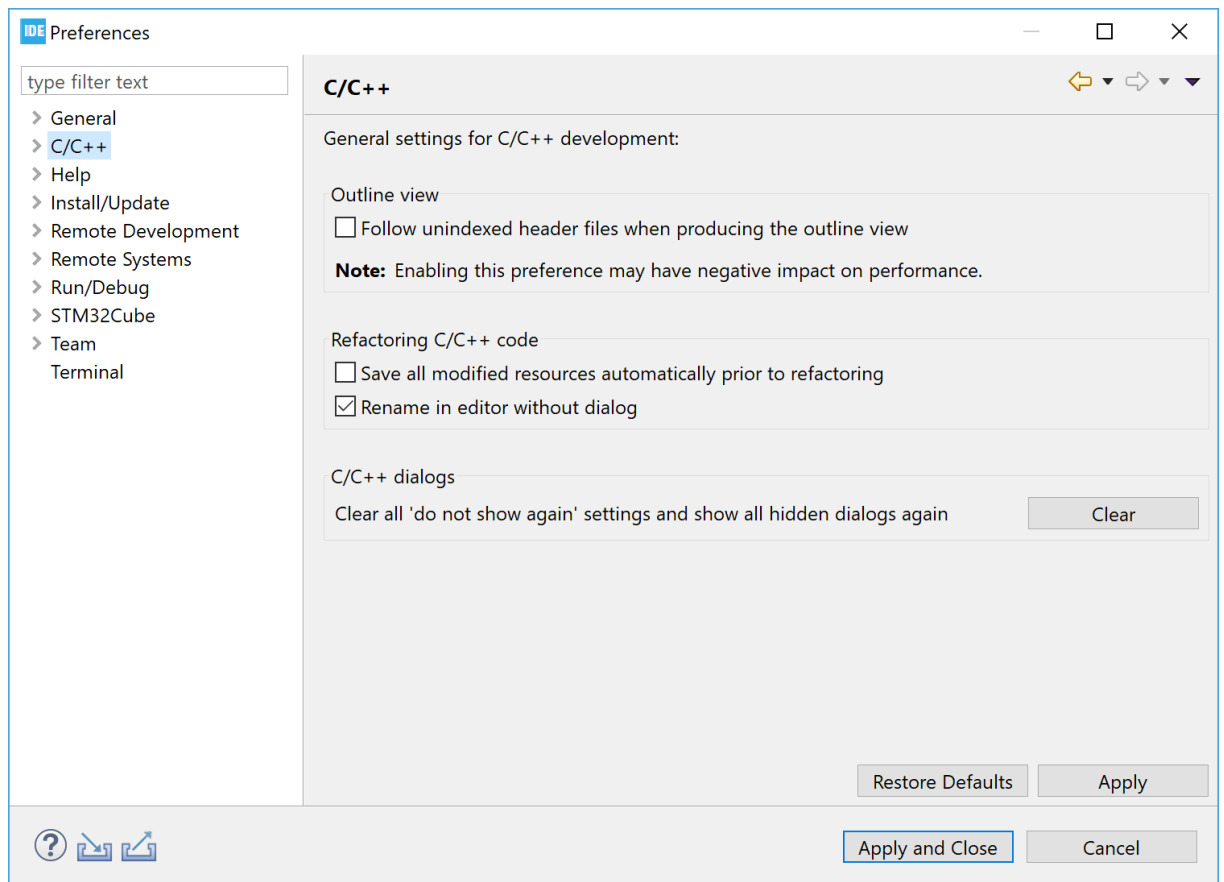


1.5 Configuration - Preferences

STM32CubeIDE can be customized in many ways. The menu **[Window]>[Preferences]** is used to open the *Preferences* dialog. In this dialog, the left pane is used to navigate to certain preference pages. There is also a filter field, which can be used to narrow down the content displayed. The arrow controls on the upper-right side of the dialog can be used to navigate back and forth across pages. The right pane contains the setting of the displayed preferences. Make any preferred change and press **[Apply]** to update the setting.

[Restore Defaults] resets all changes. The preference settings are stored in a metadata folder in the workspace of the application. [Section 1.7 Managing existing workspaces](#) in this user manual provides information on how to backup preferences and copy preferences across workspaces.

Figure 18. Preferences



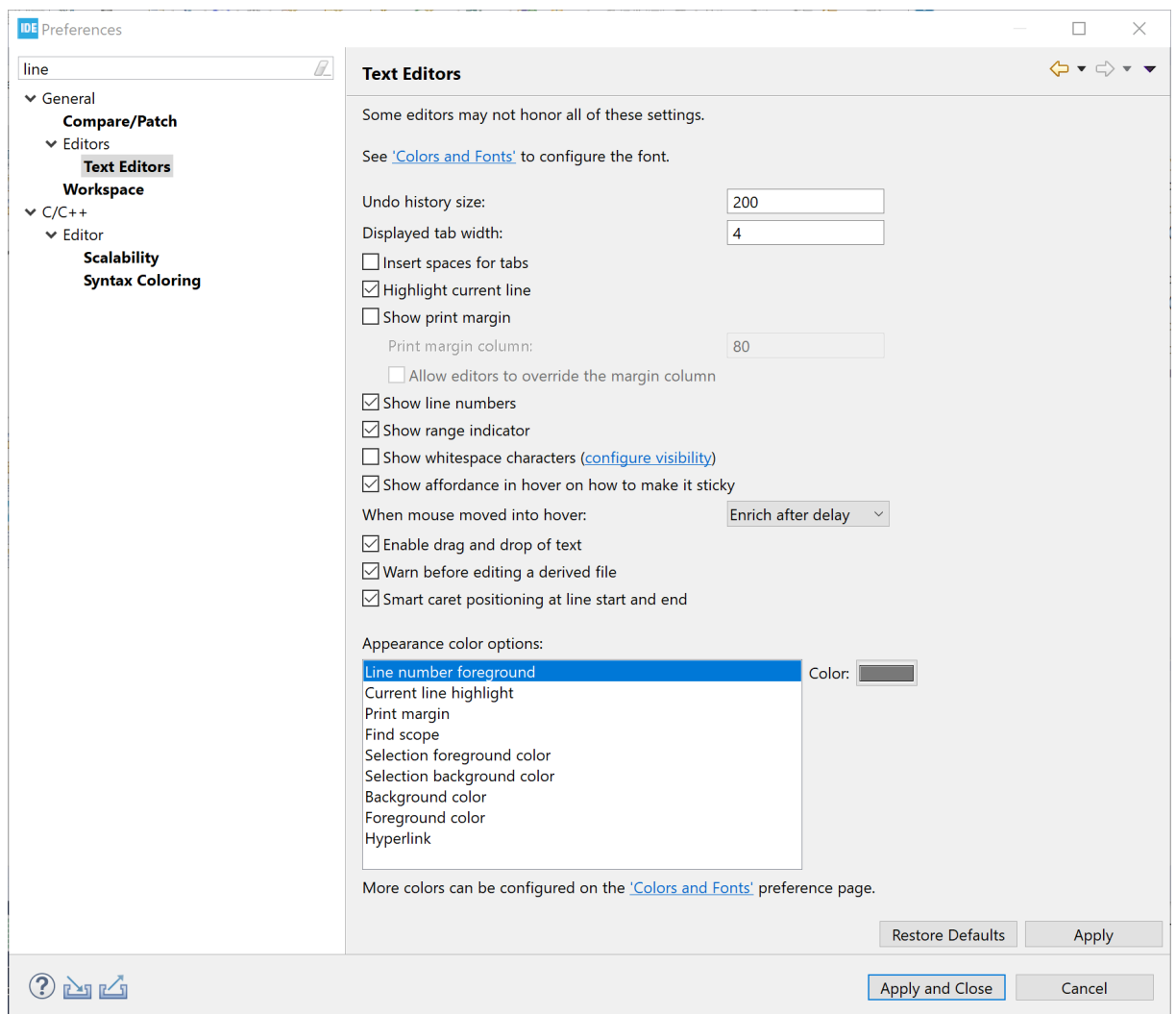
It is advised to walk through the preferences pages and get an understanding of the possible configuration options. The following sections present some of them.

1.5.1 Preferences - Editors

The editor can be configured in many ways. For instance, the menu selection **[General]>[Editors]>[Text Editors]** provides a *Preferences* pane containing general editor settings such as:

- Displayed tab width
- Insert spaces for tabs
- Highlight current tab
- Show line numbers
- Others

Figure 19. Preferences - Text Editors

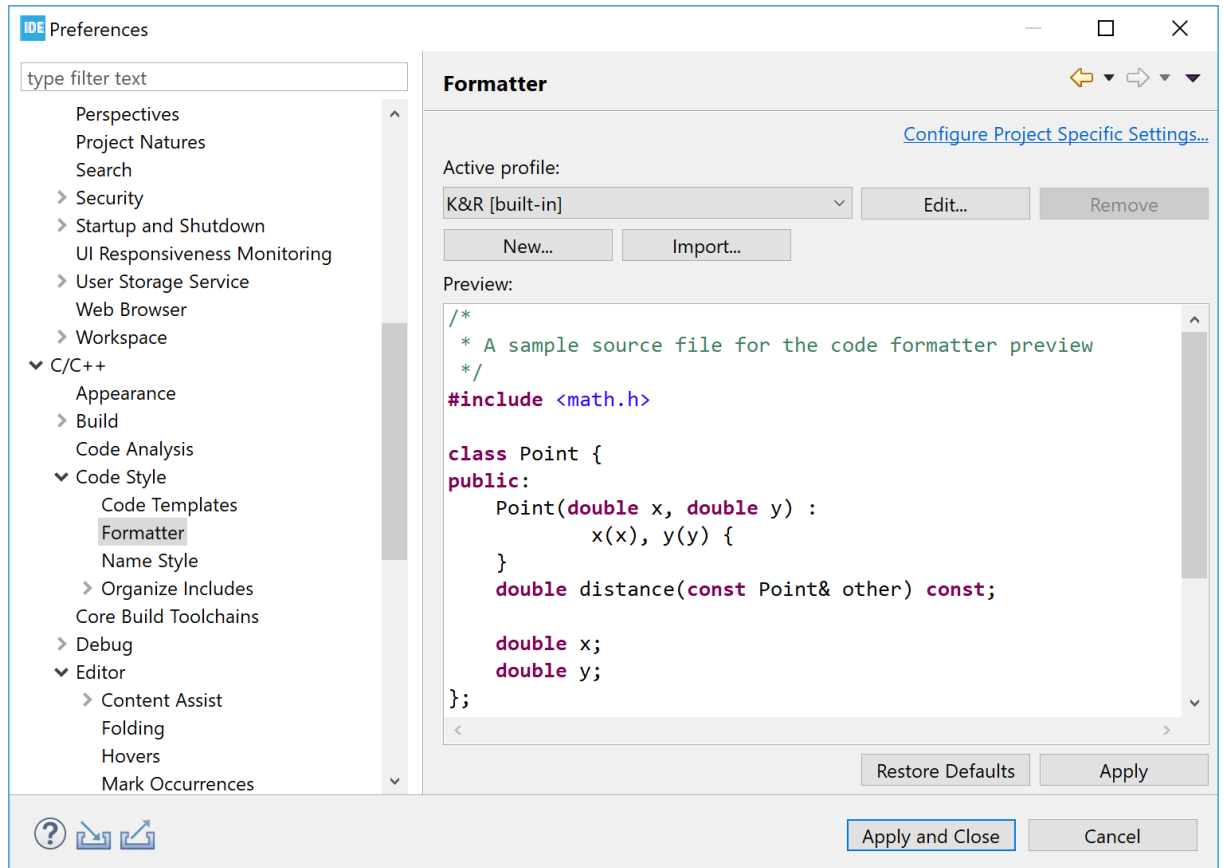


1.5.2 Preferences - Code style formatter

It is possible to configure the editor to use special formatting.

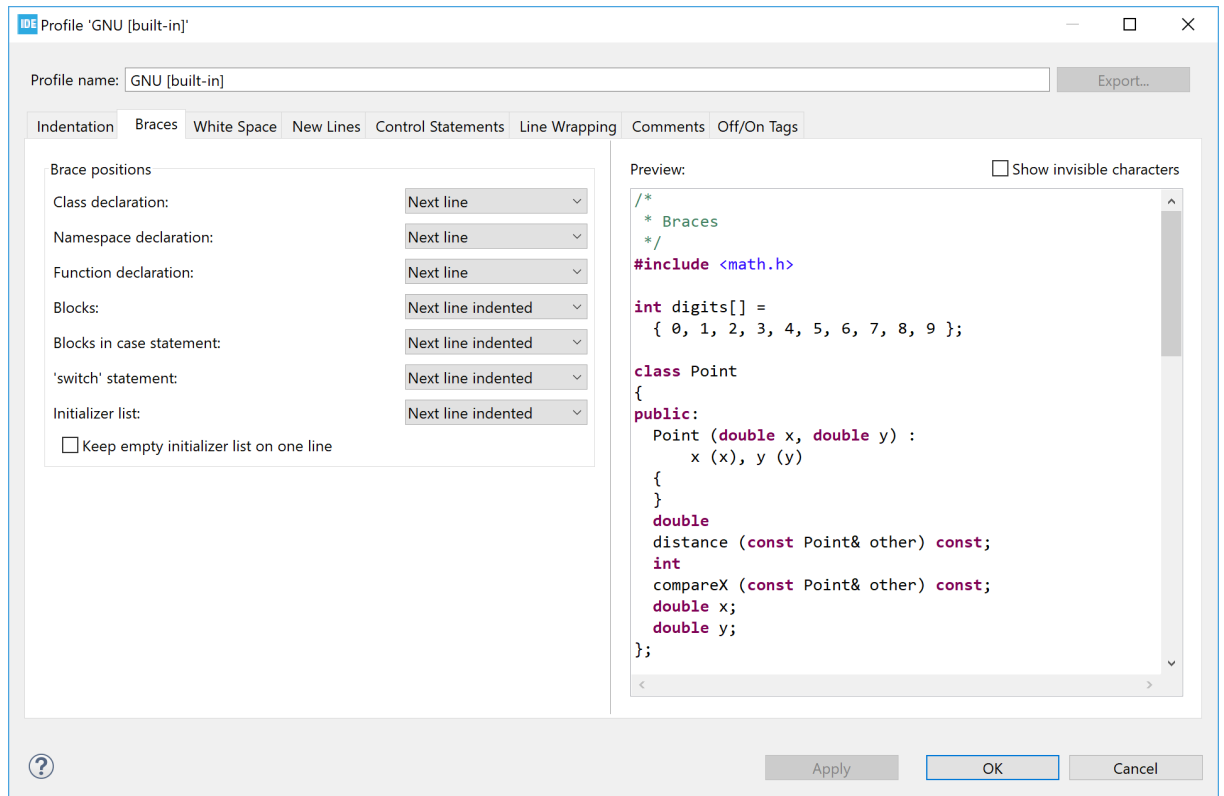
The menu selection [C/C++]>[Code Style]>[Formatter] provides a *Preferences* pane containing settings to set an active profile.

Figure 20. Preferences - Formatter



At this point, if **[Edit...]** is pressed, a new dialog is opened, where the selected profile can be updated according to specific coding rules. This is displayed in [Figure 21](#).

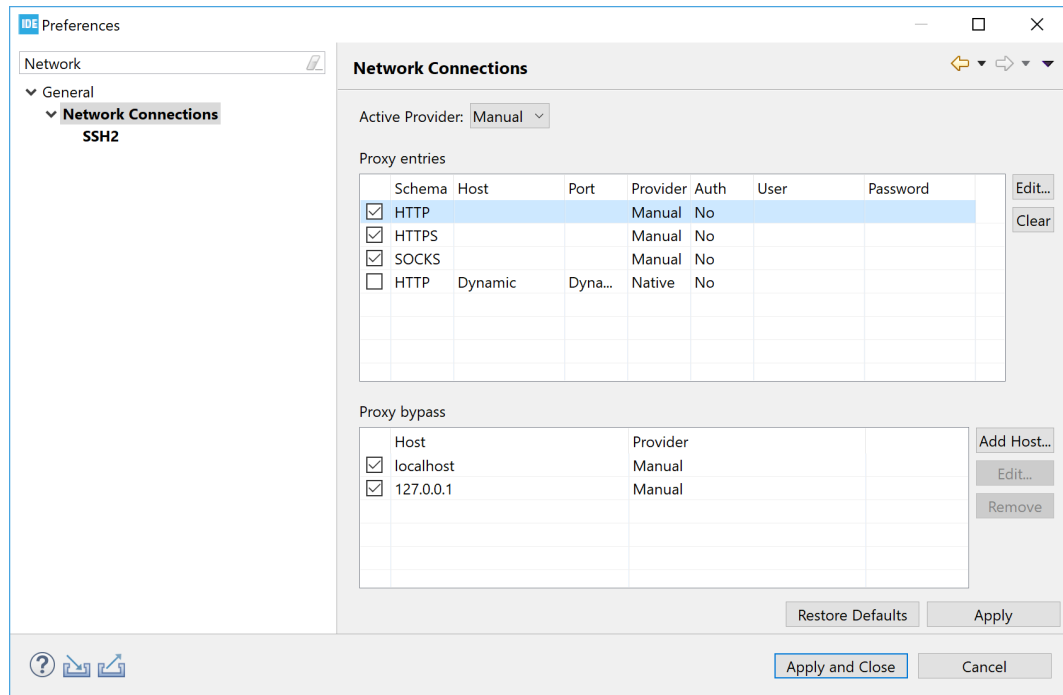
Figure 21. Preferences - Code style edit



1.5.3 Preferences - Network proxy settings

STM32CubeIDE uses the Internet for instance to get access to STM32 devices information. If a proxy server is used for Internet access, some configuration settings are required in STM32CubeIDE. The proxy settings are set in the *Preferences* pane obtained through [General]>[Network Connections]. To change the settings, set [Active provider] to *Manual* and update the *Proxy entries* for HTTP and HTTPS with specific *Host*, *Port*, *User* and *Password* using the [Edit...] button.

Figure 22. Preferences - Network Connections



- Note:** If there is a problem to save the proxy settings, the reason can be a corrupt *secure_storage* file. Proceed as follows to solve the problem:
1. Close all running STM32CubeIDE applications
 2. Rename file *C:\Users\user_name\.eclipse\org.eclipse.equinox.security\secure_storage* to a new name
 3. Restart STM32CubeIDE
 4. Update the proxy network settings, with user and password information, and save them to create a new *secure_storage* file

1.5.4 Preferences - Build variables

The STM32CubeIDE preferences feature build variables that are only visible in the IDE.

The menu selection [C/C++]>[Build]>[Build Variables] provides a *Preferences* pane with *Build Variables*, which can be used as $\${VAR}$ in STM32CubeIDE. Enable [Show system variables] to display all available variables.

Figure 23. Preferences – Build variables

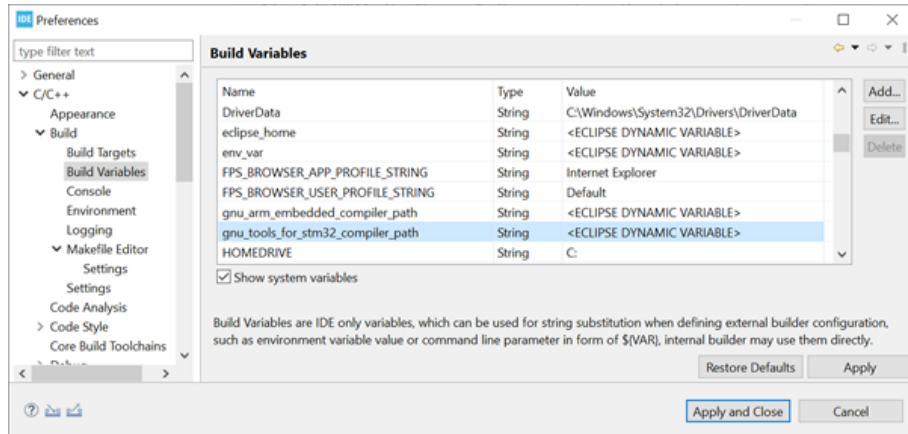
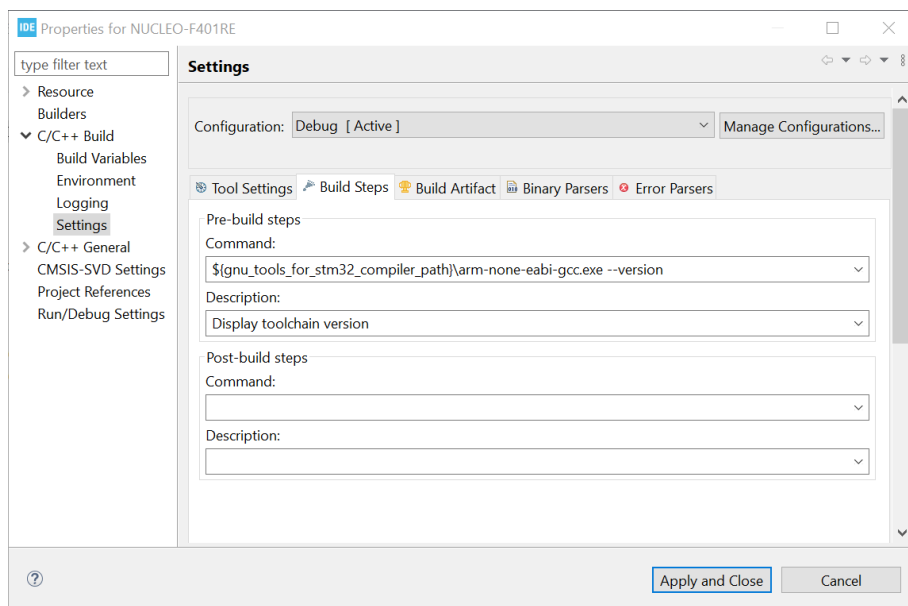


Table 1. Examples of toolchain build variables

Variable	Description
<code>gnu_tools_for_stm32_compiler_path</code>	Path to GNU Tools for STM32 toolchain.
<code>gnu_arm_embedded_compiler_path</code>	Path to GNU Arm Embedded toolchain.
<code>stm32cubeide_make_path</code>	Path to make and BusyBox.

A pre-build step example using build variables to display toolchain version is given in Figure 24.

Figure 24. Pre-build step using build variables



1.6 Workspaces and projects

The basic concepts of workspaces and projects compares as follows:

- A workspace contains projects. Technically, a workspace is a directory containing project directories or references to them.
- A project contains files. Technically, a project is a directory containing files that may be organized in sub-directories.
- A single computer may hold several workspaces at various locations in the file system. Each workspace may contain several projects.
- The user may switch between workspaces, but only one workspace can be active at one time.
- The user may access any project within the active workspace. Projects located in another workspace cannot be accessed, unless the user switches to that workspace.
- The files included in a project do not need to be physically located in a folder in the project but can be located somewhere else and linked into the project.
- Switching workspaces is a quick way of shifting from one set of projects to another. It triggers a quick restart of the product.

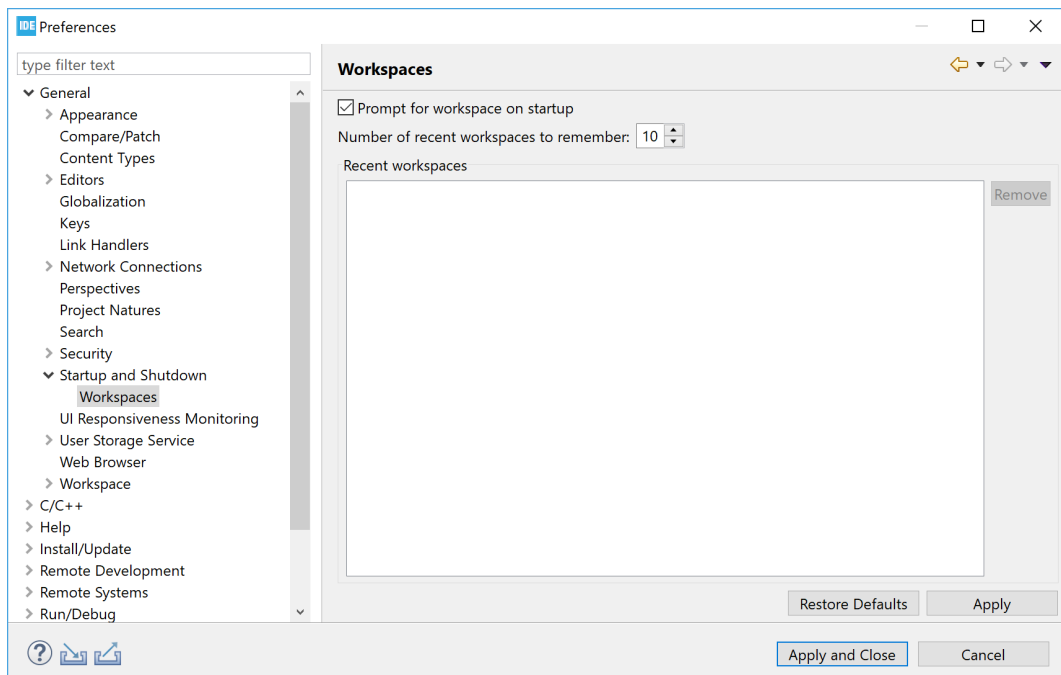
In practice, the project and workspace model facilitates a well-structured hierarchy of workspaces, containing projects, containing files.

1.7 Managing existing workspaces

The workspace can be selected when starting STM32CubeIDE. It is also possible to switch to another workspace during the use of STM32CubeIDE. In this case STM32CubeIDE restarts after the new workspace is selected. To restart STM32CubeIDE with a new workspace, select menu **[File]>[Switch Workspace]**.

The workspaces known to STM32CubeIDE can be managed by selecting **[Window]>[Preferences]** then, in the *Preferences* dialog, selecting **[General]>[Startup and Shutdown]>[Workspaces]**. In the right pane, it is possible to enable **[Prompt for workspace on startup]** and set **[Number of recent workspaces to remember]** to the desired value.

Figure 25. Preferences - Workspaces



It is also possible to select and remove recent workspaces from the list of recent workspaces. However, removing a workspace from that list does not remove the files. Neither does it remove the files from the file system.

1.7.1 Backup of preferences for a workspace

It is generally a good practice to take a copy of the existing preferences for a workspace. It can be especially useful to recreate the workspace after a crash without the time-consuming process to redo the settings manually. In the menu, select **[File]>[Export]**. Then, in the panel, select **[General]>[Preferences]**. Press the **[Next]** button and, in the next page, enable **[Export All]** along with a correct filename.

1.7.2 Copy preferences between workspaces

To copy workspace preferences from one workspace to another, an existing export of preferences must first be created as explained in [Backup of preferences for a workspace](#).

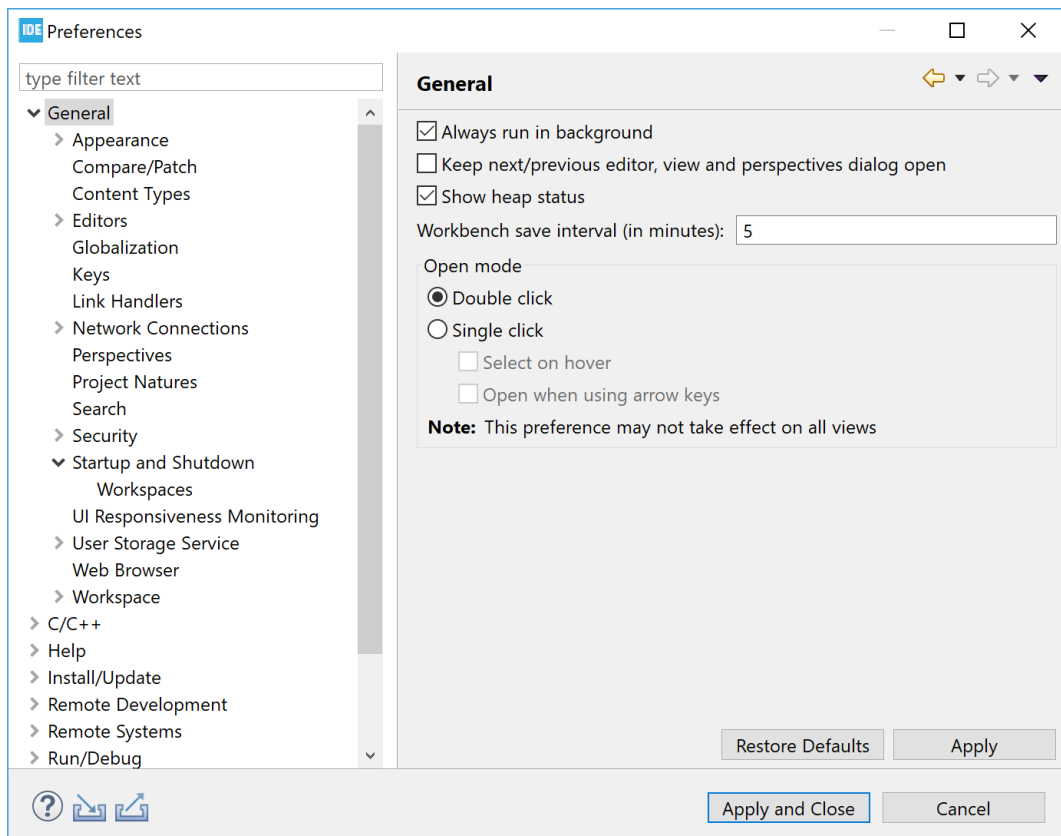
Then select **[File]>[Switch Workspace]** and the new workspace. STM32CubeIDE restarts and opens with the new workspace.

In the menu, select **[File]>[Import]** and in the panel select **[General]>[Preferences]**. Press the **[Next]** button and, on the next page, enable **[Import All]** and enter the file name. The preferences are now the same in both workspaces.

1.7.3 Keeping track of Java heap space

To keep track on how much Java heap space is used, select the **[Window]>[Preferences]** menu. In the *Preferences* page, select the **[General]** node and then enable **[Show heap status]**. The currently used and available Java heap space is then displayed in the STM32CubeIDE status bar. The garbage collector can also be triggered manually from the status bar.

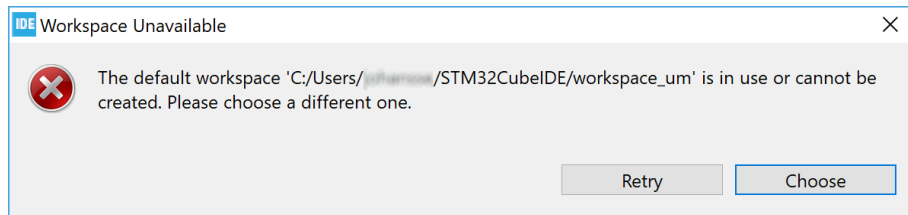
Figure 26. Display of Java heap space status



1.7.4 Unavailable workspace

Only one instance of STM32CubeIDE can access one workspace at a time. This is to prevent conflicting changes in the workspace. If STM32CubeIDE is started with a workspace that is already used by another instance of the program, the following error message is displayed.

Figure 27. Workspace unavailable



If this message is displayed, choose a different workspace, or return to the already running STM32CubeIDE.

1.8 STM32CubeIDE and Eclipse® basics

STM32CubeIDE contains so many features that it is easy to miss some really useful capabilities. Noteworthy features are spell checking of C/C++ comments, word- and code completion, content assist, parameter hints and code templates. The editor also includes an include-file dependency browser, code navigation using hypertext-links, bookmark and to-do lists, and powerful search mechanisms. The next sections remind some of the useful tools that can be easily missed.

1.8.1 Keyboard shortcuts

It is convenient to use keyboard shortcuts instead of the mouse. One important shortcut to know is the shortcut **CTRL+Shift+L**. This shortcut opens a cheat sheet with all available shortcuts.

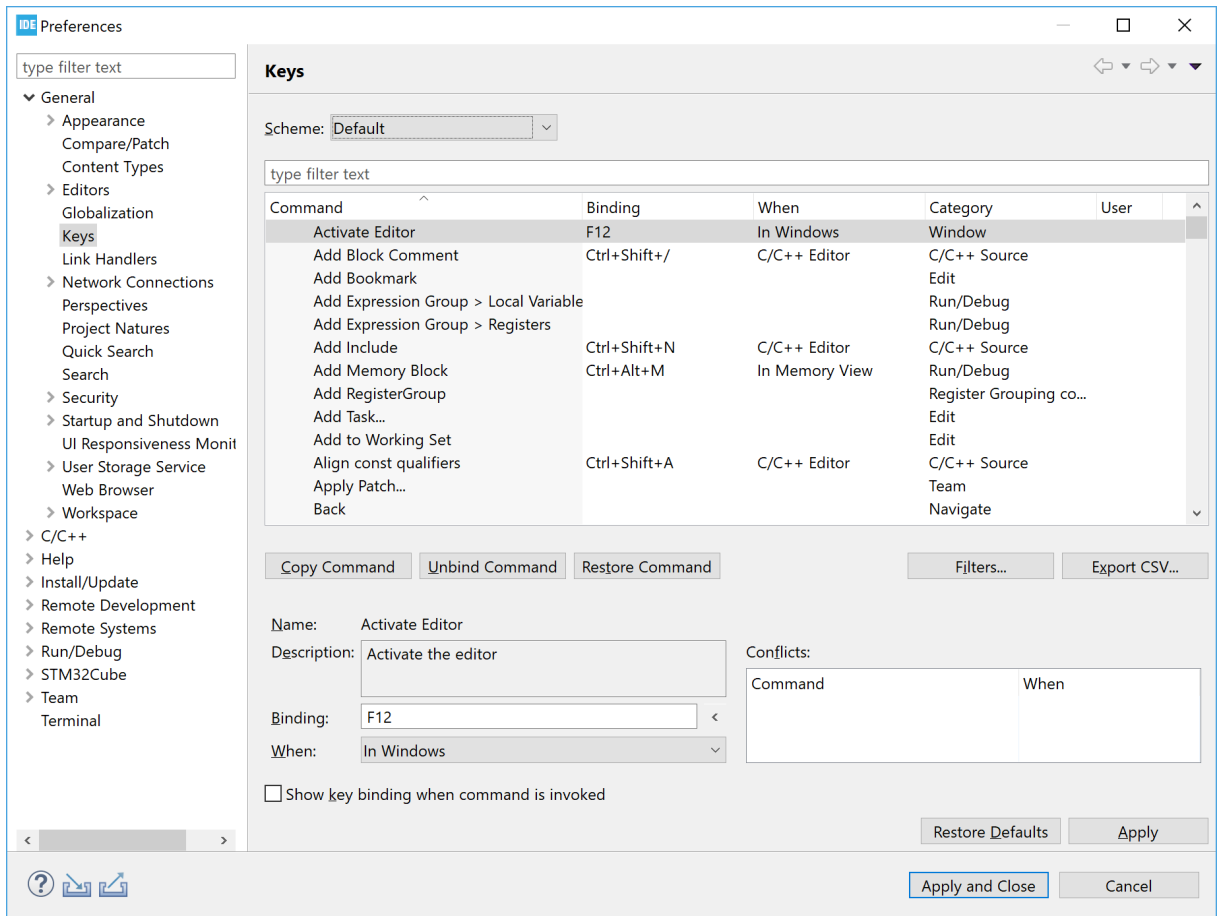
Figure 28. Shortcut keys

Activate Editor	F12
Backward History	Alt+Left
Build All	Ctrl+B
Build Target Build	Shift+F9
Close	Ctrl+F4
Close All	Ctrl+Shift+F4
Collapse All	Ctrl+Shift+Numpad_Divide
Content Assist	Ctrl+Space
Context Information	Ctrl+Shift+Space
Copy	Ctrl+Insert
Cut	Shift+Delete
Debug	F11
Delete	Delete
Expand All	Ctrl+Shift+Numpad_Multiply
Find Text in Workspace	Ctrl+Alt+G
Find and Replace	Ctrl+F
Forward History	Alt+Right
Last Edit Location	Ctrl+Q
Maximize Active View or Editor	Ctrl+M

Press 'Ctrl+Shift+L' to open the preference page

Pressing **CTRL+Shift+L** in this sheet opens the *Keys* pane in the *Preferences* dialog.

Figure 29. Shortcut preferences



The Keys pane offers the possibility to examine the shortcuts in detail and change the scheme (Default, Emacs, or Microsoft® Visual Studio), reconfigure shortcut keys, and others.

Table 2 presents some of the keys to mention with their default bindings.

Table 2. Key shortcut examples

Command	Binding	Where
Copy	Ctrl+C	In dialogs and windows
Cut	Ctrl+X	In dialogs and windows
Paste	Ctrl+V	In dialogs and windows
Debug	F11	In windows
Open declaration	F3	In C/C++ editor
References	Ctrl+Shift+G	In C/C++ editor/views
Find and open files	Ctrl+Shift+R	In C/C++ editor/views
Toggle selection mode normal/block	Alt+Shift+A	In C/C++ editor/views
Zoom In	Ctrl++	Editing text
Zoom Out	Ctrl+-	Editing text

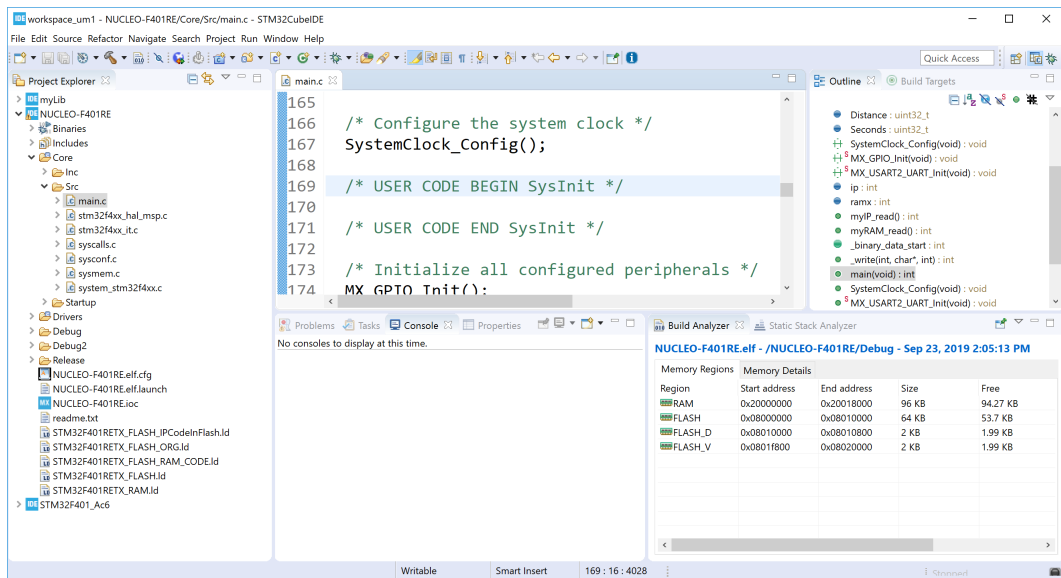
1.8.2 Editor zoom in and zoom out

It is possible to increase or decrease the default font size for text editors by pressing **Ctrl++** and **Ctrl+-**:

- **Ctrl++** : zoom in text
- **Ctrl+-** : zoom out text

Note: If a keyboard with a numeric keypad is used and the **+** or **-** keys are pressed on the numeric keypad, use the **Shift** key in addition to make the zoom work (**Ctrl+Shift++** or **Ctrl+Shift--**).

Figure 30. Editor with text zoomed in



1.8.3 Quickly find and open a file

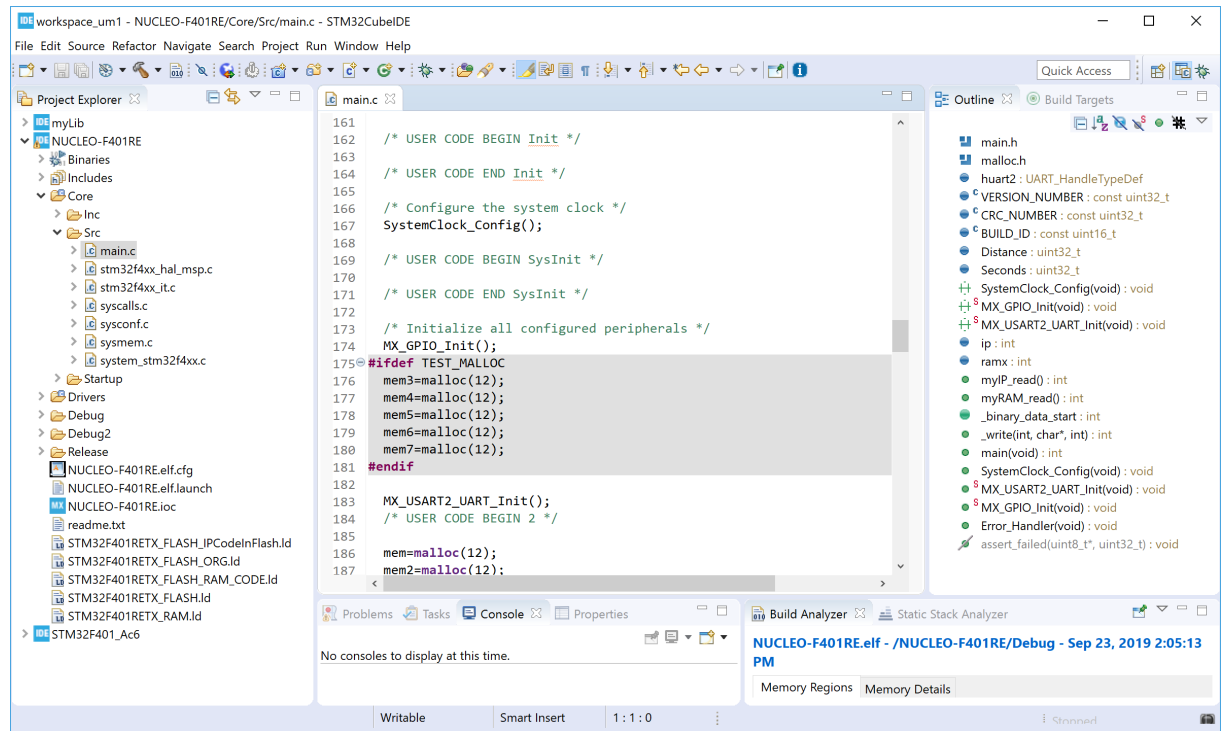
Pressing **Ctrl+Shift+R** to find and open a file quickly is one of the featured easily missed. Type a couple of characters part of the name of the file to open. It is possible to add the ***** and **?** search wildcards as appropriate. The editor then lists the matching filenames. Select the desired file in the search result list, and open the file using any of these three ways:

- **[Show In]**: sends the file to one of the views chosen in the drop-down list (such as the #include file dependency browser view)
- **[Open With]**: opens the file in the editor selected in the drop-down list
- **[Open]**: probably the most commonly used option, simply opens the file in the standard C/C++ editor

1.8.4 Branch folding

A block of code enclosed within `#if` and `#endif` can be folded. To activate the functionality, go to **[Window]>[Preferences]**, then **[C/C++]>[Editor]>[Folding]** and check the **[Enable folding of preprocessor branches (#if/#endif)]** checkbox. Once the checkbox is checked, the editor must be restarted. Close the file, open it again, and the small icon in the left margin of the editor showing that the functionality is activated.

Figure 31. Editor folding



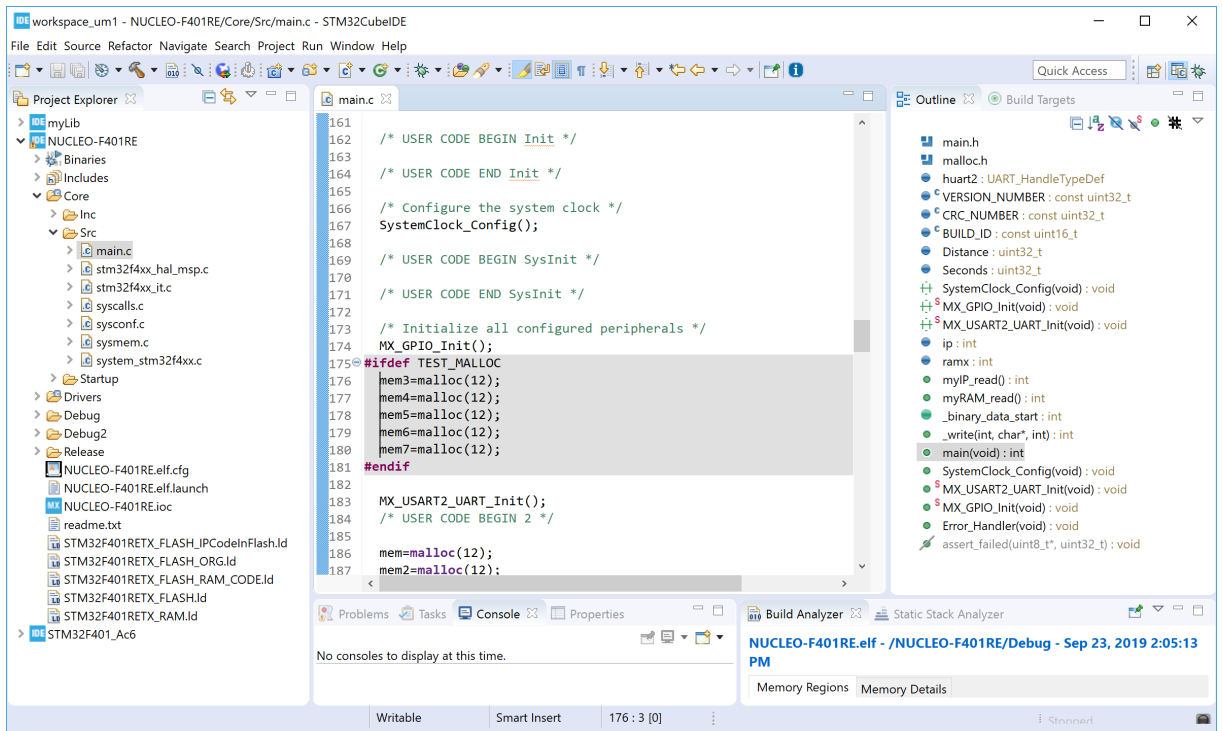
1.8.5 Block selection mode

Alt+Shift+A toggles the selection mode between normal and block. When the block mode is enabled, a block of text can be selected by either the mouse or the keyboard using the **SHIFT+ARROW** buttons.

Use of the block selection mode

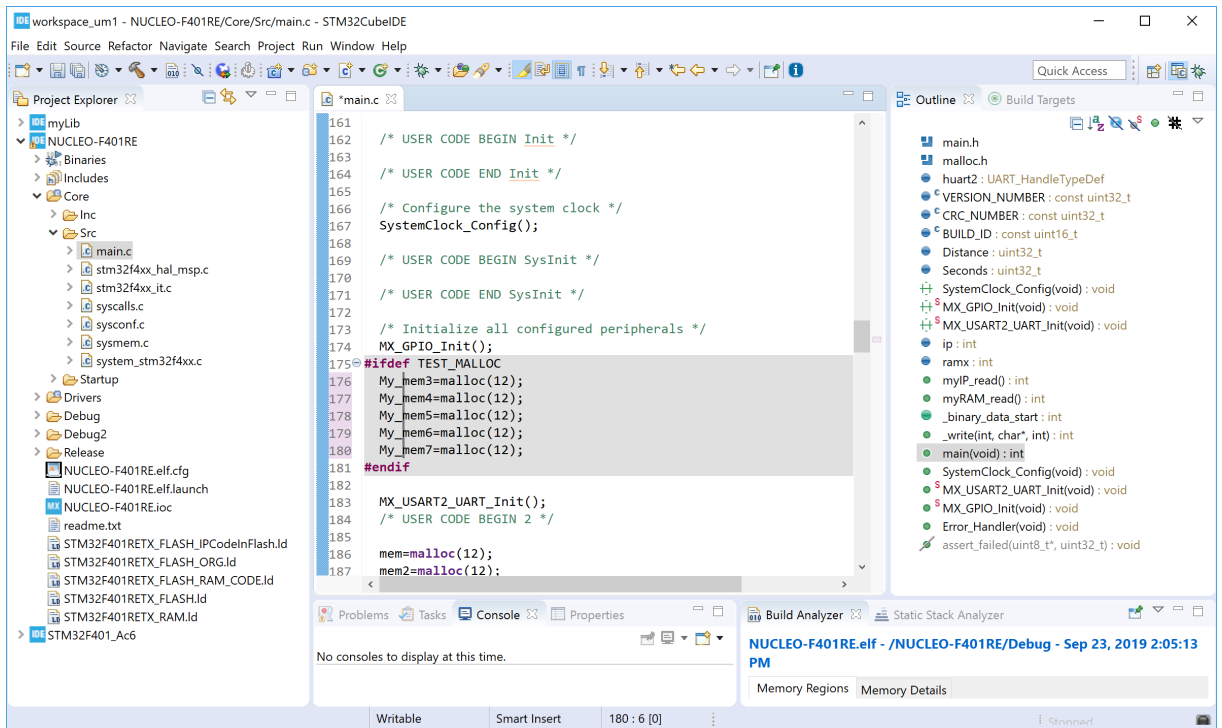
To start using the block selection mode, press **Alt+Shift+A**. Click somewhere in the text and drag down. A column is then marked as shown in Figure 32.

Figure 32. Editor block selection



Add some text and see that this text is entered in all marked rows. As an example, the text “My_” is added and displayed in Figure 33.

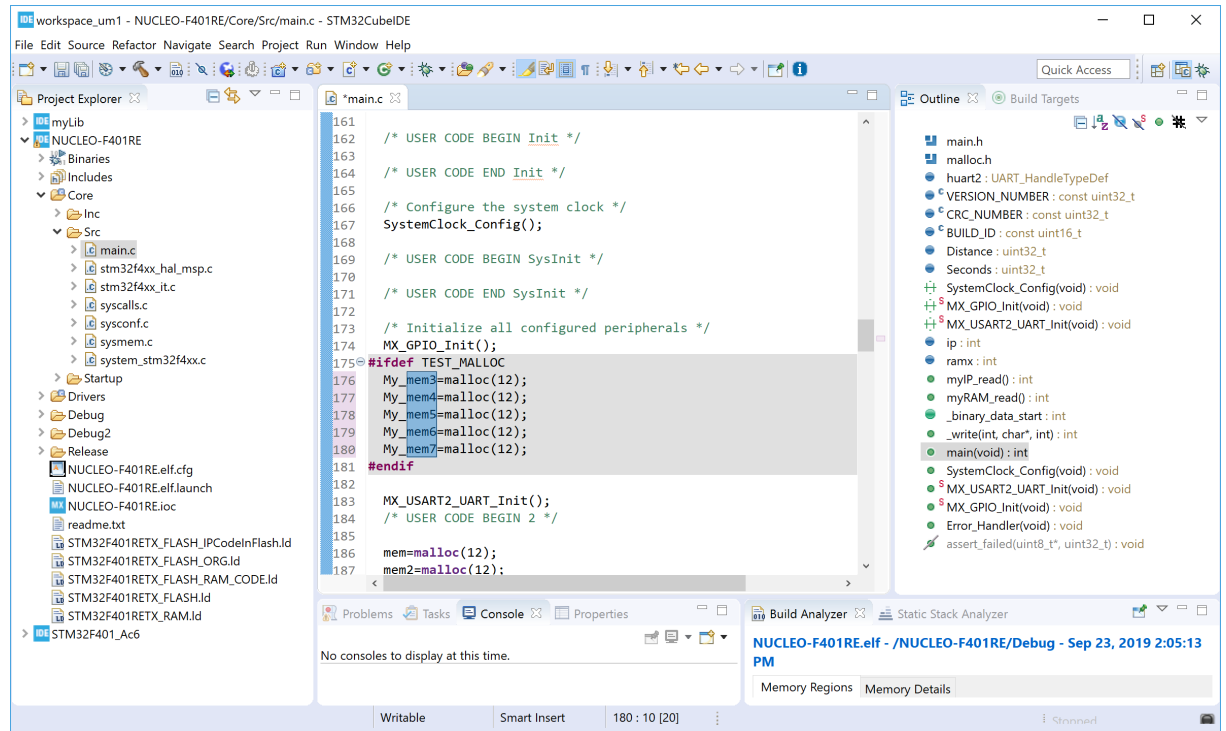
Figure 33. Editor text block addition



Selection and edition of areas

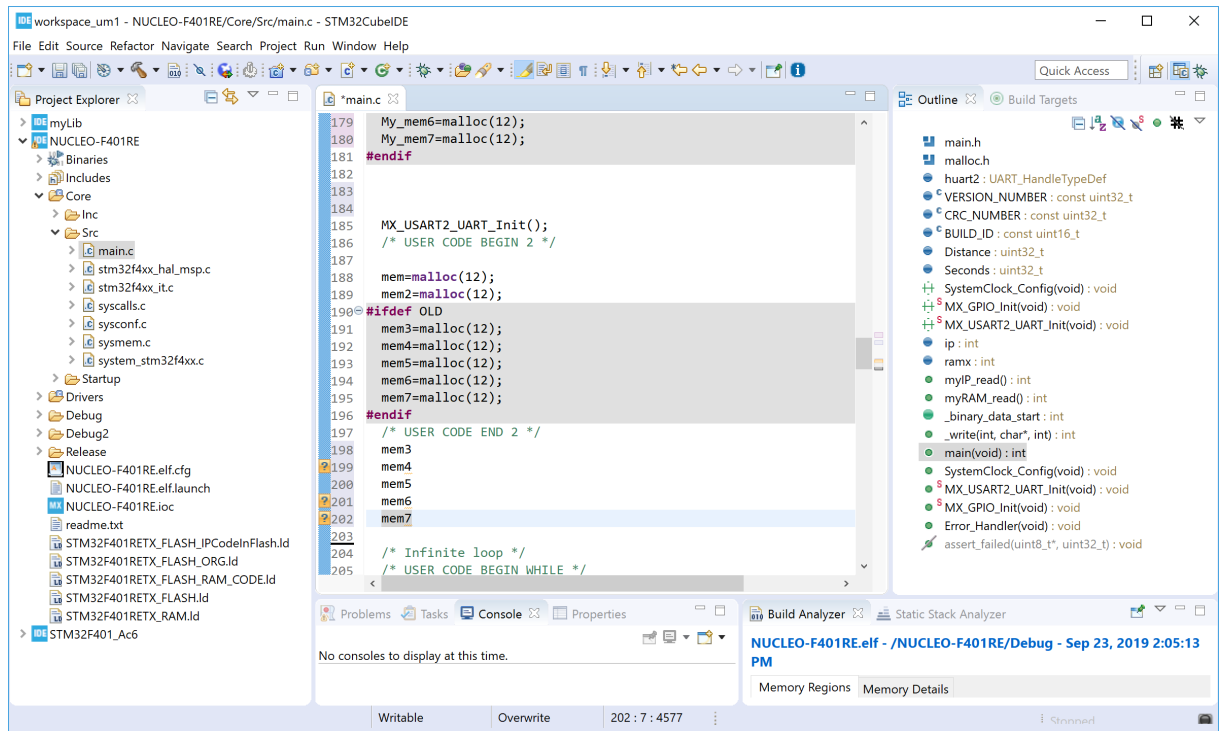
Select a block. In Figure 34, the block starting with “mem3” to “mem7” is selected.

Figure 34. Editor column block selection



Copy the selected block by using **Ctrl+C**. This copied text can then be inserted elsewhere. To do so, type **Alt+Shift+A** to toggle the selection mode back to the normal mode, move the cursor to another line, and type **Ctrl+V** to paste the copied columns to the new lines.

Figure 35. Editor column block paste



1.8.6

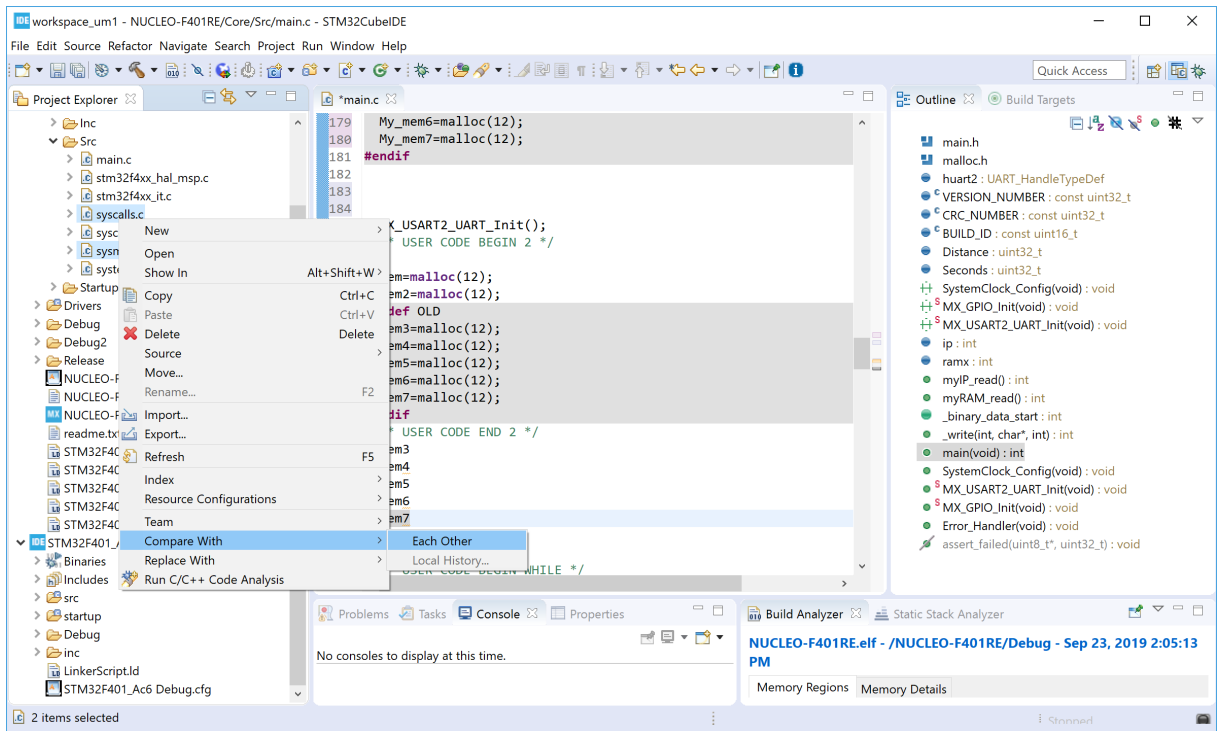
Compare files

To compare two files easily in STM32CubeIDE:

1. Select the two files in the *Project Explorer* view
2. Click on one file
3. Press **CTRL**
4. Click on the other file
Both files are now marked in the *Project Explorer* view
5. Right-click and select **[Compare With]>[Each Other]**

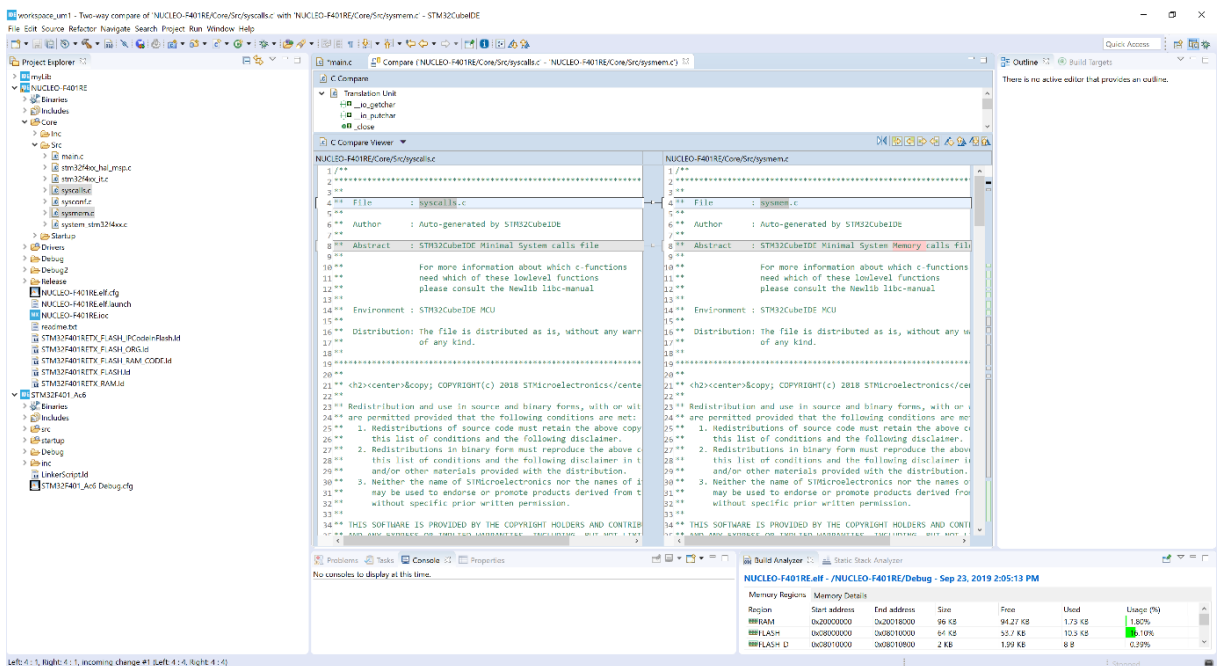
Note: It is possible to configure how the comparison of files is managed. For instance, ignoring white space can be enabled from the preferences. Open the Preferences page using **[Window]>[Preferences]**, select **[General]>[Compare/Patch]**, and enable **[Ignore white space]**.

Figure 36. Editor - Compare files



The *File Differences* editor opens and compares both files.

Figure 37. Editor - File differences

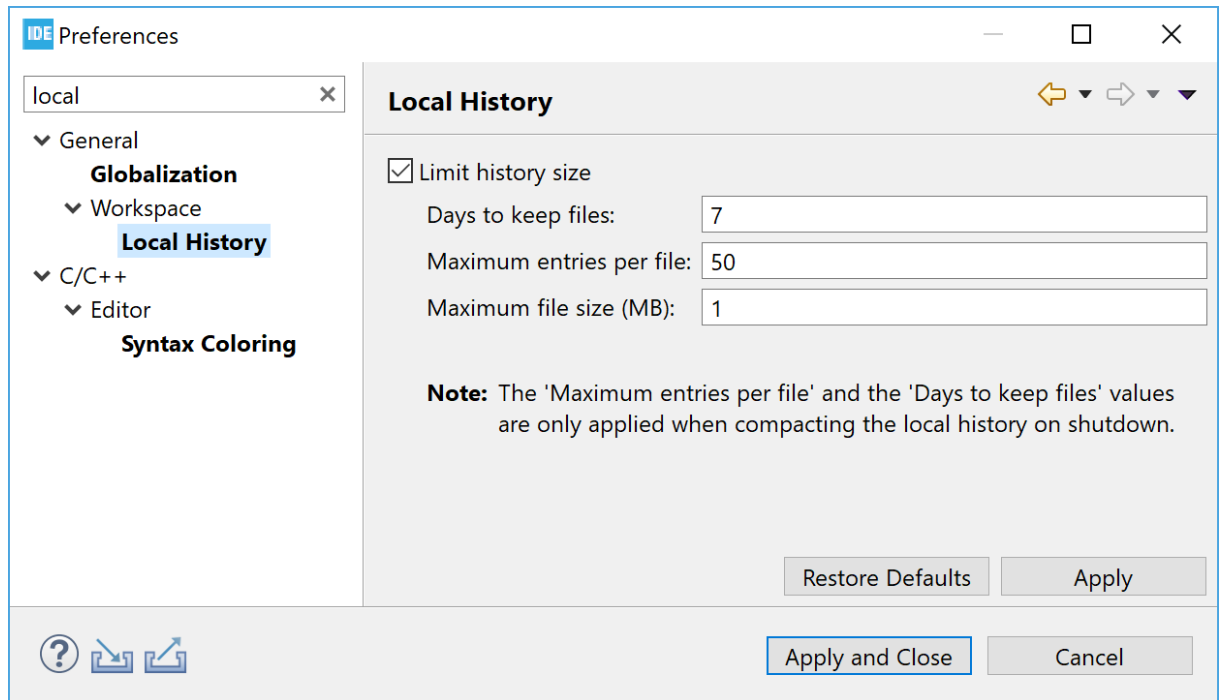


Use the navigation buttons to navigate between differences, or simply navigate in the view using the scroll bar to see the file differences.

1.8.7 Local file history

It is recommended to maintain projects with a version control system such as Apache® Subversion® (SVN) or Git™. Still, STM32CubeIDE contains a local file with the history of edited files, which can be useful if some investigation is needed after a file has become not functional. The workspace preferences contain a *Local History* page.

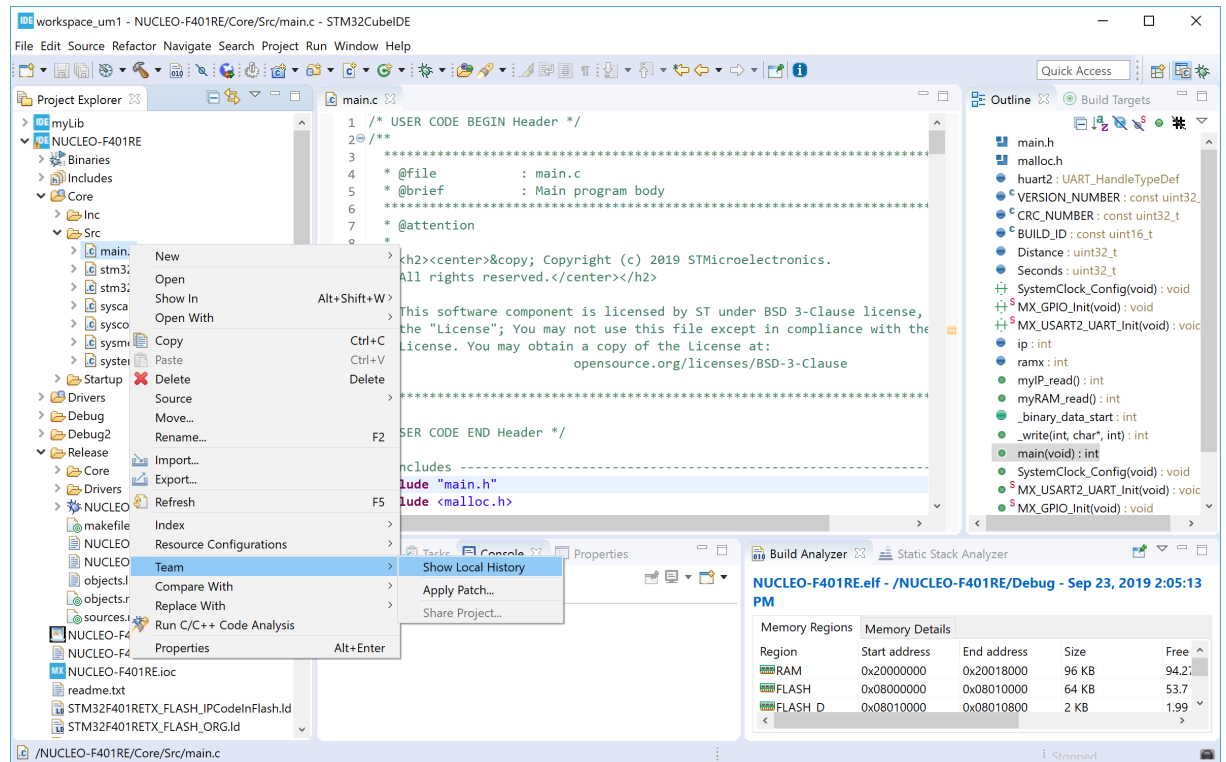
Figure 38. Local history



To show the local history of a file:

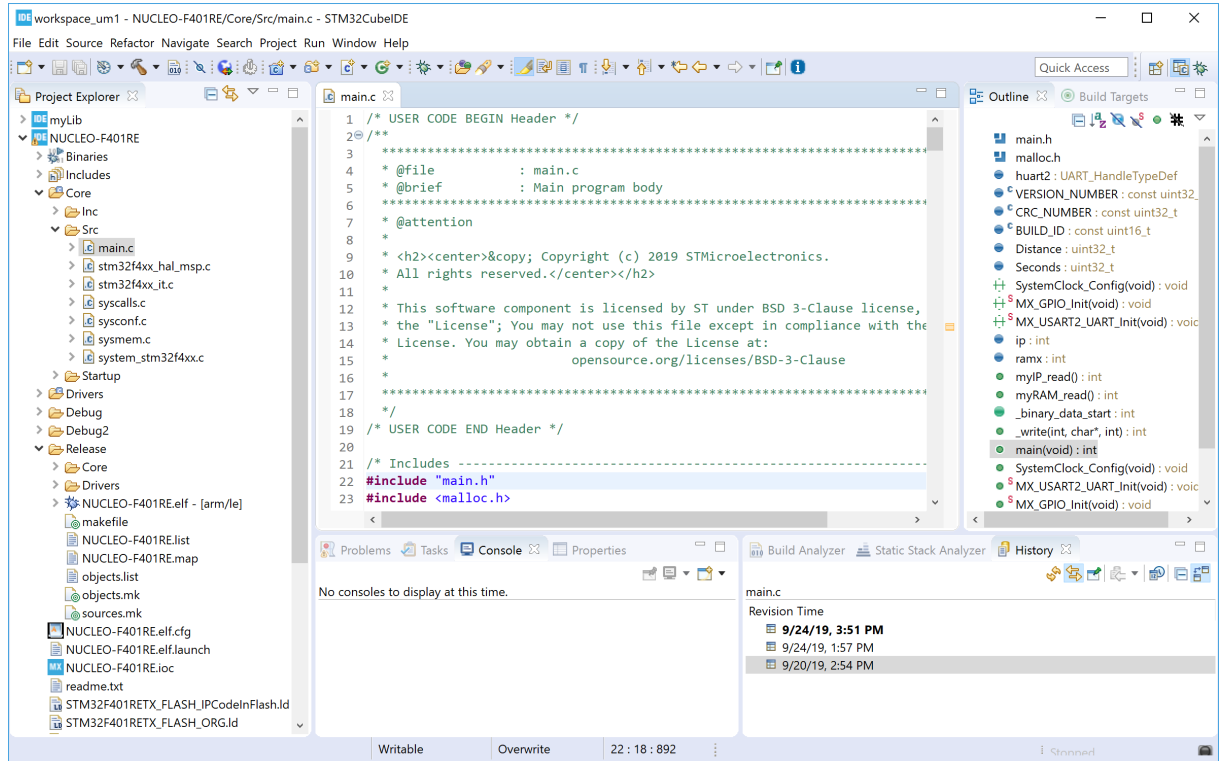
1. Select the file in the *Project Explorer* view
2. Right-click
3. Select **[Team]>[Show local History]**

Figure 39. Show local history



The *History* view opens and displays the file history.

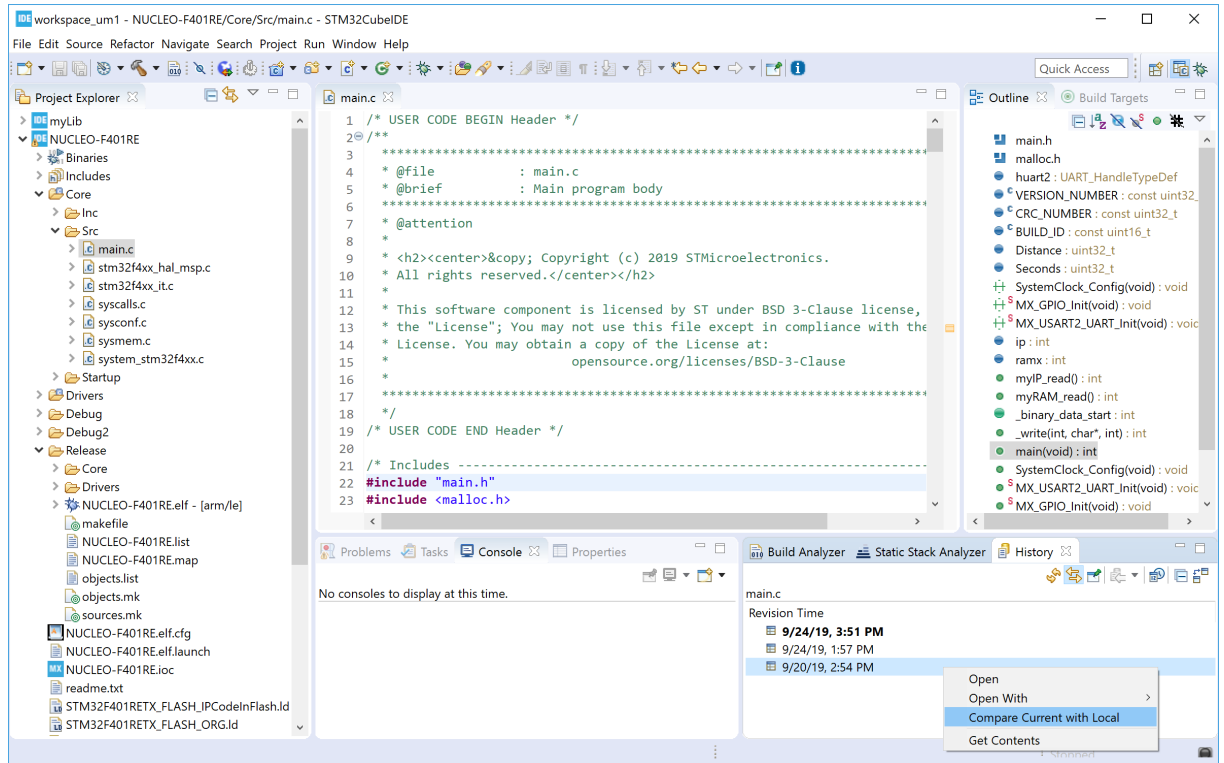
Figure 40. File history



In the case presented in Figure 40, there are three revisions of `main.c`. Double-click on a file in the *History* view to open it in the editor.

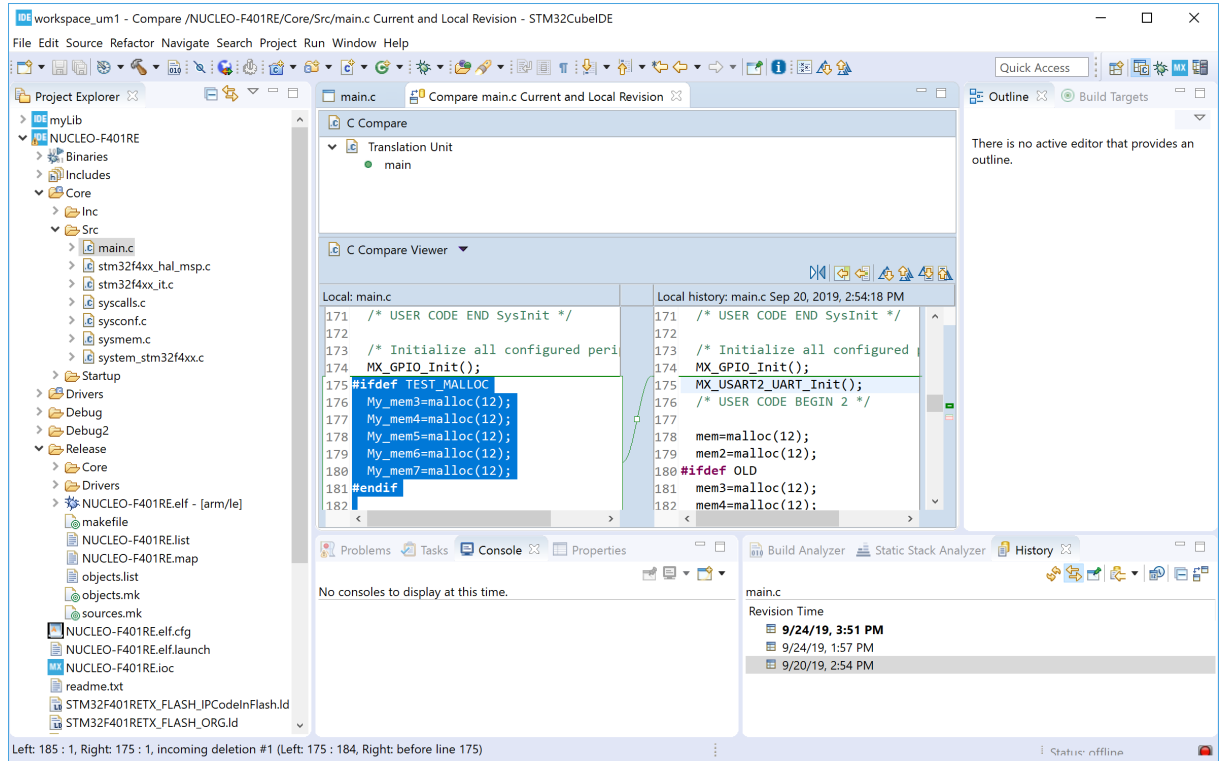
Right-click on a file in the history and select **[Compare Current with Local]** to compare it with the current version of the file.

Figure 41. Compare current history with local history



This opens the *File Differences* editor and displays the file changes.

Figure 42. Compare local file differences



2 Creating and building C/C++ projects

As mentioned in [Section 1.6 Workspaces and projects](#), a workspace is a directory containing projects. The first time a workspace is created, it is empty without any projects. The projects need to be created or imported in the workspace. This section contains information on how to create projects in the workspace and build projects. It also covers how to import and export projects.

2.1 Introduction to projects

A project is a directory in the workspace containing files that may be organized in sub-directories. It is possible to access any project within the active workspace. The files included in a project do not need to be physically located in a folder in the project but can be located somewhere else and linked into the project. Projects located in another workspace cannot be accessed, unless the user switches to that workspace or import some of these projects into the workspace in use.

It is possible to rename and delete a project. If a workspace contains many projects, it is also possible to close some of them to make the work easier. Closed projects can be reopened again at any time.

This section focuses on the two types of STM32 projects supported by STM32CubeIDE:

- Executable programs
- Static library projects

However, the Eclipse® C/C++ Development Toolkit (CDT™), which STM32CubeIDE is based on, contains also basic project wizards, which can be used to create C managed build, C++ managed build, and makefile projects.

The STM32 projects can be:

- C or C++
- Generated executable or library file
- Based on [STM32Cube](#) (using STM32 firmware library package) or empty projects

STM32 projects also support an advanced umbrella project structure, where one project contains many projects, for instance one project per core for multi-core devices.

2.2 Creating a new STM32 project

2.2.1 Creating a new STM32 executable project

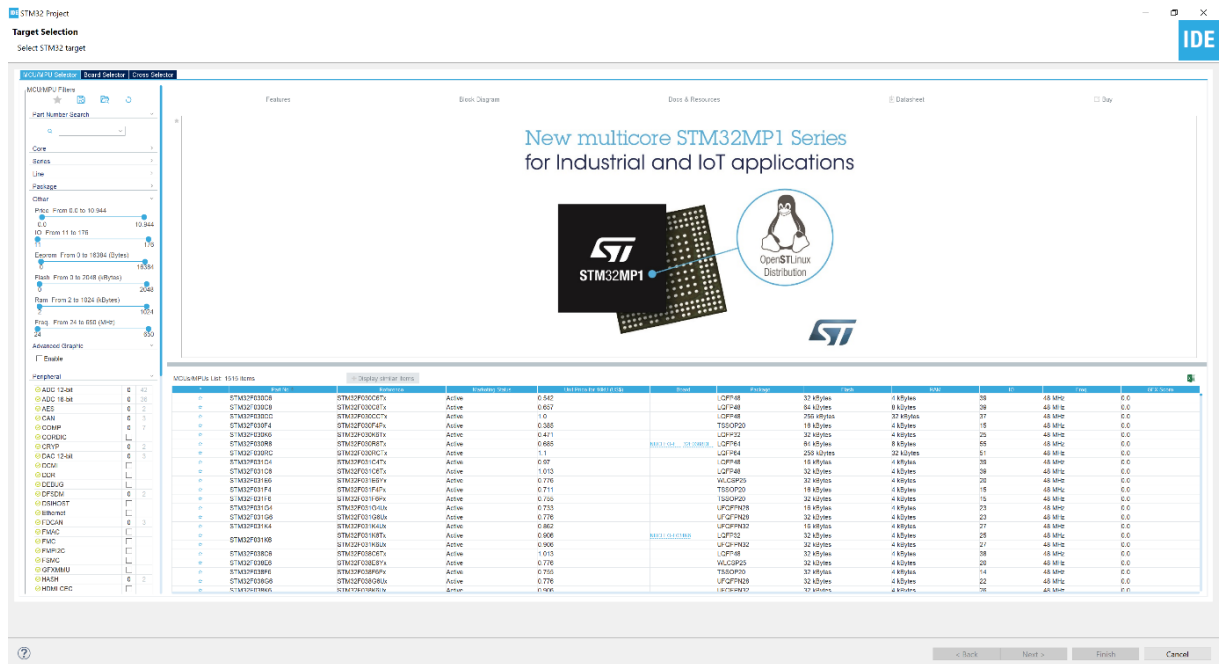
The easiest way to create a new STM32 C/C++ project is to use the STM32 project wizard. It is selected through the menu **[File]>[New STM32 Project]**.

Another way to create a new C/C++ project is to open the *Information Center* and press **[Start new STM32**

project]. As mentioned in [Section 1.3 Information Center](#), the *Information Center* can be opened using the button on the toolbar or via the menu **[Help]>[Information Center]**. 

Both ways initialize and launch the *STM32 Project Target Selection* tool.

Figure 43. STM32 target selection

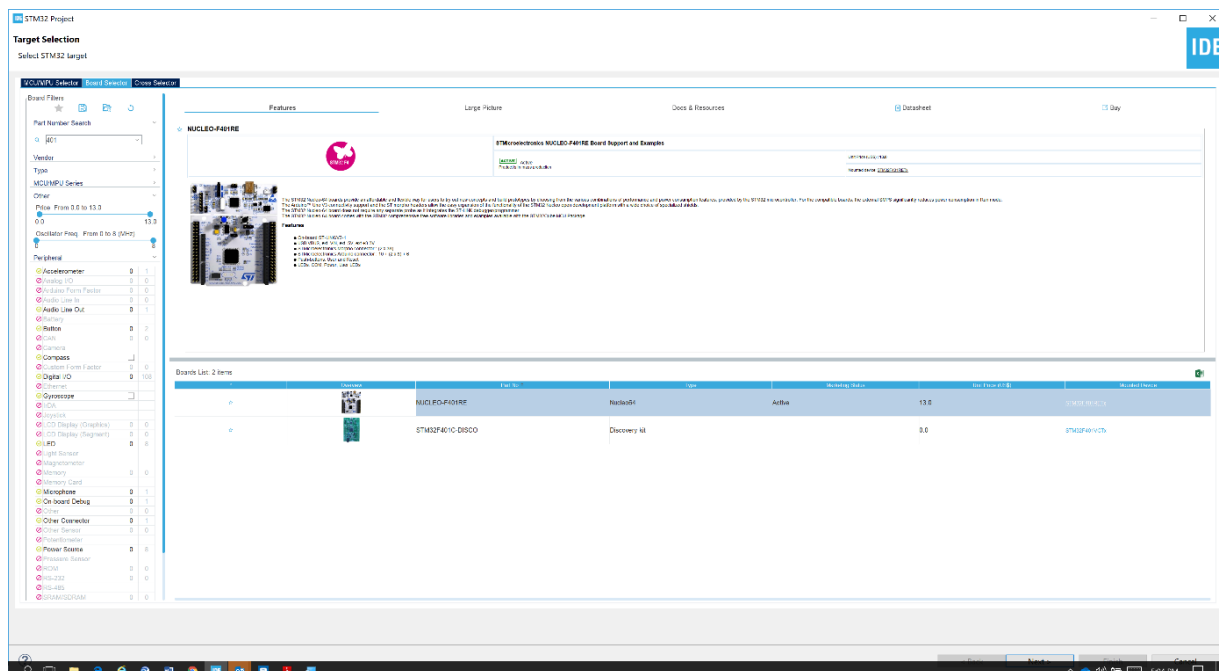


The *MCU/MPU selector* and *Board Selector* tabs can be selected at the top of the window. Use the first tab to create project for a specific device and the second if a project for a specific board is needed.

This section presents the creation of a project for the **NUCLEO-F401RE** board using the *Board Selector*.

Among the different filters available for use on the left of the window, type “401” in the *Part Number Search* field to filter the boards with names containing this string. In **Figure 44**, two boards are listed, a Nucleo board board and a Discovery board. The **NUCLEO-F401RE** board is selected.

Figure 44. STM32 board selection

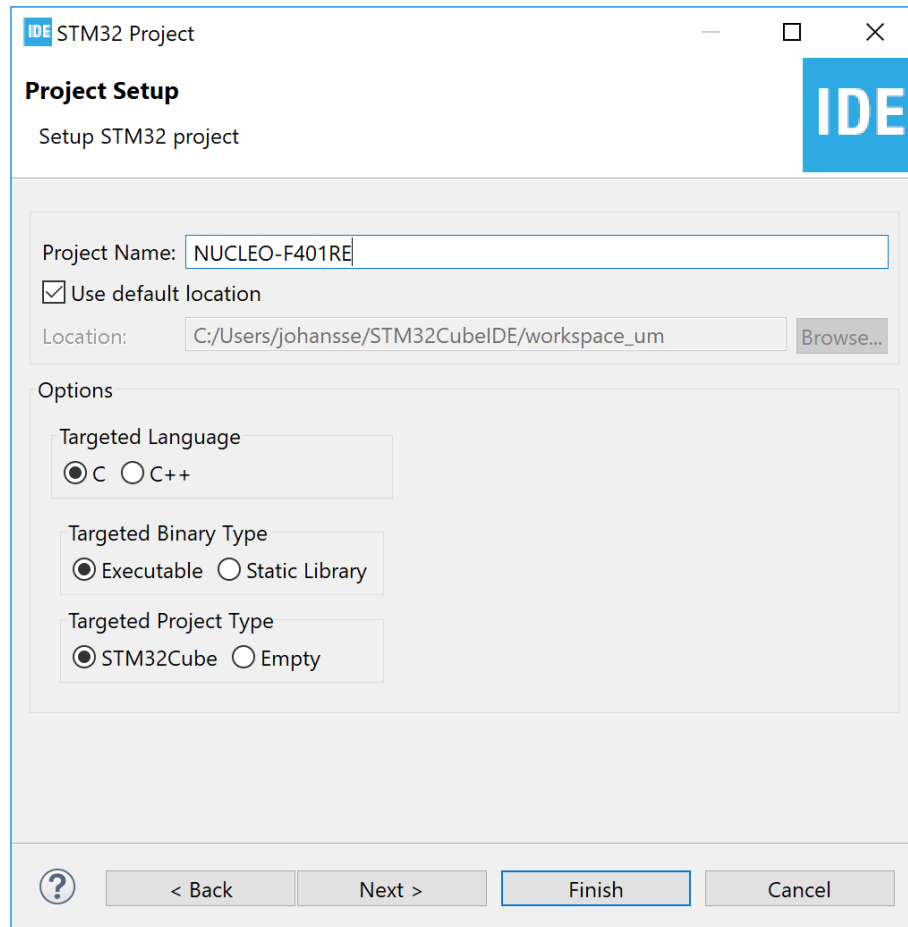


Five tabs, *Features*, *Large Pictures*, *Docs & Resources*, *Datasheet*, and *Buy*, offer the possibility to display detailed information about the selected board or device. For instance, documentation available for the board is displayed and can be opened when *Docs & Resources* is selected. When *Datasheet* is selected, the board datasheet is downloaded from STMicroelectronics web site.

Pressing **[Next]** when the NUCLEO-F401RE board is selected opens the *Project setup* page.

Enter a project name and select the desired setting for the project in the dialog boxes. The project named "NUCLEO-F401RE" is filled in as an example in [Figure 45](#).

Figure 45. Project setup

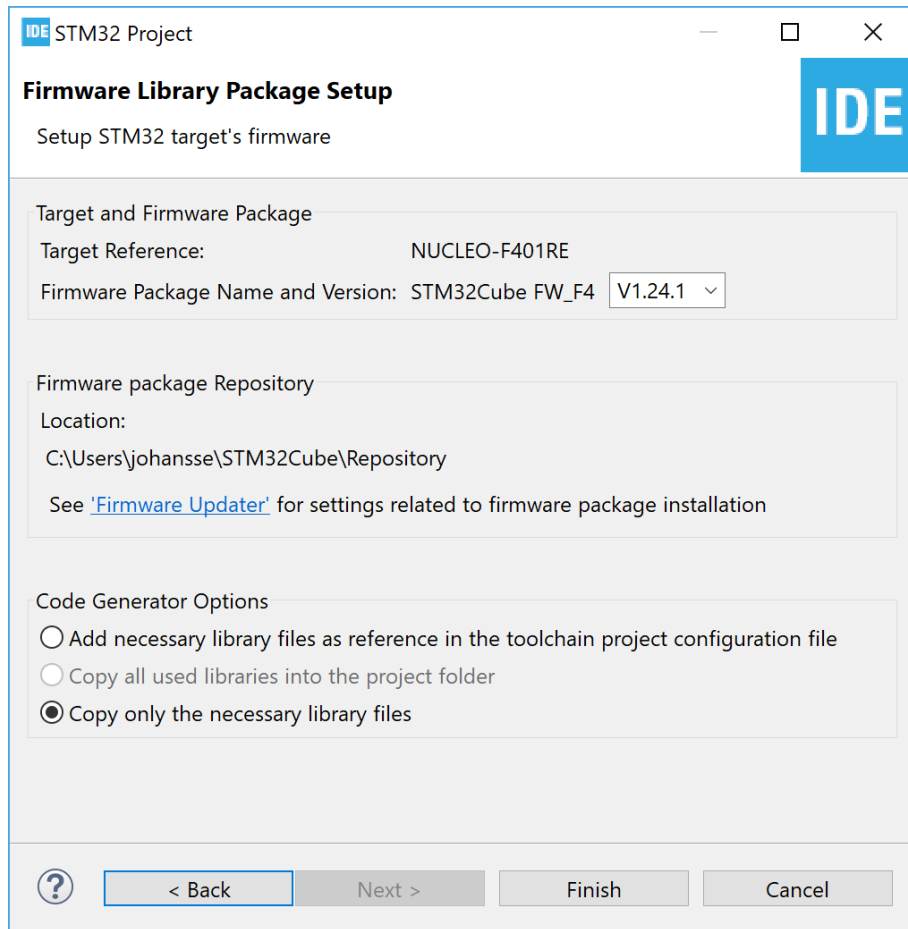


According to the settings in [Figure 45](#), the project is meant to be stored in the default location with the following options set:

- C project
- Executable binary type
- STM32Cube targeted project type

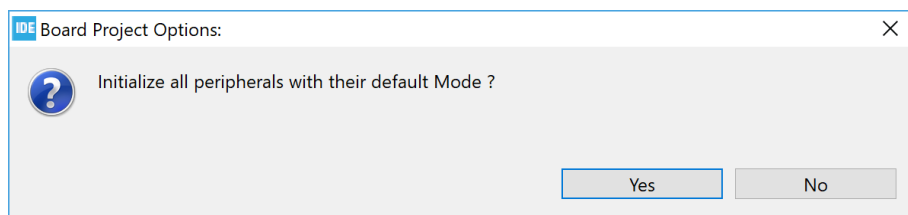
Press **[Next]** to open the *Firmware Library Package Setup* page.

Figure 46. Firmware library package setup



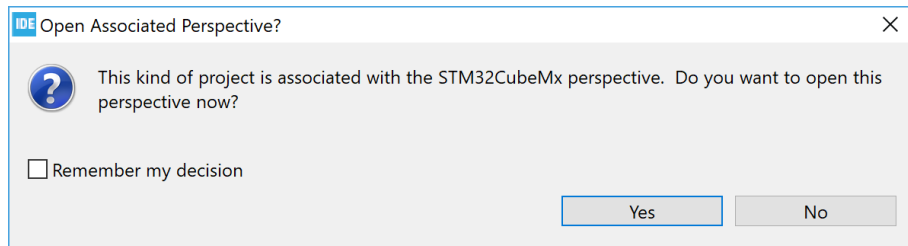
In this page, it is possible to select the STM32Cube firmware package to use when creating the project. In this case, the default settings are used. Press **[Finish]** to create the project. As a result, the following dialog is displayed.

Figure 47. Initialization of all peripherals



Press **[Yes]** since it is a good practice to get the software needed to initialize the peripherals. This opens the new dialog shown in Figure 48.

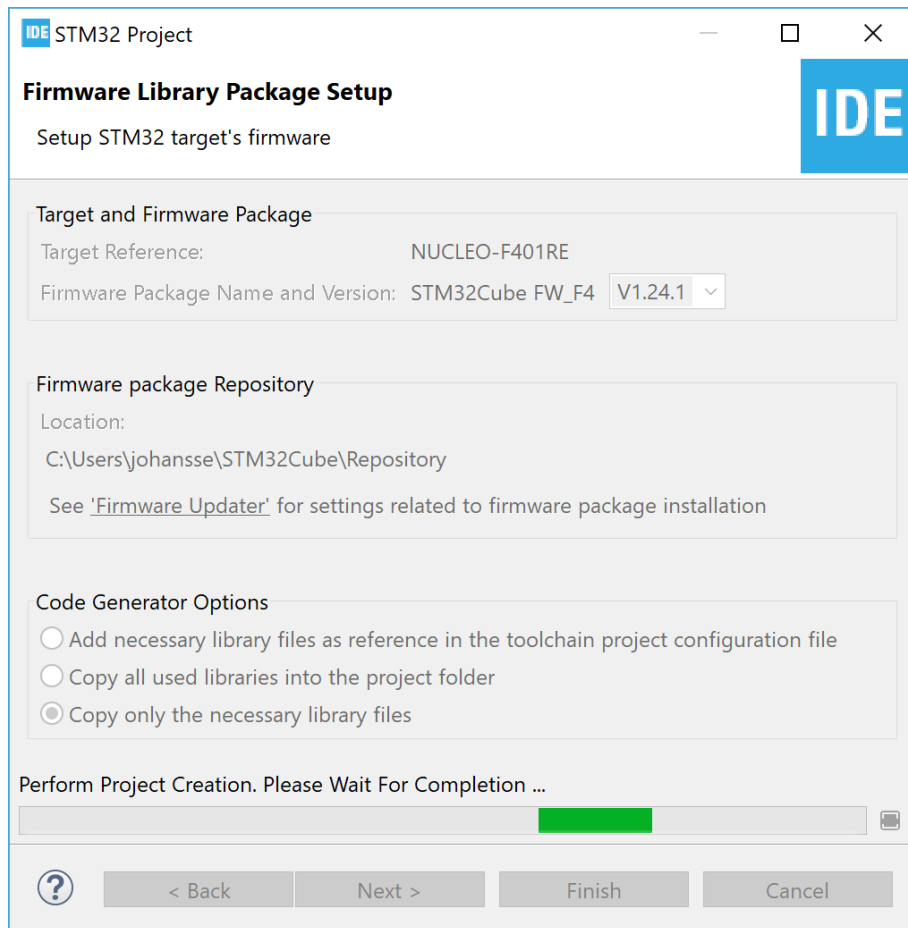
Figure 48. STM32CubeMX perspective opening



Opening the *STM32CubeMX* perspective is a good decision if there are any needs to configure the device. Enable **[Remember my decision]** if the question must not be asked the next time a new project is created. Press **[Yes]** to continue.

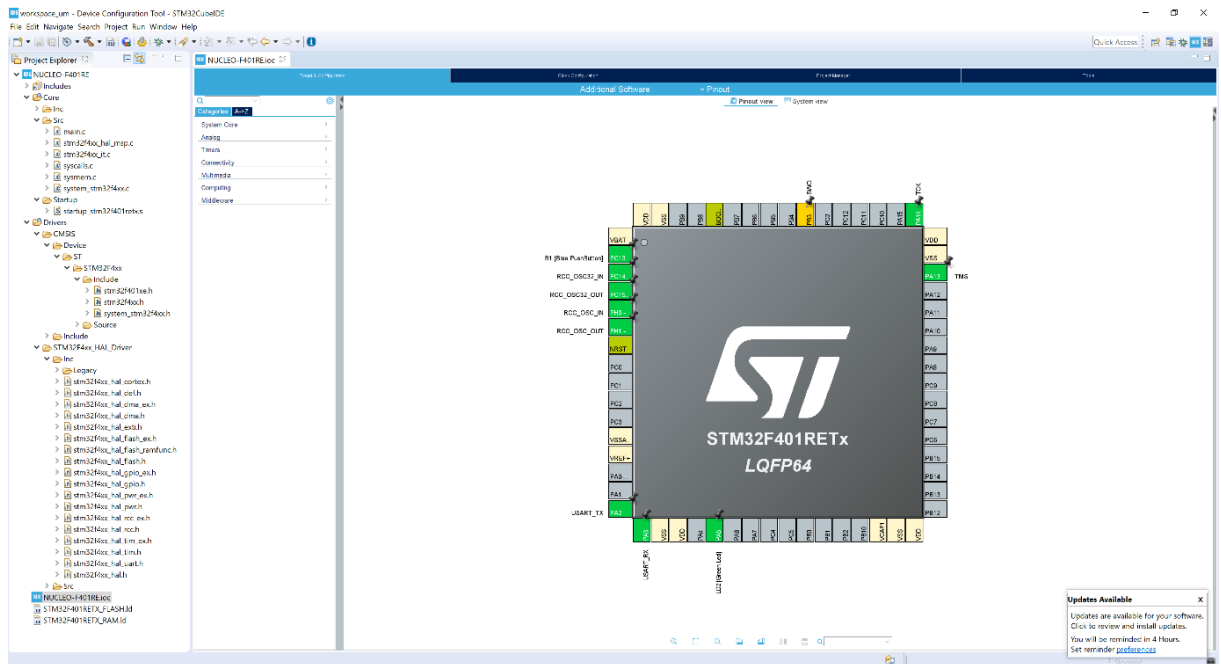
At this point, the project creation starts. The time it takes depends on the amount of files that need to be downloaded to create the project.

Figure 49. Project creation started



When the project is created, the *STM32CubeMX* perspective is opened with a window for configuring the peripherals, clock, middleware, and power consumption.

Figure 50. STM32CubeMX



The new project is listed in the *Project Explorer* view with some of the folders and files it contains.

The `NUCLEO-F401RE.ioc` file contains the configuration settings and is opened in the STM32CubeMX editor. This editor contains tabs for *Pinout & configuration*, *Clock configuration*, *Project manager* and *Tools*. When changes are made in the STM32CubeMX editor, the `.ioc` file in the tab is marked as changed. If the file is saved, a dialog opens asking “Do you want to generate Code?”, making it easy to generate new code in the project that supports the new device configuration. For more information on how to use the STM32CubeMX editor, refer to [ST-14].

It is possible to create an STM32 project with less files and folders by selecting the targeted project type **[Empty]** instead of **[STM32Cube]** (refer to Figure 45. Project setup). When **[Empty]** is selected, the generated project only contains some folders, a device startup file with `Reset_Handler` code and vector table, the `main.c` file, and some other c files and linker script files. STM32 header files, system files and CMSIS files must be added manually. These files can for instance be copied from some other STM32Cube targeted project or from an STM32 example project.

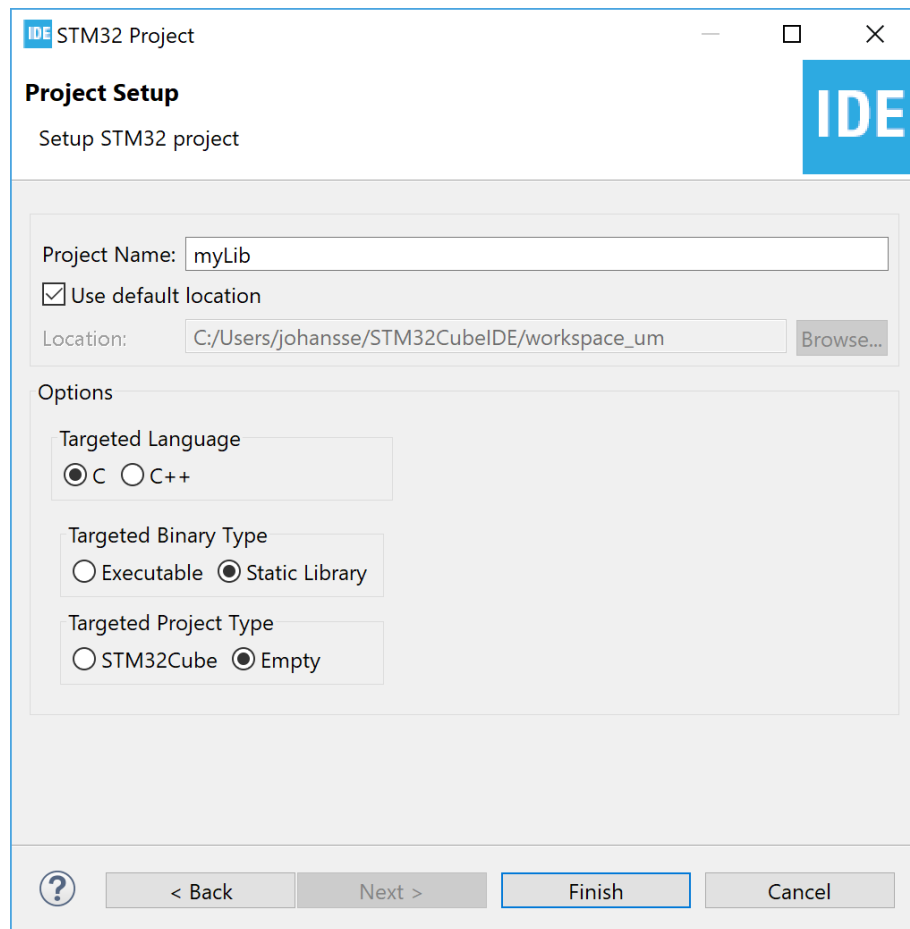
Note: *For empty projects, make sure to configure the floating-point unit setting to use software FPU or hardware FPU according to application requirements. When using hardware FPU, initialize the FPU. For non-empty projects, the initialization of the FPU is normally done in the `SystemInit` function in file `system_stm32fxxx.c`. To notify that the FPU configuration may be needed, the `main.c` file created in an empty project contains a compiler warning stating `#warning "FPU is not initialized, but the project is compiling for an FPU. Please initialize the FPU before use."`*

2.2.2 Creating a new STM32 static library project

The process to create an STM32 static library project is similar to the process presented in Section 2.2.1 *Creating a new STM32 executable project*. Open the STM32 project wizard through **[File]>[New STM32 Project]** and select the device or board to create a library for. The selection lets the project wizard know how to setup the compiler options appropriately for the library. When the dialog shown in Figure 51 appears, enter a project name, such as “*myLib*” in this example, and select **[Static Library]** to create a library project.

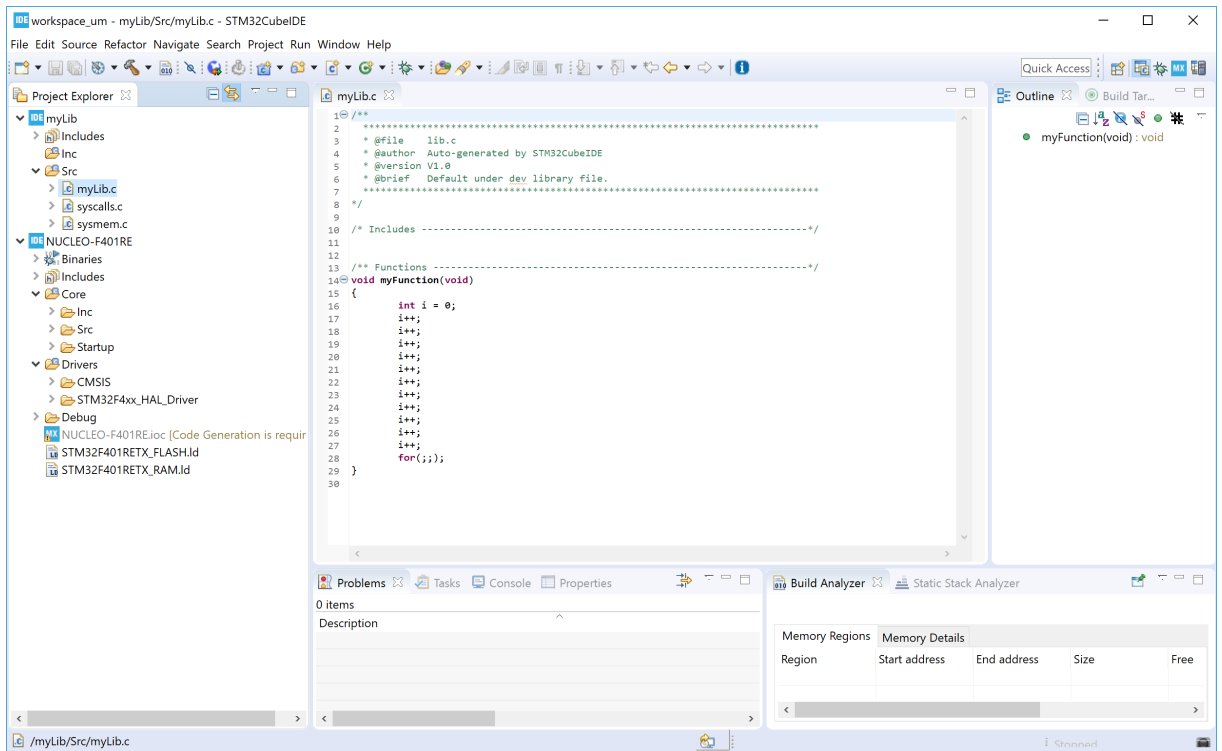
Note: *When creating an STM32 static library project, it is not important to select the correct device exactly, but the selected device must have the same Cortex® core as the device the library is intended for, for instance Cortex®-M0+ or Cortex®-M7.*

Figure 51. STM32 static library project



In Figure 51, the *Targeted Project Type* is selected to be empty. If project type **STM32Cube** is selected instead, the generated project contains STM32 drivers also. Press **Finish** to create the project.

Once the project is created, the *Project Explorer* contains the library project and the previously created NUCLEO-F401RE board project.

Figure 52. STM32 library and board project


As shown in the *Project Explorer* view in Figure 52, the `myLib` project only contains an `Inc` and `Src` folder. Folder `Inc` is empty by default. It is intended that header files are added to this folder. The `Src` folder contains three C files. The `<myLib.c>` file is intended to be updated with the library functions that must be included in the library. The name of the library file is the same as the project name. The two other C files are used by Newlib if needed and can be updated if required.

The name of the function in file `myLib.c` can be renamed. More functions can be added into the `myLib.c` file. Additional C files can also be added if a bigger library is created. It is good practice to create header files containing prototypes of the library functions that are callable from the applications.

Note: *The library project folder does not contain any linker script. When building the library project, `gcc` is used as compiler for the C files and `ar` is used to create an archive file, which can be linked into and used by other executable projects. It is not important to select the correct device for memory settings. The device name is used to set the Cortex® core to the one intended to be used with the library. This makes the compiler generate correct instructions.*

2.3 Configure the project build setting

When an STM32 project is created, it contains default C/C++ build settings for the project. There are however a lot of different options that can be used by GCC, each embedded system having its own requirements. It is therefore possible to configure the project build settings further than the default build settings.

It is also common to have different requirements on build settings during different phases of the project development; for instance during the debugging and release phases. To handle this, different build configurations for each project are supported by STM32CubeIDE. This section presents the build configurations first, and then the project build settings.

2.3.1 Project build configuration

Each build configuration allows different variants of a project and contains a specific build setting. When an STM32 project is created in STM32CubeIDE, two build configurations, *Debug* and *Release*, are created by default. The *Debug* configuration makes the project built with debug information and without any optimization. The *Release* configuration makes the project optimized for smaller code size and with no debug information. By default, the *Debug* configuration is set as the active build configuration when the project is created.

It is possible to create new build configurations for a project at any time. Such new build configuration can be based on an earlier available build configuration.

When building the project, the active build configuration is used and during build the files generated are written into a folder with the same name as the active build configuration.

Note: The build configuration only handles the build settings. How to configure debug settings is described later in this manual.

2.3.1.1 **Change the active build configuration**

To change the active build configuration:

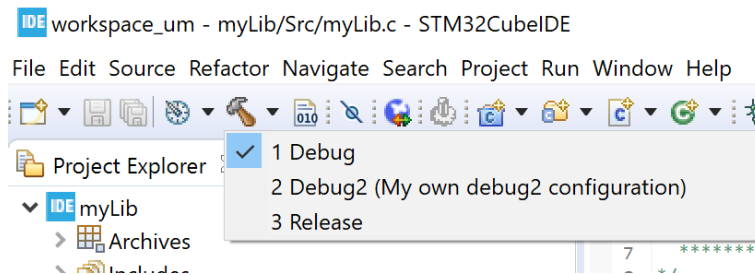
1. Select the project name in the *Project Explorer*
2. Use the toolbar in the C/C++ perspective and click on the arrow to the right of the **[Build]** toolbar button



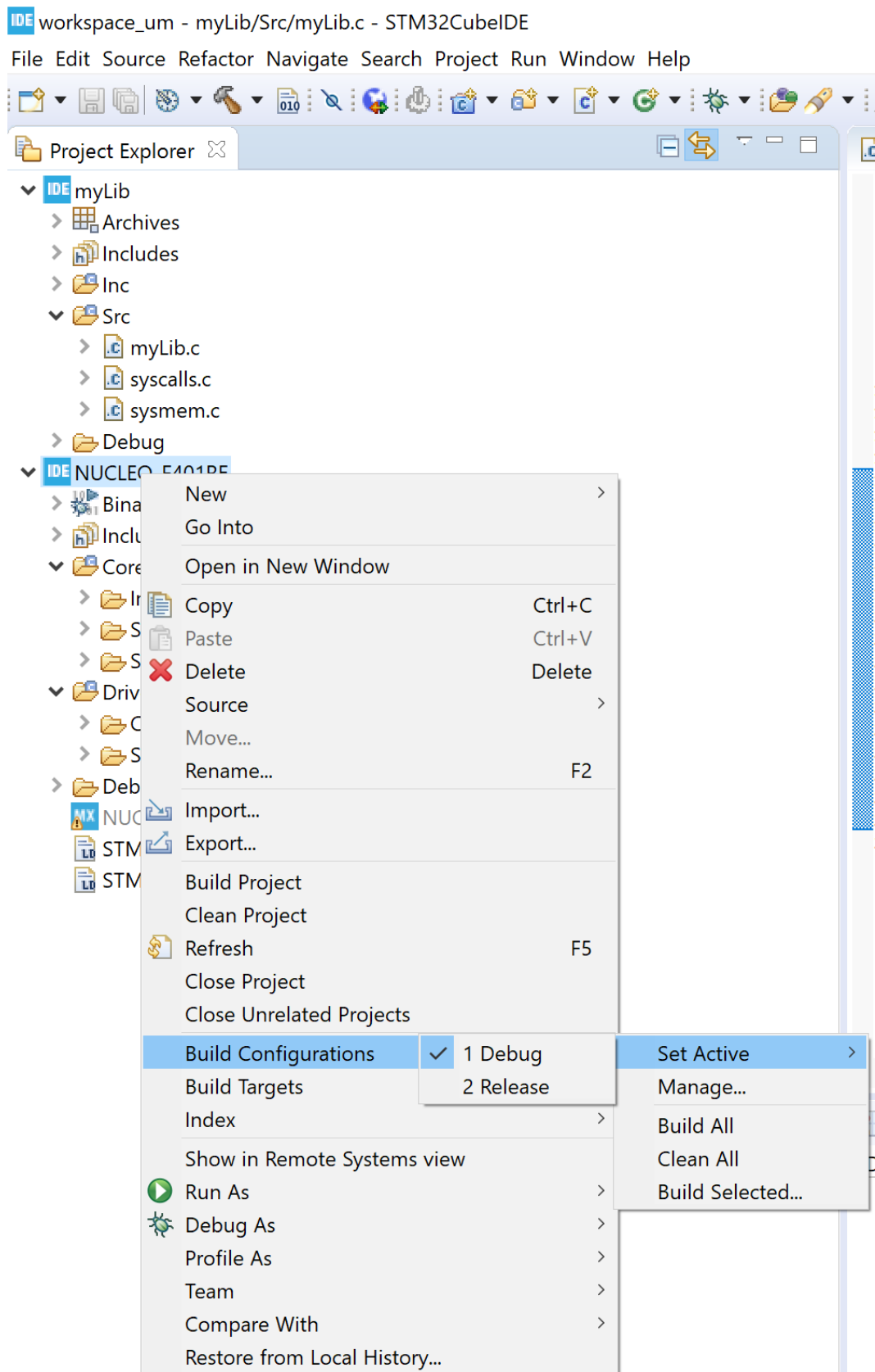
3. The build configurations are listed

Select the build configuration to use from the list.

Figure 53. Set the active build configuration using the toolbar

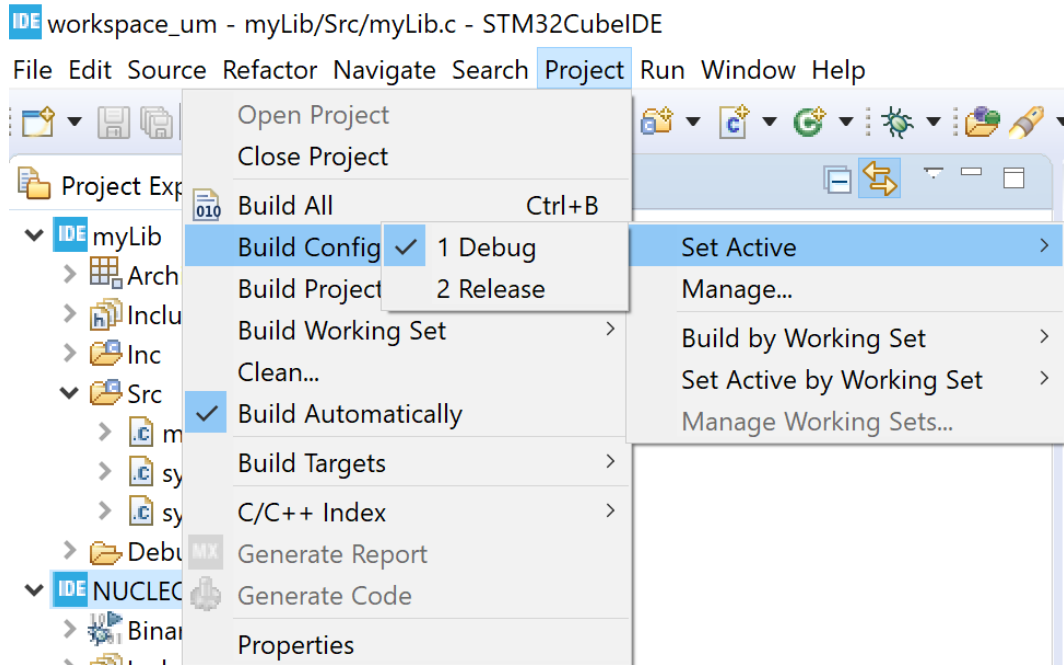


Another way to change the active build configuration is to right-click on the project name in the *Project Explorer* view, select **[Build Configurations]>[Set Active]**, and select the preferred build configuration.

Figure 54. Set active build configuration using right-click


It is also possible to select the active build configurations using the menu [Project]>[Build Configurations]>[Set Active] and select the chosen build configuration.

Figure 55. Set active build configuration using menu



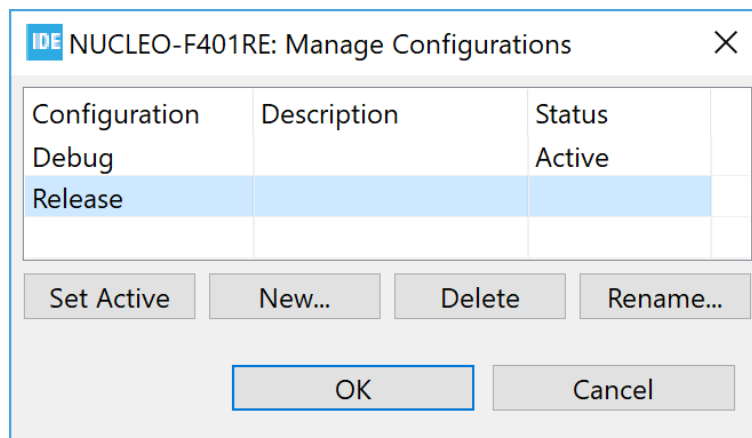
2.3.1.2 Create a new build configuration

To create a new build configuration:

1. Right-click on the project name in the *Project Explorer* view
2. Either:
 - Select [Build Configurations]>[Manage...]
 - Use the menu [Project]>[Build Configurations]>[Manage...]

Both methods open the *Manage Configurations* dialog.

Figure 56. Manage Configurations dialog

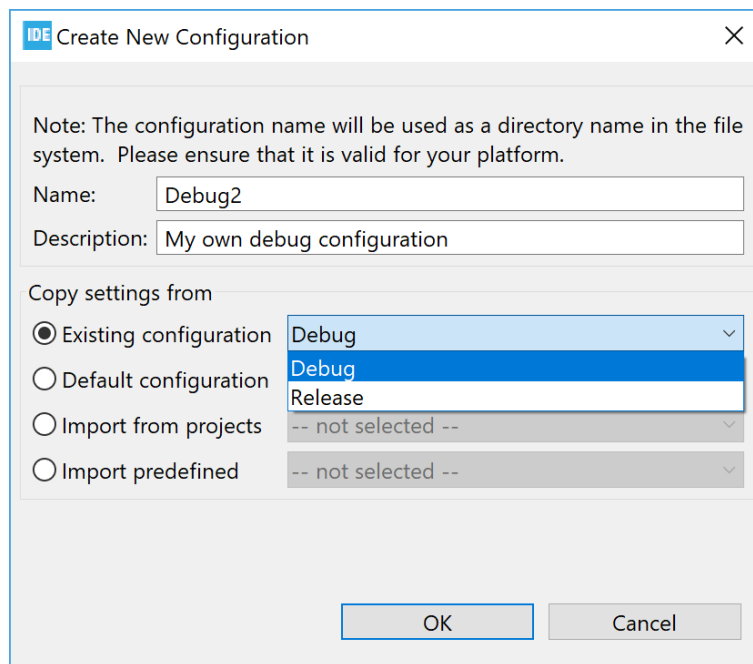


As shown in Figure 56, some buttons in the dialog are used to manage the configurations:

- **[Set Active]** is used to change and select another configuration to be active
- **[New...]** is used to create a new build configuration
- **[Delete]** is used to delete an existing build configuration
- **[Rename...]** is used to rename the build configuration

To create a new build configuration, press the **[New...]** button. This opens the *Create New Configuration* dialog. In this dialog, a name and description is entered. The name must be a valid directory name since it is used as the directory name when building the project with the new configuration.

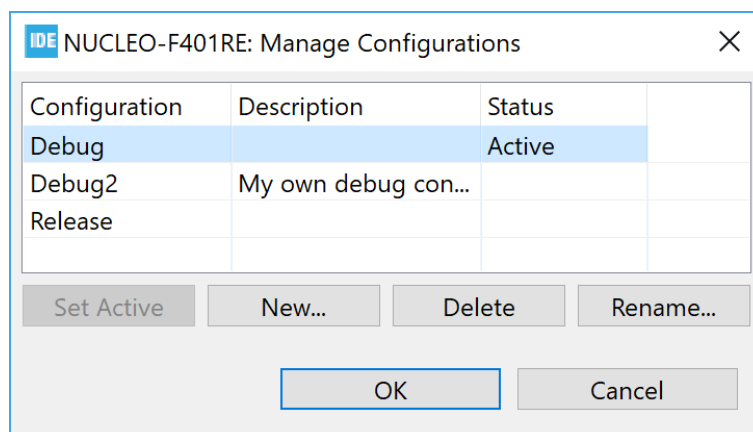
Figure 57. Create a new build configuration



As seen in Figure 57, the new build configuration is based on an existing build configuration. In the case illustrated, the new configuration is based on the existing *Debug* configuration. Press **[OK]** when finished with the settings.

The *Manage Configurations* dialog opens and the new debug configuration is displayed.

Figure 58. Updated Manage Configurations dialog



Change the active configuration to another configuration if needed and press **[OK]** to save and close the configurations dialog when finished managing configurations.

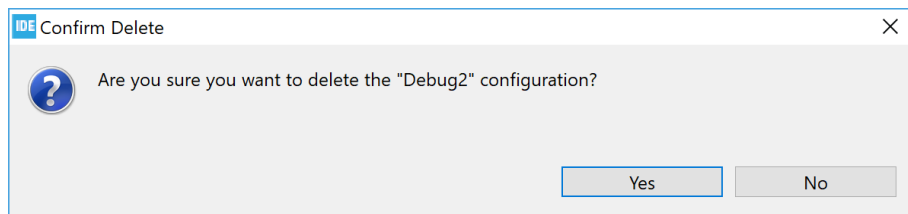
2.3.1.3 Delete a build configuration

To delete a build configuration:

1. Open the *Manage Configurations* dialog
2. Select the configuration to be deleted
3. Press the **[Delete]** button

For instance, if the *Debug2* configuration is selected and **[Delete]** button is pressed, the following confirmation dialog opens.

Figure 59. Configuration deletion dialog



In this case, select **[No]** to keep the *Debug2* configuration.

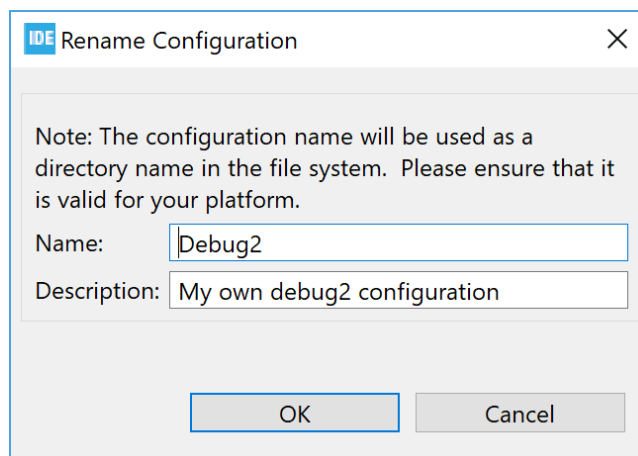
2.3.1.4 Rename a build configuration

To rename a build configuration:

1. Open the *Manage Configurations* dialog
2. Select the configuration to be renamed
3. Press the **[Rename...]** button

For instance, if the *Debug2* configuration is selected and **[Rename...]** button is pressed, the following confirmation dialog opens.

Figure 60. Configuration renaming dialog



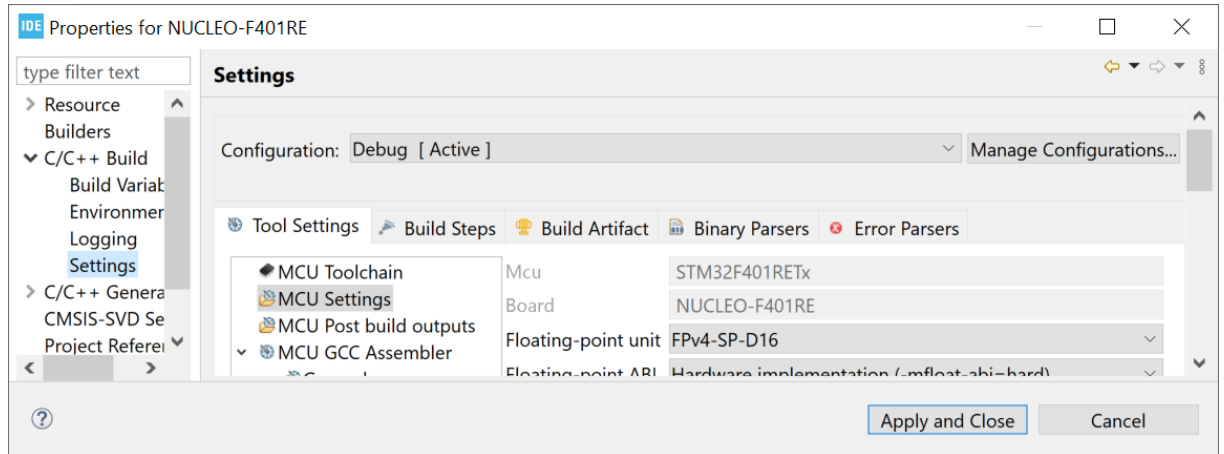
Update the name, description, or both and press **[OK]** to rename the *Debug2* configuration. In this case, press **[Cancel]** and keep the name.

2.3.2 Project C/C++ build settings

Each build configuration contains one project C/C++ build setting. The project C/C++ build setting is updated in project properties. To update the build setting, right-click on the project name in the *Project Explorer* view and select **[Properties]** or use the menu **[Project]>[Properties]**. Both these ways open the *Properties* window for the project.

Select **[C/C++ Build]>[Settings]** in the *Properties* left pane. The right part is then filled with tabs *Tool Settings*, *Build Steps*, *Build Artifact*, *Binary Parsers*, and *Error Parsers*. The first two tabs are the most useful ones.

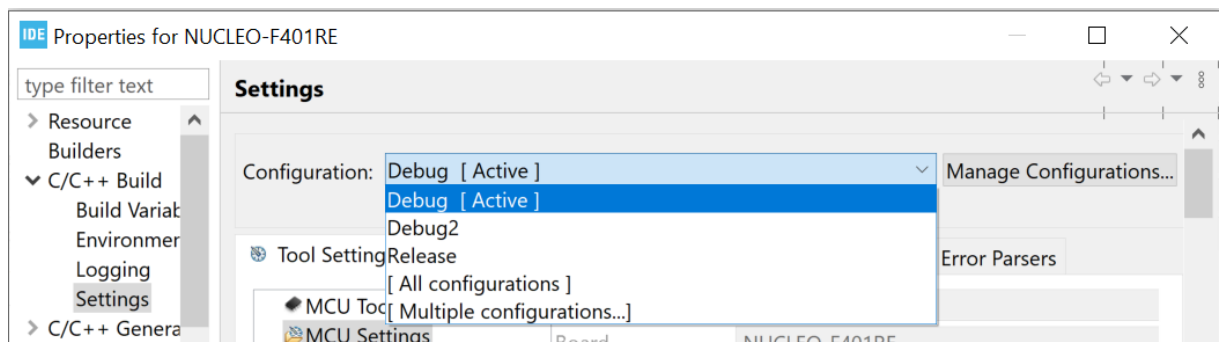
Figure 61. Properties tabs



Note: Resize the dialog window or use the top-right arrow buttons if all tabs are not visible.

The *Settings* pane contains a **[Configuration]** selection to decide if new selections are used in the active configuration only, in another configuration, in all configurations or in multiple configurations. Press **[Manage Configurations]** to open the *Manage Configurations* dialog.

Figure 62. Properties configurations

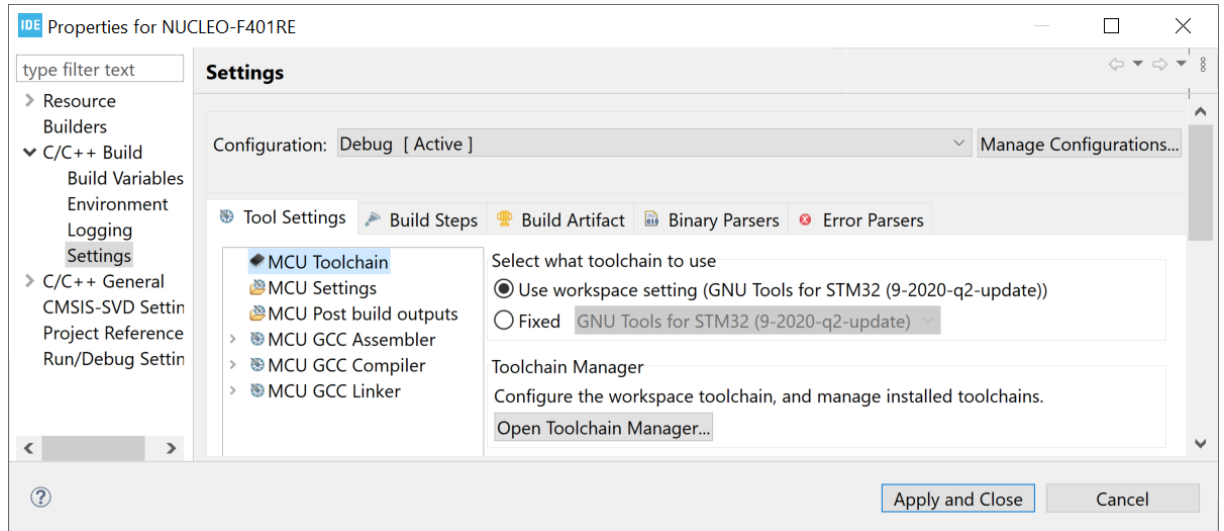


The *Tool Settings* tab is further split into *MCU Toolchain*, *MCU Settings*, *MCU Post build outputs*, *MCU GCC Assembler*, *MCU GCC Compiler* and *MCU GCC Linker*.

MCU Toolchain is used to change toolchains. *STM32CubeIDE* includes one version of the *GNU Tools for STM32* toolchain. The *Toolchain Manager* is used to download other *GNU ARM Embedded* toolchains and to configure to use local *GNU ARM Embedded* toolchains.

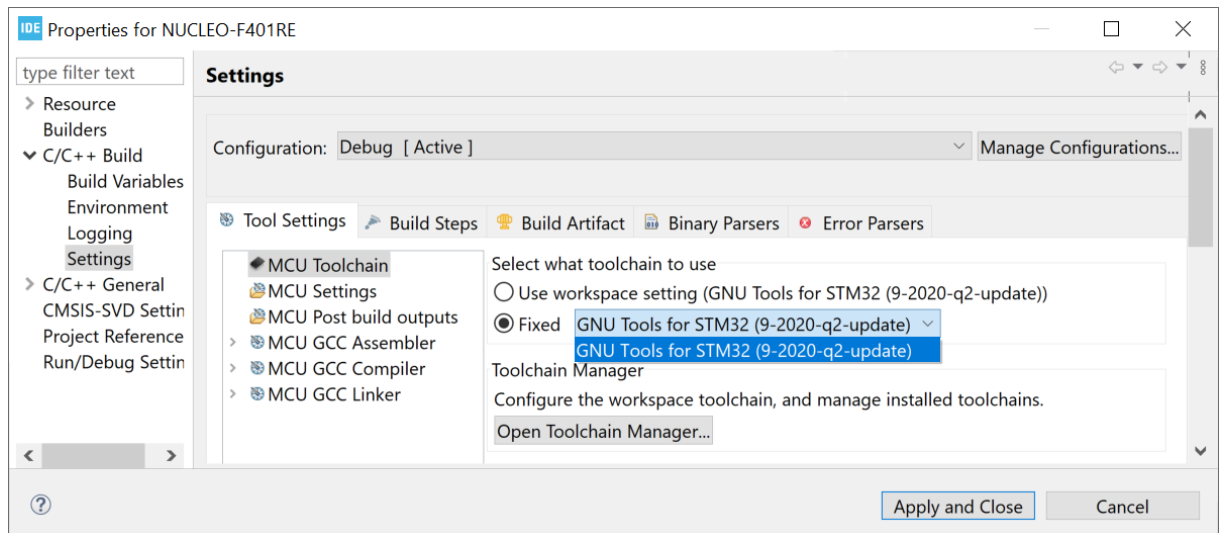
Information about patches made in *GNU Tools for STM32* can be read in [EXT-12]. The document can be opened from the *Technical Documentation* page in the *Information Center*.

Figure 63. Properties toolchain version



Select **[Fixed]** to enable the toolchain selection.

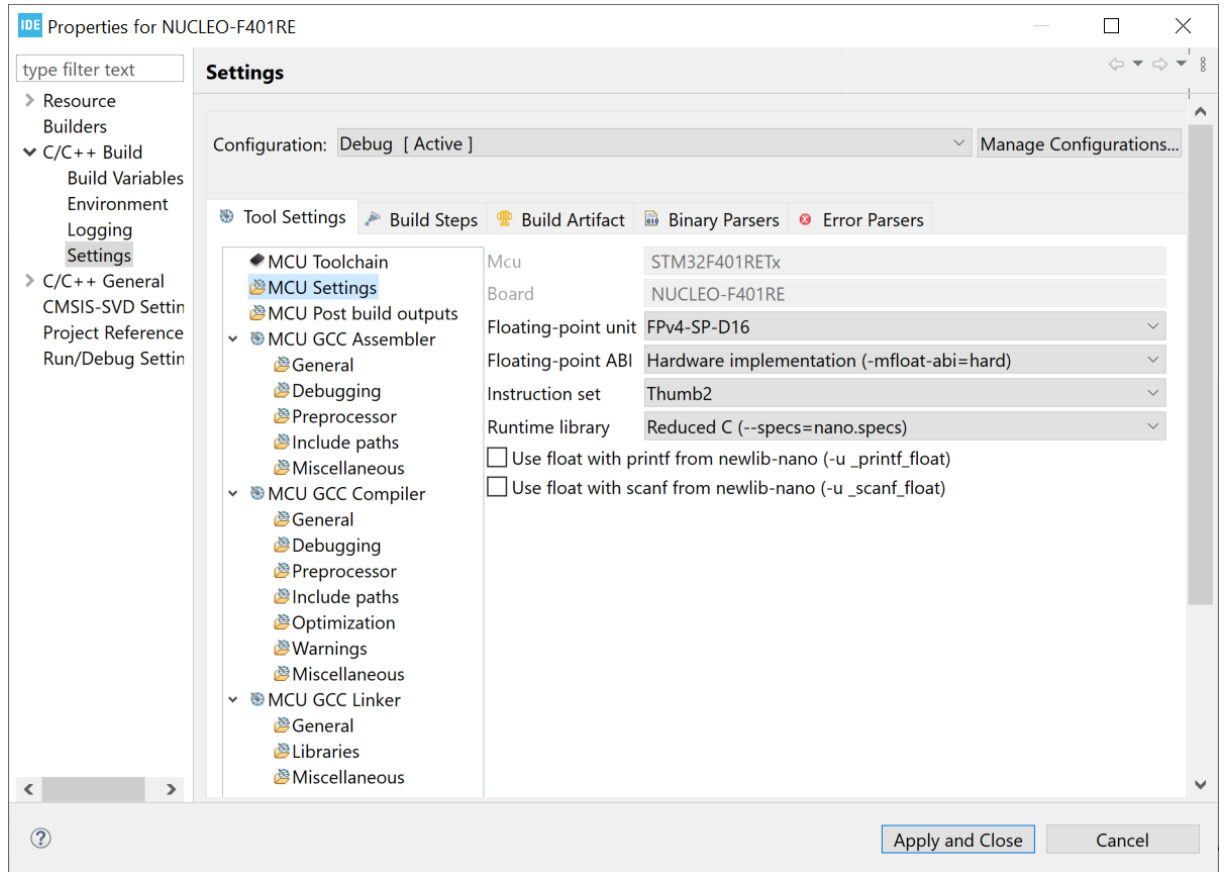
Figure 64. Properties toolchain selection



As shown in Figure 64, only the default toolchain *GNU Tools for STM32* is available by default. To install additional toolchains, click on the **[Open Toolchain Manager...]** button to open the *Toolchain Manager*. [Section 2.11 Toolchain Manager](#) contains detailed information on how to install, uninstall toolchains and select the default workspace toolchain.

MCU Settings displays the selected MCU and board for the project and proposes to select how to handle floating point, instruction set and runtime library.

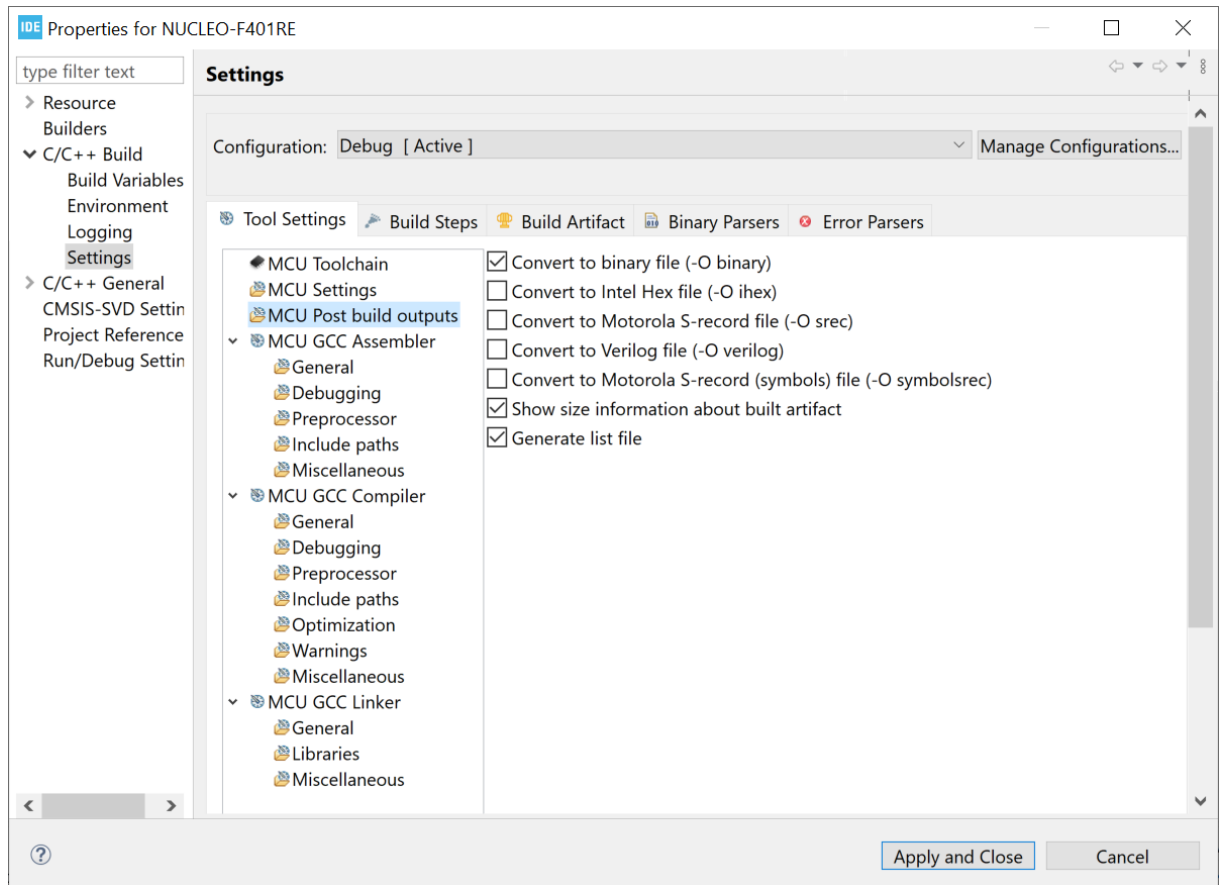
Figure 65. Properties tool MCU settings



MCU Post build outputs proposes to convert the `elf` file to another file format, show build size information, and generate list file. The output file can be converted to:

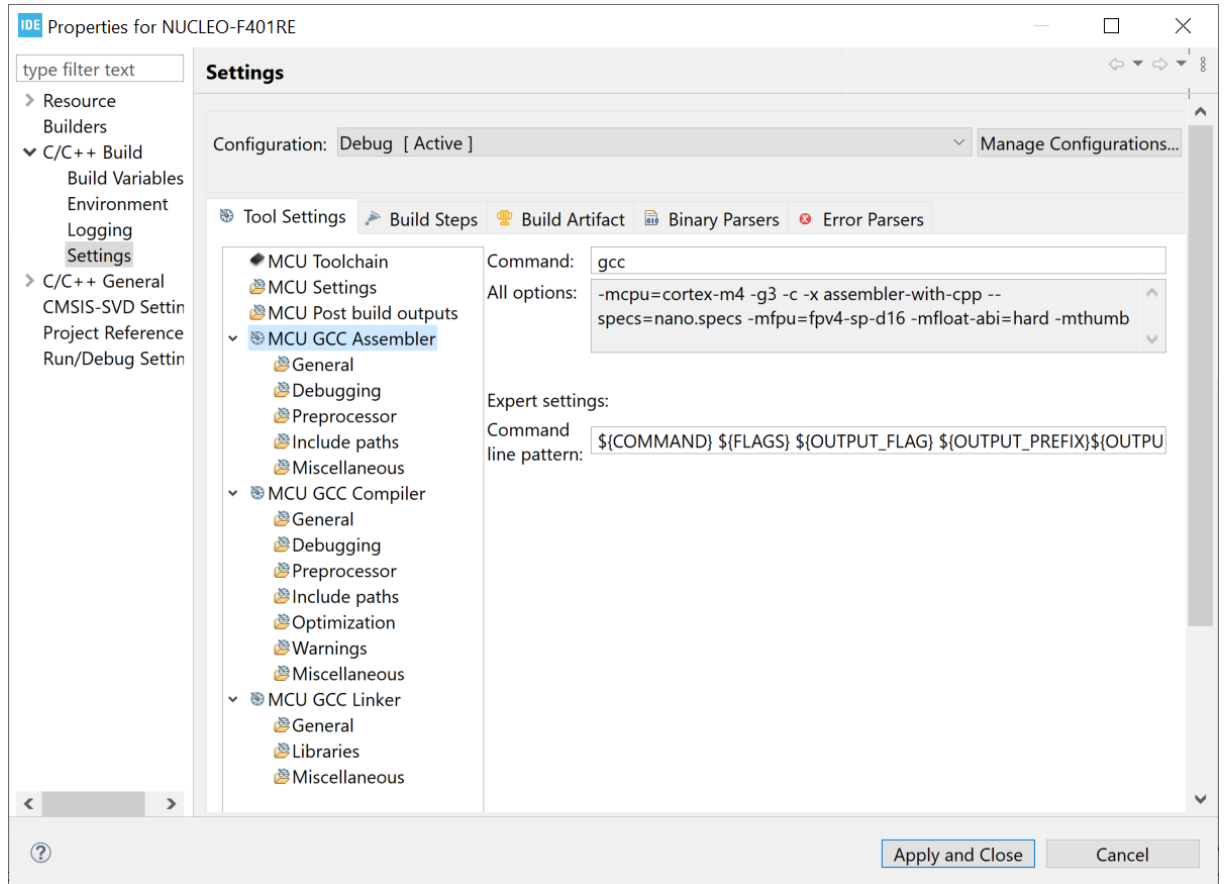
- Binary file
- Intel Hex file
- Motorola S-record file
- Motorola S-record symbols file
- Verilog file

Figure 66. Properties tool MCU post-build settings



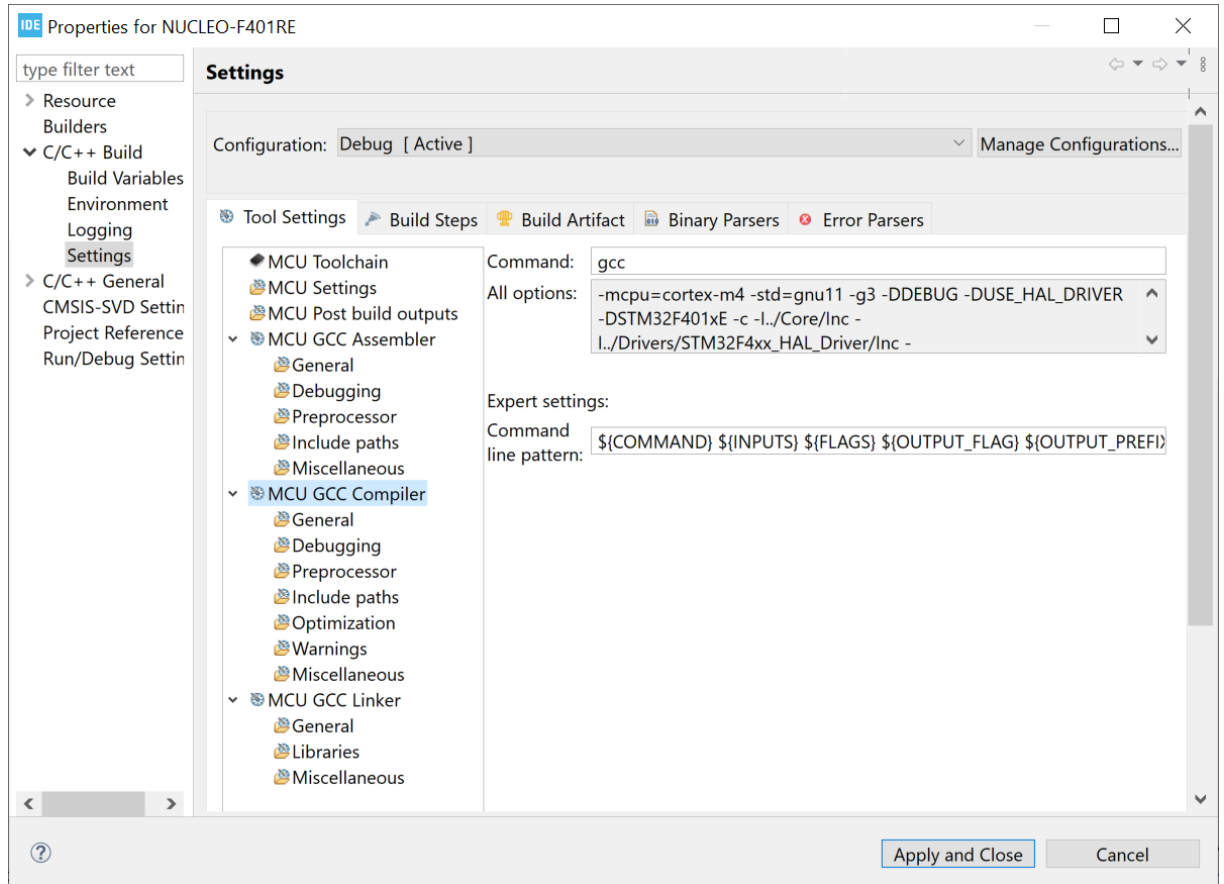
The *MCU GCC Assembler* settings contains selections for the assembler. The main node presents all the assembler command-line options that are currently enabled in the sub-node settings. The sub-nodes are used to view the current settings or change any settings for the assembler.

Figure 67. Properties tool GCC assembler settings



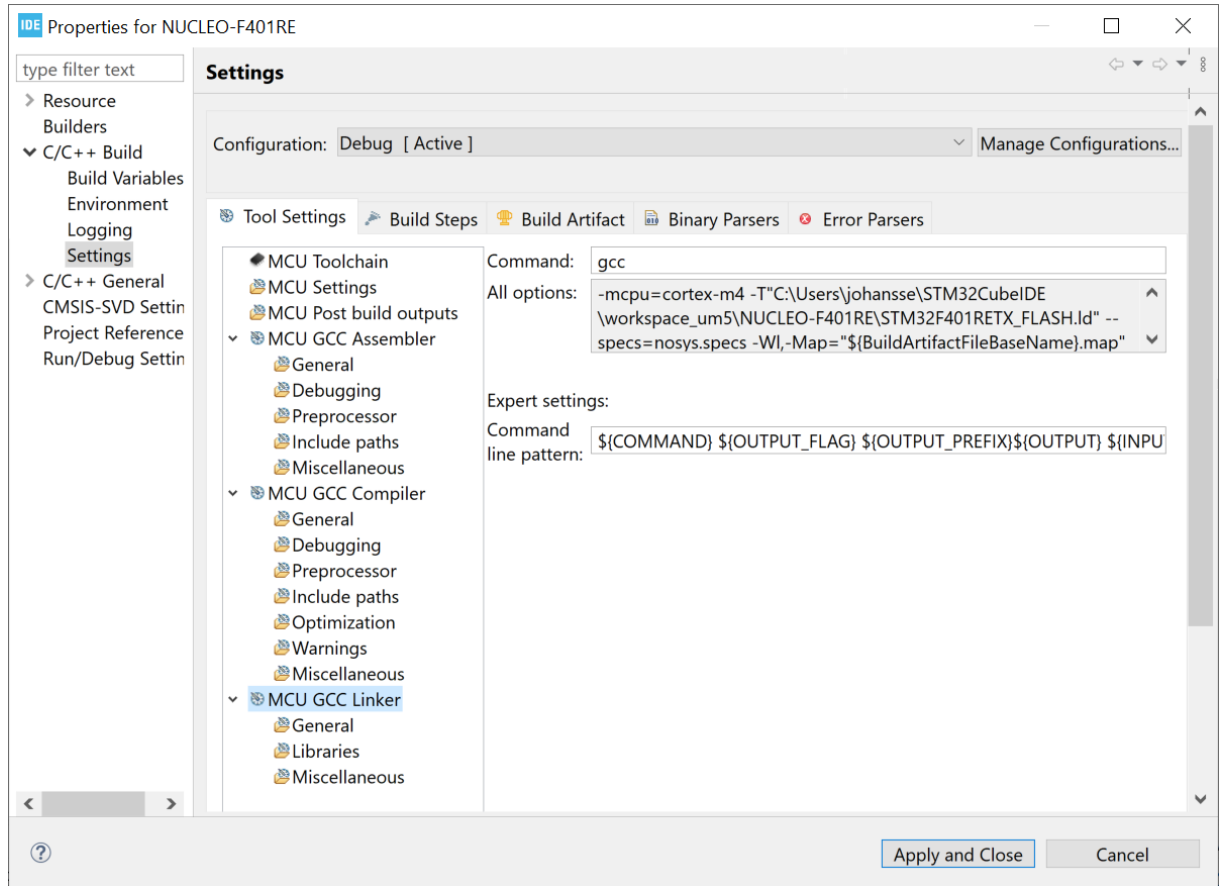
The *MCU GCC Compiler* settings contains selections for the compiler. The main node presents all the compiler command-line options that are currently enabled in the sub-node settings. The sub-nodes are used to view the current settings or change any settings for the compiler.

Figure 68. Properties tool GCC compiler settings



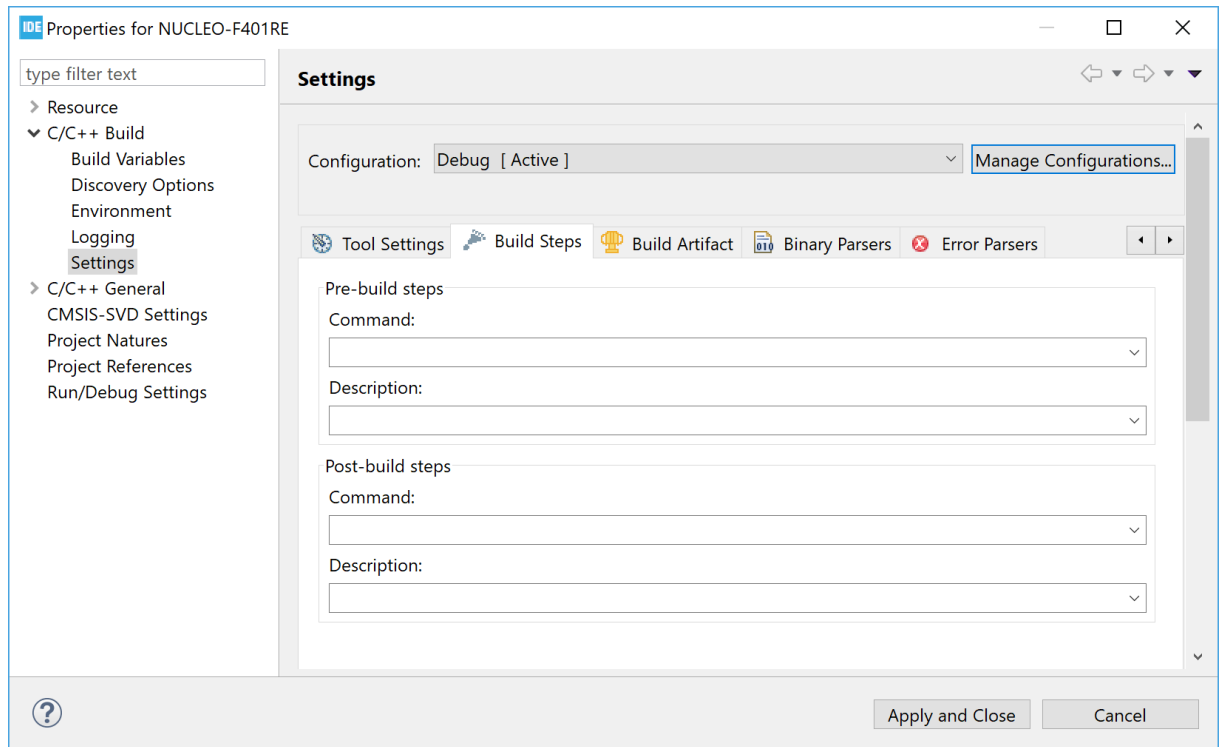
The *MCU GCC Linker* settings contains selections for the linker. The main node presents all the linker command-line options that are currently enabled in the sub-node settings. The sub-nodes are used to view the current settings or change any settings for the linker.

Figure 69. Properties tool GCC linker settings



The *Build Steps* settings contains fields used to provide pre-build and post-build steps, which run before and after building the project. Edit the fields to run any pre-build or post-build step.

Figure 70. Properties build steps settings

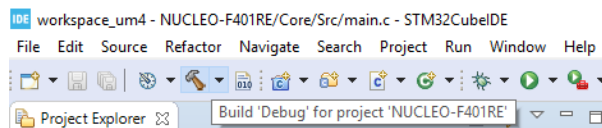


Note: It is possible to add more advanced post-build operations using makefile targets as described in Section 2.4.7 .

2.4 Building the project

To start a build, select the corresponding project in the *Project Explorer* view and click on the **[Build]**  toolbar button.

Figure 71. Project build toolbar

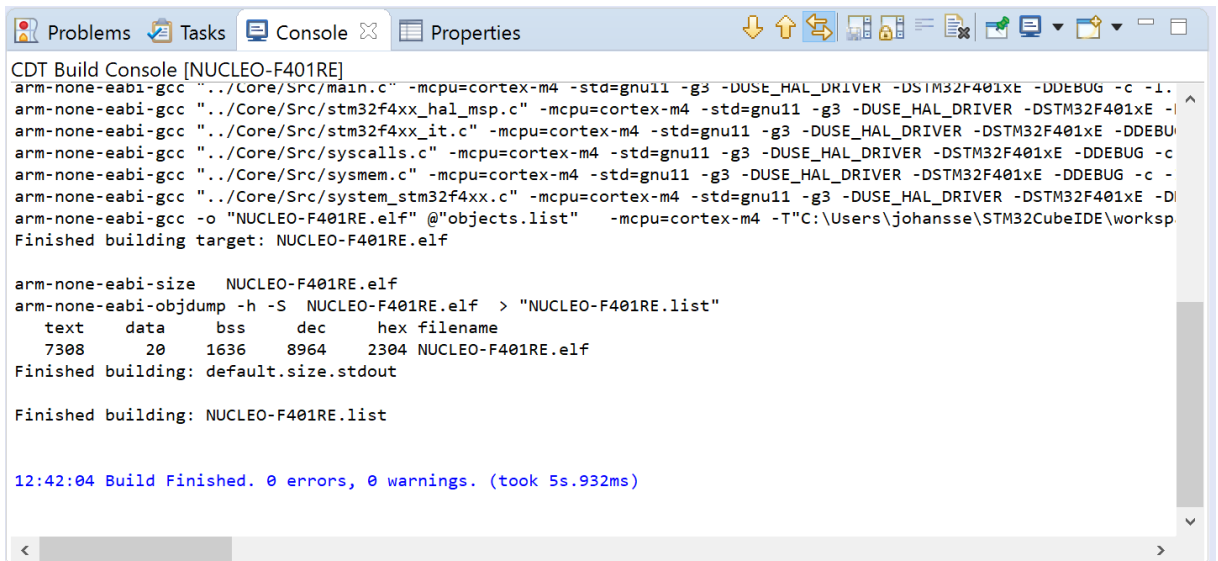


The build can also be started from menu **[Project]>[Build Project]**. The **[Project]** menu contains also some other usable build commands such as **[Build All]**, **[Build Project]** or **[Clean]**.


Another way to start a build is to right-click on the project in the *Project Explorer* view. This opens the context menu with the **[Build]** command and some other build options.

During the build, the *Console* view lists the build process. At the end, when the `elf` file is created normally, it lists size information.

Figure 72. Project build console

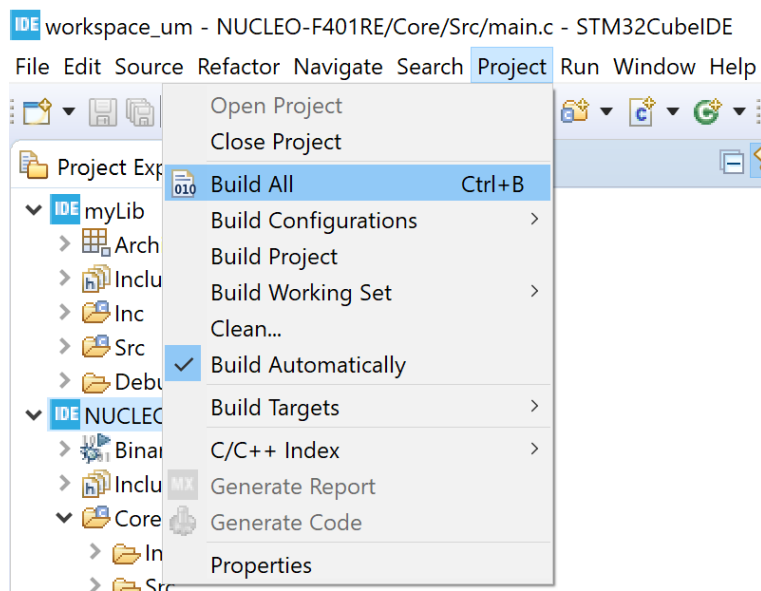


2.4.1 Building all projects

The toolbar contains the **[Build all]**  button, which is used to build the active build configuration for all open projects in workspace.

It is also possible to use the menu **[Project]>[Build All]** to start a build of all projects.

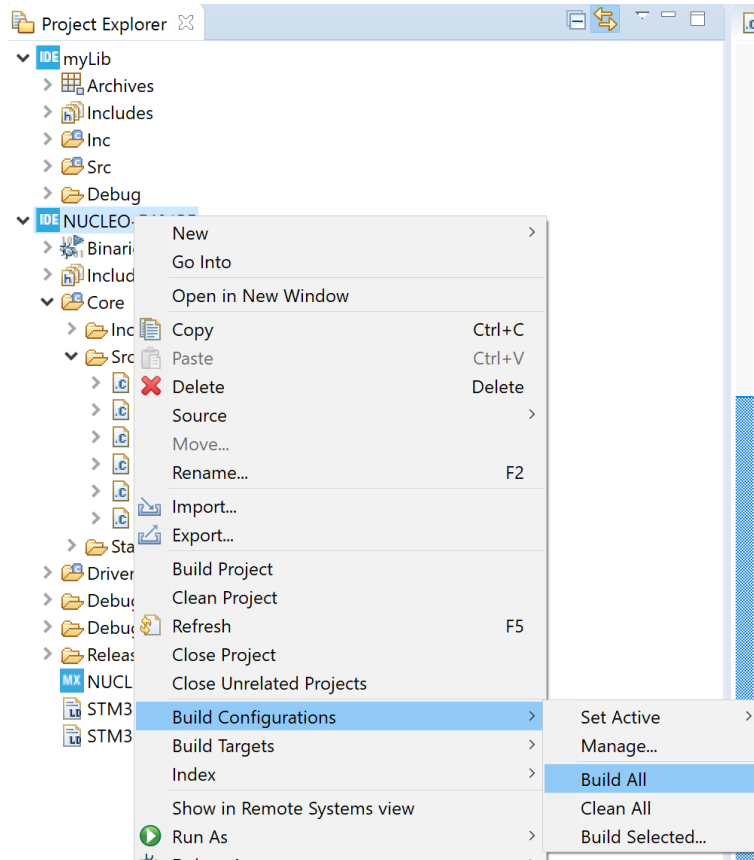
Figure 73. Project build all



2.4.2 Build all build configurations

To build all build configurations for a project, right-click the project and select **[Build Configurations]>[Build All]** in the context menu.

Figure 74. Project build-all configurations



2.4.3 Headless build

Headless build is intended to be used to build projects that must be integrated into script-controlled builds, such as nightly builds on build servers for continuous integration process methods or others. The STM32CubeIDE GUI is never displayed in this case, and the user is not requested any manual interaction with STM32CubeIDE.

STM32CubeIDE includes a `headless-build` command file to run headless builds. For instance, when using Windows®, it is located in the `C:\ST\STM32CubeIDE_1.7.0\STM32CubeIDE` STM32CubeIDE installation folder. The `headless-build.bat` file is intended to be run from a command prompt.

Note: *Before running any headless build, make sure that the workspace is not opened by STM32CubeIDE. If there is an STM32CubeIDE running already using the workspace, it is not possible for the headless-build process to open and build the project.*

To run headless build in Windows®, use the following procedure:

1. Open a command prompt.
2. Navigate to the STM32CubeIDE installation directory. Open the folder in which the IDE is stored.
For example: `cd C:\ST\STM32CubeIDE_1.7.0\STM32CubeIDE`
3. Enter the following command to build the NUCLEO-F401RE project in the workspace

```
C:\Users\Name\STM32CubeIDE\workspace_1.7.0:
$ headless-build.bat -data C:\Users\Name\STM32CubeIDE\workspace_1.7.0
-cleanBuild NUCLEO-F401RE
```

To get help on headless build parameters, use headless build with option `-help`. Figure 75 shows the result of command `$ headless-build.bat -help`.

Figure 75. Headless build

```

C:\ST>cd STM32CubeIDE_1.7.0.21alpha1
C:\ST\STM32CubeIDE_1.7.0.21alpha1>headless-build.bat -help
Usage: PROGRAM -data <workspace> -application org.eclipse.cdt.managedbuilder.core.headlessbuild [ OPTIONS ]

-data {/path/to/workspace}
-import {[uri:/]path/to/project}
-importAll {[uri:/]path/to/projectTreeURI} Import all projects under URI
-build {project_name_reg_ex{/config_reg_ex} | all}
-cleanBuild {project_name_reg_ex{/config_reg_ex} | all}
-markerType Marker types to fail build on {all | cdt | marker_id}
-no-indexer Disable indexer
-printErrorMarkers Print all error markers
-I {include_path} additional include path to add to tools
-include {include_file} additional include_file to pass to tools
-D {preproc_define} addition preprocessor defines to pass to the tools
-E {vars=value} replace/add value to environment variable when running all tools
-Ea {vars=value} append value to environment variable when running all tools
-Ep {vars=value} prepend value to environment variable when running all tools
-Er {var} remove/unset the given environment variable
-T {toolid} {optionid=value} replace a tool option value in each configuration build
-Ta {toolid} {optionid=value} append to a tool option value in each configuration build
-Tp {toolid} {optionid=value} prepend to a tool option value in each configuration build
-Tr {toolid} {optionid=value} remove a tool option value in each configuration build
Tool option values are parsed as a string, comma separated list of strings or a boolean based on the options type
  
```

2.4.4 Temporary assembly file and preprocessed C code

Save the temporary assembly file by adding the `-save-temps` flag to the compiler:

1. In the menu, select **[Project]>[Properties]**
2. Select **[C/C++ build]>[Settings]**
3. Open the *Tool Settings* tab
4. Add `-save-temps` in the **[C Compiler]>[Miscellaneous]** settings
5. Rebuild the project

The assembler file is located in the build output directory with name `filename.s`.

The file `filenamz.i` containing the preprocessed C code is generated also. It shows the code after the preprocessor but before the compilation. It is advise to examine the content of this file in case of problems with defines.

2.4.5 Build logging

To enable or disable project build logging, right-click on the project in the *Project Explorer* view and select **[Properties]**. Then, select **[C/C++ Build]>[Logging]**. The log file location and name are also specified.

To enable a global build log for all projects in a workspace, select **[Window], [Preferences]**, and open **[C/C++, Build, Logging]>[Enable global build logging]**.

2.4.6 Parallel build and build behaviour

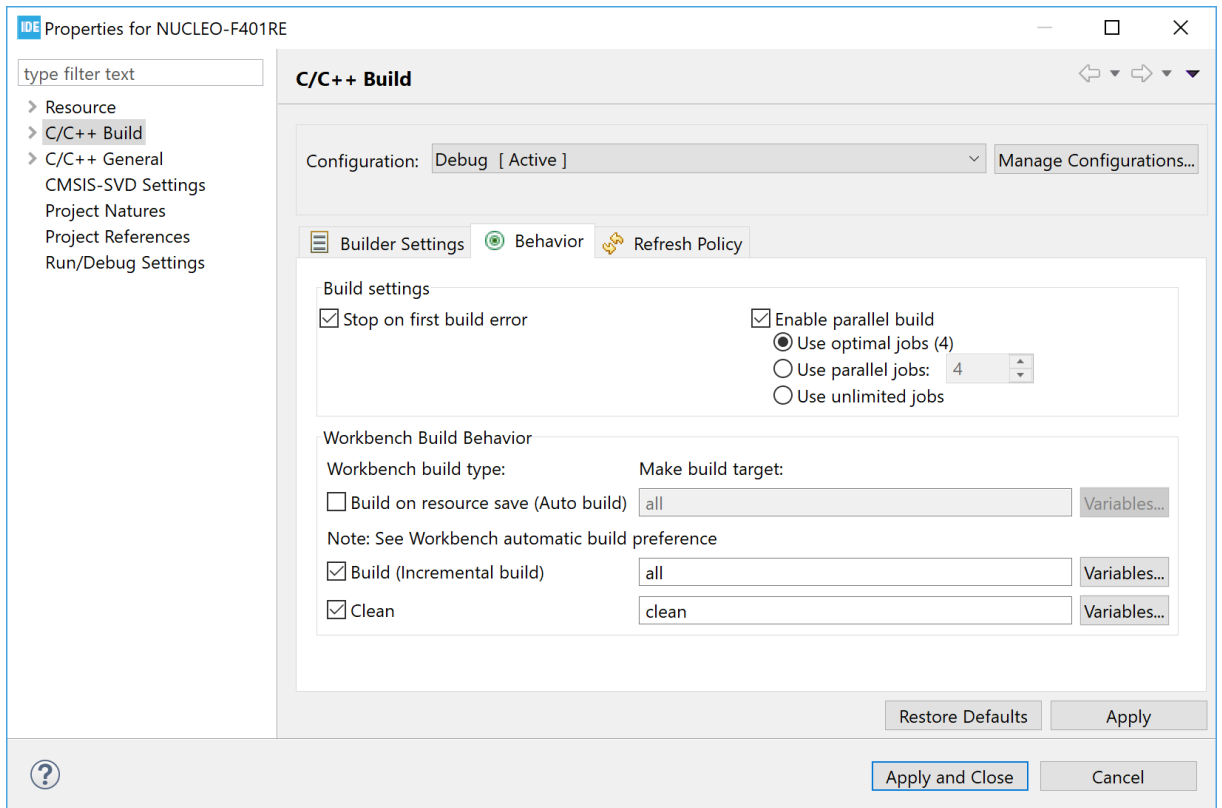
Parallel build occurs when more than one thread is used at the same time to compile and build the code. Most often, it reduces build time significantly. The optimal number of threads to use is usually equal to the number of CPU cores of the computer. Parallel build can be enabled and disabled.

To configure parallel build:

1. Right-click on the project in the *Project Explorer* view
2. Select menu **[Project]>[Properties]**
3. Select **[C/C++ Build]** in the *Properties* panel
4. Open the *Behavior* tab and configure **[Enable parallel build]**

The *Behavior* tab also contains build settings on how to behave on errors, build on resource save, incremental build, and clean.

Figure 76. Parallel build



2.4.7 Post-build with makefile targets

It is possible to add advanced post-build scripts by using makefile targets. To do this:

1. Create a new file
2. Name it `makefile.targets`
3. Place it in the root directory of the project

The content of the file must be similar to the example presented below. The example just copies the `elf` generated file to a new file and uses macros `BUILD_ARTIFACT`, `BUILD_ARTIFACT_PREFIX`, `BUILD_ARTIFACT_NAME`, and `BUILD_ARTIFACT_EXTENSION`, which are generated into the makefile by STM32CubeIDE from v1.5.0.

```
secure_target := \  
    $(BUILD_ARTIFACT_PREFIX)$(BUILD_ARTIFACT_NAME)-secure.$(BUILD_ARTIFACT_EXTENSION)  
main-build: $(secure_target)  
  
$(secure_target): $(BUILD_ARTIFACT)  
    # Do what you want here... simple copy file for demo  
    cp "$<" "$@"
```

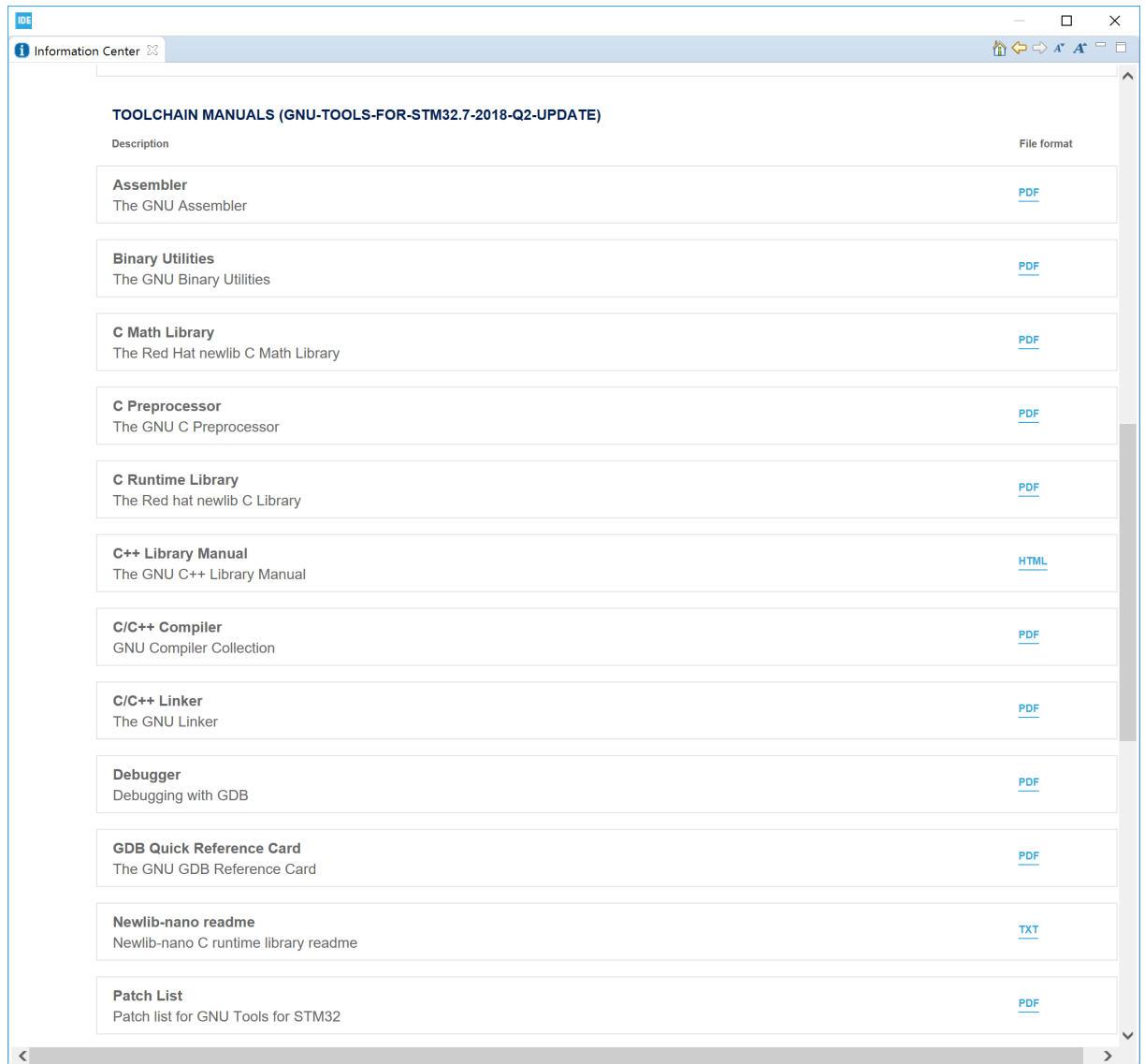
Note: *make* requires that tabs are used instead of spaces.

2.5 Linking the project

This section contains basic information about the linker and linker script files. Detailed information about the linker can be found in the *GNU Linker* manual ([EXT-05]), which is accessed from the *Information Center*. Click on the

[Information Center] toolbar button and open the *Information Center* view. Open the linker documentation using the [C/C++ Linker The GNU Linker PDF] link.

Figure 77. Linker documentation



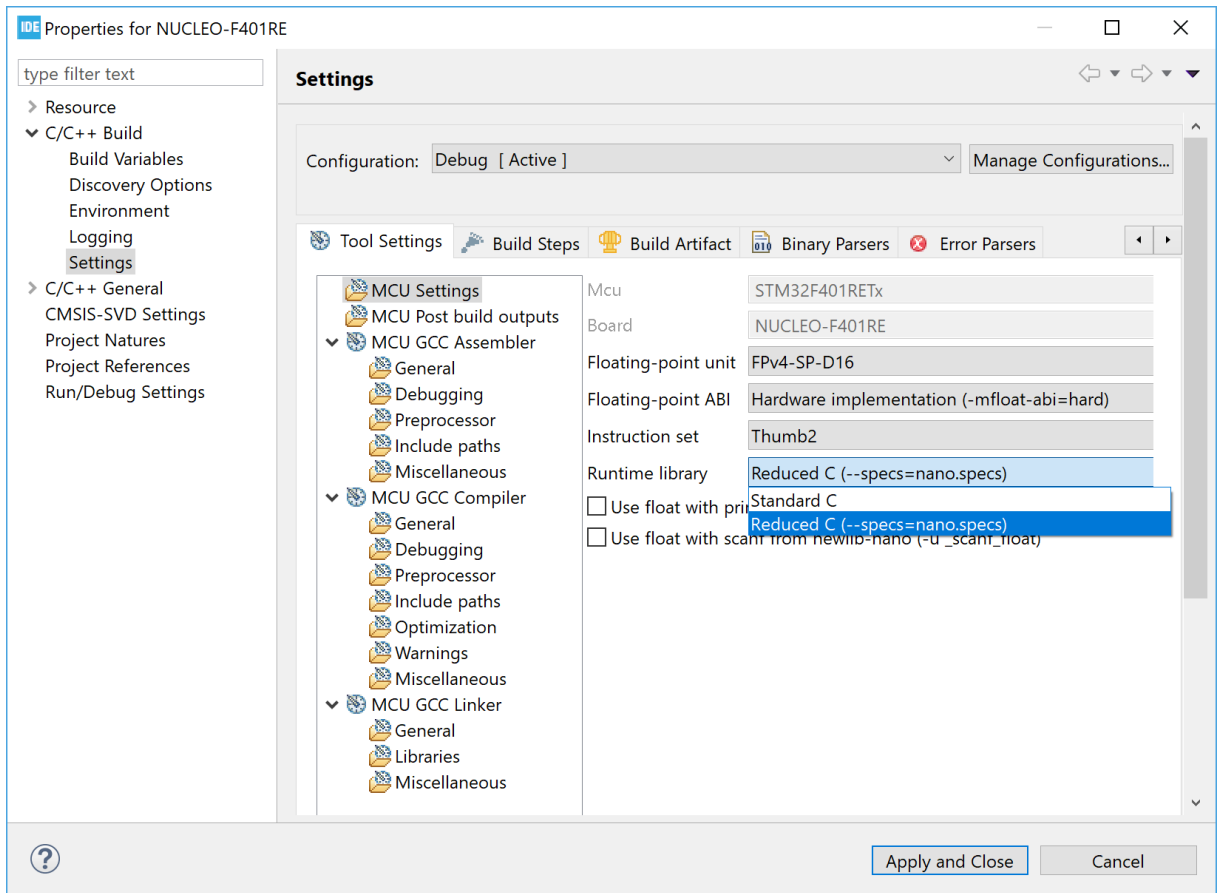
2.5.1 Run-time library

The toolchains included in [STM32CubeIDE](#) contain two prebuilt run-time C libraries based on `newlib`. One is the *standard C* `newlib` library and the other is the *reduced C* `newlib-nano`. Use `newlib-nano` to achieve smaller code size. For information about the differences between `newlib-nano` and the standard `newlib`, refer to the `newlib-nano` readme file ([\[ST-09\]](#)), accessible from the *Information Center*.

To select the desired run-time library for use in the project.

1. Right-click on the project in the *Project Explorer* view
2. Select menu **[Project]>[Properties]**
3. Select **[C/C++ Build]>[Settings]** in the *Properties* panel
4. Open the *Tool Settings* tab, select **[MCU Settings]** and configure the **[Runtime library]** setting

Figure 78. Linker run-time library



When `newlib-nano` is used while floating-point numbers must be handled by `scanf/printf`, additional options are required. The reason is that `newlib-nano` and `newlib` handle floating-point numbers differently. In `newlib-nano`, formatted floating-point number inputs and outputs are implemented as weak symbols. Therefore, the symbols must be pulled by specifying explicitly if `%f` is used with `scanf/printf` using the `-u` option:

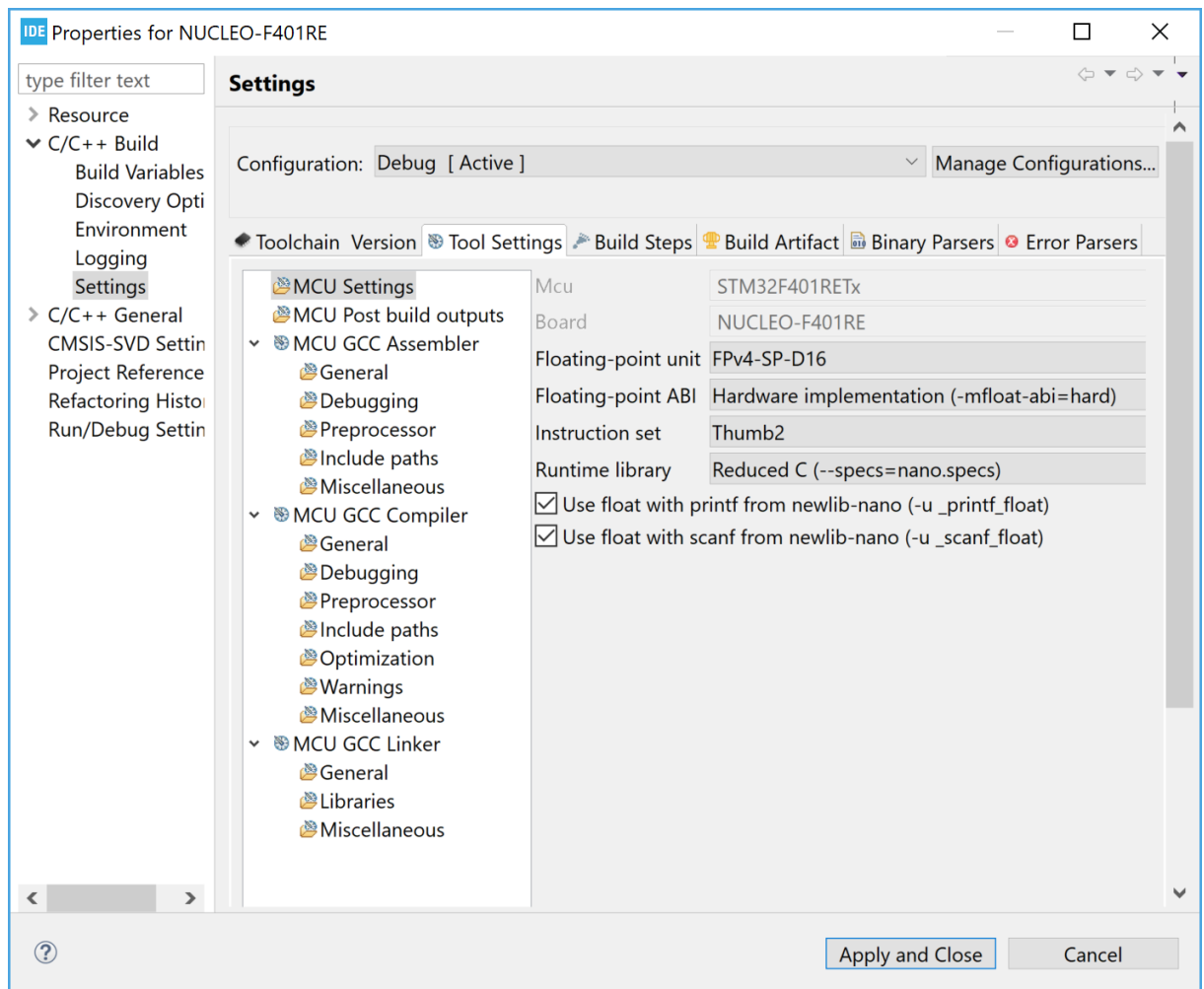
- `-u _scanf_float`
- `-u _printf_float`

For example, to enable output `float` with `printf`, the command line is as follows:

```
$ arm-none-eabi-gcc --specs=nano.specs -u _printf_float $(OTHER_LINK_OPTIONS)
```

The options can be enabled using the **[Use float ...]** checkboxes in **[MCU Settings]** in the *Tool Settings* tab.

Figure 79. Linker newlib-nano library and floating-point numbers



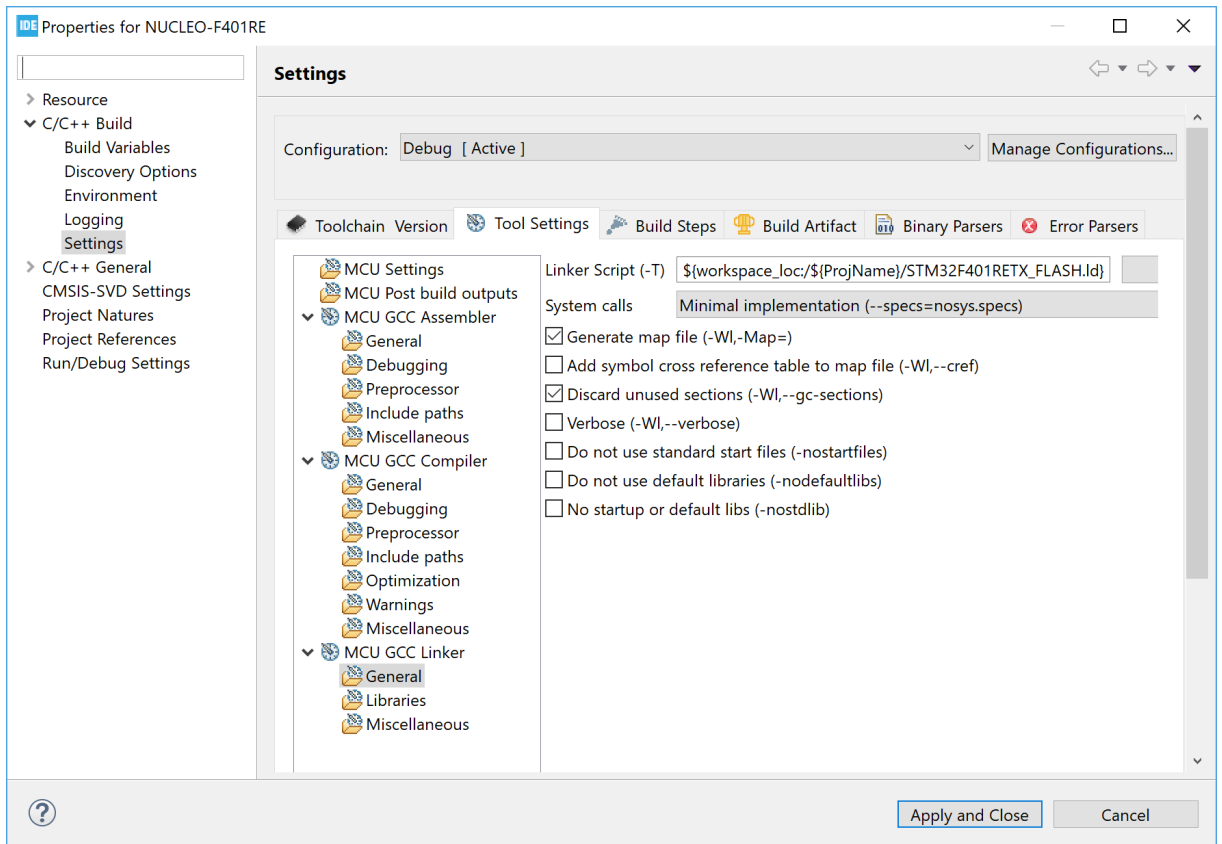
2.5.2 Discard unused sections

Linker optimization is the process where the linker removes unused code and data sections, dead code, from the output binary. Run-time and middleware libraries typically include many functions that are not used by all applications, thus wasting valuable memory unless removed from the output binary.

When using the project wizard to create new projects, the default configuration is that the linker discards unused sections. To check or change the setting about unused sections, open at any time the build settings for the project:

1. Right-click the project in the *Project Explorer* view and select **[Properties]**
2. In the dialog, select **[C/C++ Build]>[Settings]**
3. Select the *Tool Settings* tab in the panel
4. Select **[MCU GCC Linker]>[General]**
5. Configure **[Discard unused sections (-Wl,--gc-sections)]** according to the project requirements
6. Rebuild the project

Figure 80. Linker discard unused sections



2.5.3 Page size allocation for malloc

When the *GNU Tools for STM32* toolchain is used with the standard C `newlib` library, the page size setting for `malloc` can be changed. The `newlib` default page size is 4096 bytes. If a `sysconf()` function is implemented in the user project, this user function is called by `_malloc_r()`.

The following example shows how to implement a `sysconf()` function with a 128-byte page size. Add a similar function if there is a need for the application to use a smaller page size than the default 4096 bytes.

```
/**
*****
** File : sysconf.c
*****
**/

/* Includes */
#include <errno.h>
#include <unistd.h>

long sysconf(int name)
{
    if (name==_SC_PAGESIZE)
    {
        return 128;
    }
    else
    {
        errno=EINVAL;
        return -1;
    }
}
```

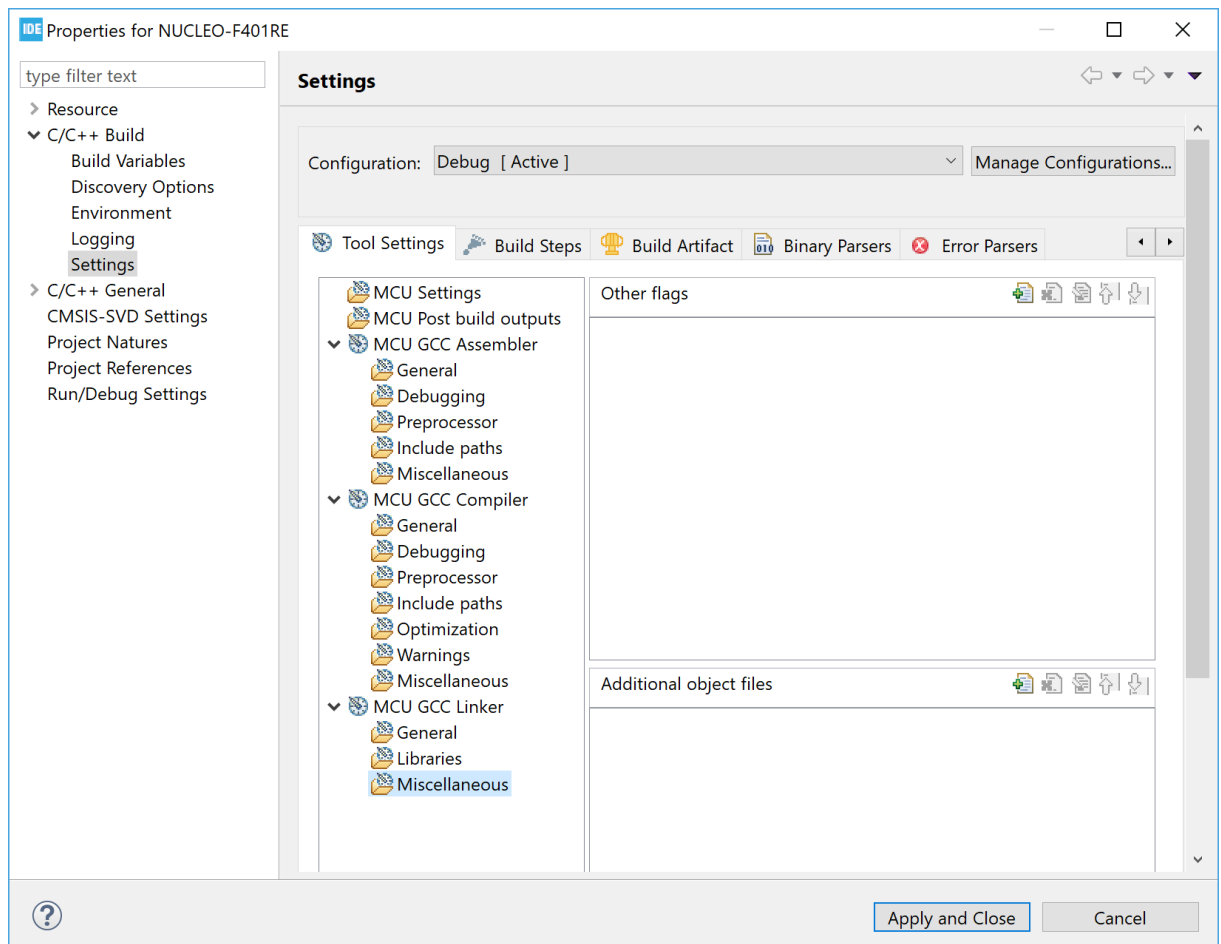
Note: If the “GNU ARM Embedded” toolchain is used, it does not call any `sysconf()` function implemented in the application but always uses the default `sysconf()` function in `newlib`. Also, no call to `sysconf()` is made if the “GNU Tools for STM32” toolchain is used with the reduced C `newlib-nano` library.

2.5.4 Include additional object files

STM32CubeIDE makes it easy to include additional object files that must be linked to a project. They can be files from other projects, precompiled libraries where no source code is available, or object files created with other compilers.

1. Right-click the project in the *Project Explorer* view and select **[Properties]**
2. In the dialog, select **[C/C++ Build]>[Settings]**
3. Select the *Tool Settings* tab in the panel
4. Select **[MCU GCC Linker]>[Miscellaneous]**
5. Use the **[Add...]** icon to add additional object files in several possible ways:
 - Enter the filenames in the *Add file path* dialog
 - Use the **[Workspace...]** or **[File system...]** buttons to locate the files

Figure 81. Linker include additional object files



2.5.5 Treat linker warnings and errors

The GNU linker is normally silent for warnings. One example of such silent warning is seen if the startup code containing the normal `Reset_Handler` function is missing in the project. The GNU linker in normal silent mode creates an `elf` file and only report a warning output in the *Console* window about the missing `Reset_Handler`.

Example of warning message:

```
arm-none-eabi-gcc -o "NUCLEO-F401RE.elf" @"objects.list" -mcpu=cortex-m4 -
T"C:\Users\username\STM32CubeIDE\workspace_um\NUCLEO-F401RE\STM32F401RETX_FLASH.ld"
--specs=nosys.specs -Wl,-Map="NUCLEO-F401RE.map" -Wl,--gc-sections -static -
mcpu=fpv4-sp-d16 -mfloat-abi=hard -mthumb -Wl,--start-group -lc -lm -Wl,--end-group
c:\st\stm32cubeide_1.1.0.19w37\stm32cubeide\plugins\com.st.stm32cube.ide.mcu.extern
altools.gnu-tools-for-stm32.7-2018-q2-update.win32_1.0.0.201904181610\tools\arm-
none-eabi\bin\ld.exe: warning: cannot find entry symbol Reset_Handler; defaulting
to 0000000008000000
Finished building target: NUCLEO-F401RE.elf
```

In this case, a new `elf` file is created but, if the warning is not detected, it will not work to debug the project because the program does not contain the `Reset_Handler` function. It is possible to configure the linker to treat warnings as errors by adding the `--fatal-warnings` option.

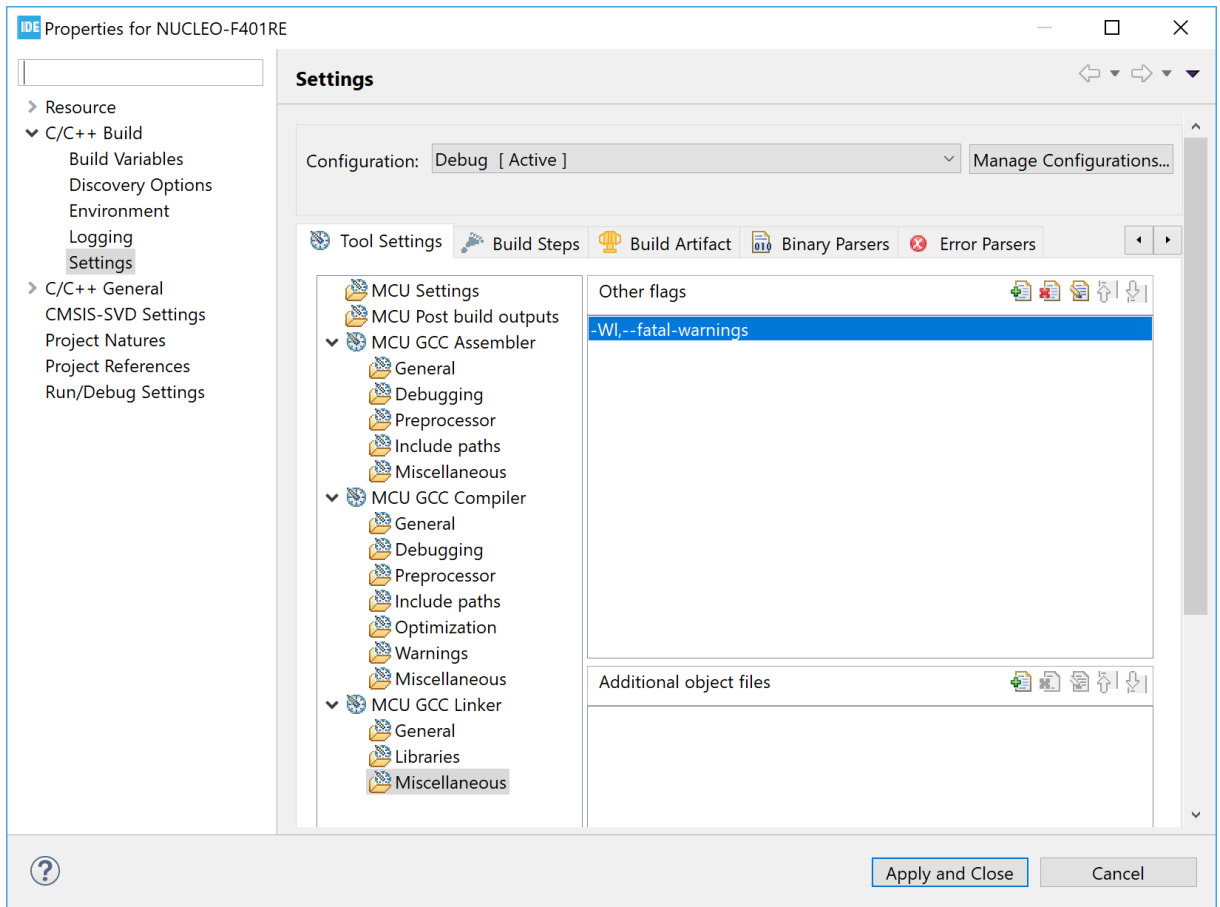
When the `--fatal-warnings` option is used, the linker does not generate the `elf` file but displays an error in the console log:

```
c:\st\stm32cubeide_1.1.0.19w37\stm32cubeide\plugins\com.st.stm32cube.ide.mcu.extern
altools.gnu-tools-for-stm32.7-2018-q2-update.win32_1.0.0.201904181610\tools\arm-
none-eabi\bin\ld.exe: warning: cannot find entry symbol Reset_Handler; defaulting
to 0000000008000000
collect2.exe: error: ld returned 1 exit status
make: *** [makefile:40: NUCLEO-F401RE.elf] Error 1
"make -j4 all" terminated with exit code 2. Build might be incomplete.
11:26:30 Build Failed. 1 errors, 6 warnings. (took 7s.193ms)
```

To use the `-Wl,--fatal-warnings` option:

1. Right-click the project in the *Project Explorer* view and select **[Properties]**
2. In the dialog, select **[C/C++ Build]>[Settings]**
3. Select the *Tool Settings* tab in the panel
4. Select **[MCU GCC Linker]>[Miscellaneous]**
5. Add `-Wl,--fatal-warnings` to the **[Other flags]** field.

Figure 82. Linker fatal warnings



2.5.6 Linker script

The linker script file (.ld) defines the files to include and where things end up in memory. Some important parts of the linker script file are described in the next sections. For detailed information about the linker, read the C/C++ linker *GNU Linker* manual ([ST-05]). This manual is available in the documentation section of the *Information Center*. Consider sections 3.6 and 3.7 especially.

The linker script specifies the memory regions and the location of the stack, heap, bss, data, rodata, text, and program entry. The size of stack and heap are configurable by editing the `_Min_Stack_Size` and `_Min_Heap_Size` values in the linker script file. However, these values are only used by the linker to validate that stack and heap fit in memory. When running the program, the stack or heap may require more memory, which may lead to unexpected results if data is overwritten.

Table 3 presents as an example the typical program and memory layout of an STM32F4 device with 512-Kbyte Flash memory and 96-Kbyte SRAM. The device is based on the Cortex®-M core with 32-bit address space (0x0000 0000 to 0xFFFF FFFF).

Table 3. Memory map layout

Example: STM32F4 96-Kbyte SRAM 512-Kbyte Flash memory	Usage	Files Linker script <code>.ld</code> , or <code>.h</code> and <code>.c</code> files	Comment
0xFFFF FFFF 0xE000 0000	Cortex-M4 internal peripherals.	<code>.h</code> and <code>.c</code> files.	SysTick, NVIC, ITM, debug and others.
0xDFFF FFFF 0x6000 0000	External memory FMC (Flexible memory controller).	Must be added in linker script, and <code>.h</code> and <code>.c</code> files. ⁽¹⁾	NOR Flash memory, NAND Flash memory, SPI Flash memory, PSRAM, SDRAM and others.
0x5FFF FFFF 0x4000 0000	STM32 Peripherals.	<code>.h</code> and <code>.c</code> files.	GPIO, ADC, timers, USB, USART and others.
0x2001 8000	96-Kbyte SRAM Stack	Linker script <code>_estack</code> <code>_Min_Stack_Size</code>	The stack contains local data ⁽²⁾
	Heap	<code>_Min_Heap_Size</code> <code>._user_heap_stack</code>	⁽³⁾ Heap used by malloc ⁽⁴⁾ Data
0x2000 0000	Data	<code>.bss</code> <code>.data</code>	Static global data (<code>.bss</code> and <code>.data</code>) <code>.bss</code> == Uninitialized data Cleared to zero by the startup code. <code>.data</code> == Initialized data Copied from Flash memory to SRAM by the startup code.
0x0808 0000	512-Kbyte Flash memory Data	Linker script <code>.data</code> <code>.rodata</code>	Initialized data to copy to SRAM. Read-only data placed in Flash memory.
	Program	<code>ENTRY</code> <code>Reset_Handler</code> ⁽⁵⁾ <code>.text</code>	<code>.text</code> == Program, such as <code>main()</code> in <code>main.c</code> , <code>SystemInit()</code> in <code>system_stm32f4xx.c</code> , <code>Reset_Handler</code> in <code>startup_stm32*.s</code> , <code>g_pfnVectors</code> in <code>startup_stm32*.s</code> , Vector table in <code>startup_stm32*.s</code> .
0x0800 0000	Interrupt vector table	<code>.isr_vector</code> ⁽⁶⁾	

Color legend

Cortex®-M internal peripherals and STM32 peripherals.

External memory. Normally the linker script, header files, and C files must be updated to use external memories.

Flash memory and SRAM where program, data, heap, and stack are located. Usually, when creating a project with STM32CubeIDE, these Flash memory and RAM regions are accessible and usable without any updates of the linker script or other files. The linker script file defines how to place code, data, heap, and stack in memory.

1. If external memory is used, the memory must be added into the linker script file. See in chapter [Section 2.5.7.1](#) how to add a new memory region.
2. The stack grows downwards and may go into the heap.
3. When running the program, the stack or heap may require more memory, which might lead to unexpected results if data is overwritten.
4. The heap grows upwards and may go into the stack.
5. The linker script file contains the entry point definition of the program. Normally, `ENTRY(Reset_Handler)`.
6. The interrupt vector table contains the reset value of the stack pointer, the start addresses of the program (`Reset_Handler`), exception handlers and interrupt handlers. Normally the `Reset_Handler` code and vector table (`g_pfnVectors`) are available in file `<startup_stm32xxx.s>`.

See below the default linker script generated by **STM32CubeIDE** for an STM32F4 device with 512-Kbyte Flash memory and 96-Kbyte SRAM.

The beginning of the code excerpt shows the linker script header, entry, stack, heap and memory definitions.

```

/**
*****
* @file      LinkerScript.ld
* @author    Auto-generated by STM32CubeIDE
* Abstract   : Linker script for NUCLEO-F401RE Board embedding STM32F401RETx Device from
stm32f4 series
*
*              512Kbytes FLASH
*              96Kbytes RAM
*
*              Set heap size, stack size and stack location according
*              to application requirements.
*
*              Set memory bank area and size if external memory is used
*****
* @attention
*
* <h2><center>&copy; Copyright (c) 2020 STMicroelectronics.
* All rights reserved.</center></h2>
*
* This software component is licensed by ST under BSD 3-Clause license,
* the "License"; You may not use this file except in compliance with the
* License. You may obtain a copy of the License at:
*
*              opensource.org/licenses/BSD-3-Clause
*
*****
*/

/* Entry Point */
ENTRY(Reset_Handler)

/* Highest address of the user mode stack */
_estack = ORIGIN(RAM) + LENGTH(RAM); /* end of "RAM" Ram type memory */

_Min_Heap_Size = 0x200; /* required amount of heap */
_Min_Stack_Size = 0x400; /* required amount of stack */

/* Memories definition */
MEMORY
{
  RAM      (xrw)  : ORIGIN = 0x20000000, LENGTH = 96K
  FLASH    (rx)   : ORIGIN = 0x80000000, LENGTH = 512K
}

```

The continuation of the code excerpt shows the definition of sections.

```

/* Sections */
SECTIONS
{
/* The startup code into "FLASH" Rom type memory */
.isr_vector :
{
. = ALIGN(4);
KEEP(*(.isr_vector)) /* Startup code */
. = ALIGN(4);
} >FLASH

/* The program code and other data into "FLASH" Rom type memory */
.text :
{
. = ALIGN(4);
*(.text) /* .text sections (code) */
*(.text*) /* .text* sections (code) */
*(.glue_7) /* glue arm to thumb code */
*(.glue_7t) /* glue thumb to arm code */
*(.eh_frame)

KEEP (*(.init))
KEEP (*(.fini))

. = ALIGN(4);
_etext = .; /* define a global symbols at end of code */
} >FLASH

/* Constant data into "FLASH" Rom type memory */
.rodata :
{
. = ALIGN(4);
*(.rodata) /* .rodata sections (constants, strings, etc.) */
*(.rodata*) /* .rodata* sections (constants, strings, etc.) */
. = ALIGN(4);
} >FLASH

.ARM.extab : {
. = ALIGN(4);
*(.ARM.extab* .gnu.linkonce.armextab.*)
. = ALIGN(4);
} >FLASH

.ARM : {
. = ALIGN(4);
__exidx_start = .;
*(.ARM.exidx*)
__exidx_end = .;
. = ALIGN(4);
} >FLASH

.preinit_array :
{
. = ALIGN(4);
PROVIDE_HIDDEN (__preinit_array_start = .);
KEEP (*(.preinit_array*))
PROVIDE_HIDDEN (__preinit_array_end = .);
. = ALIGN(4);
} >FLASH

.init_array :
{
. = ALIGN(4);
PROVIDE_HIDDEN (__init_array_start = .);
KEEP (*(SORT(.init_array.*)))
KEEP (*(.init_array*))
PROVIDE_HIDDEN (__init_array_end = .);
. = ALIGN(4);
} >FLASH

```

```

.fini_array :
{
  . = ALIGN(4);
  PROVIDE_HIDDEN (__fini_array_start = .);
  KEEP (*(SORT(.fini_array.*)))
  KEEP (*(.(fini_array*)))
  PROVIDE_HIDDEN (__fini_array_end = .);
  . = ALIGN(4);
} >FLASH

/* Used by the startup to initialize data */
_sidata = LOADADDR(.data);

/* Initialized data sections into "RAM" Ram type memory */
.data :
{
  . = ALIGN(4);
  _sdata = .;          /* create a global symbol at data start */
  *(.data)             /* .data sections */
  *(.data*)           /* .data* sections */
  *(.RamFunc)         /* .RamFunc sections */
  *(.RamFunc*)        /* .RamFunc* sections */

  . = ALIGN(4);
  _edata = .;         /* define a global symbol at data end */
} >RAM AT> FLASH

/* Uninitialized data section into "RAM" Ram type memory */
. = ALIGN(4);
.bss :
{
  /* This is used by the startup in order to initialize the .bss section */
  _sbss = .;          /* define a global symbol at bss start */
  __bss_start__ = _sbss;
  *(.bss)
  *(.bss*)
  *(COMMON)

  . = ALIGN(4);
  _ebss = .;         /* define a global symbol at bss end */
  __bss_end__ = _ebss;
} >RAM

/* User_heap_stack section, used to check that there is enough "RAM" Ram type memory left
*/
.user_heap_stack :
{
  . = ALIGN(8);
  PROVIDE ( end = . );
  PROVIDE ( _end = . );
  . = . + _Min_Heap_Size;
  . = . + _Min_Stack_Size;
  . = ALIGN(8);
} >RAM

/* Remove information from the compiler libraries */
/DISCARD/ :
{
  libc.a ( * )
  libm.a ( * )
  libgcc.a ( * )
}

.ARM.attributes 0 : { *(.ARM.attributes) }
}

```

2.5.6.1 The **ENTRY** command defines the start of the program

The first instruction to execute in a program is defined with the `ENTRY` command.

Example:

```
/* Entry Point */  
ENTRY(Reset_Handler)
```

The `ENTRY` information is used by GDB so that the program counter (PC) is set to the value of the `ENTRY` address when a program is loaded. In the example, the program starts to execute from `Reset_Handler` when a `step` or `continue` command is given to GDB after a load.

Note: *The start of the program can be overridden if the GDB script contains a monitor reset command after the load command. Then the code starts to run from reset.*

2.5.6.2 Stack location

The stack location is normally used by the startup file using the `_estack` symbol. The startup code normally initializes the stack pointer with the address given in the linker script. For Cortex[®]-M based devices, the stack address is also set at the first address in the interrupt vector table.

Example:

```
/* Highest address of the user mode stack */  
_estack = ORIGIN(RAM) + LENGTH(RAM); /* end of "RAM" Ram type memory */
```

2.5.6.3 Define heap and stack minimum sizes

It is common to define in the linker script the heap and stack minimum sizes to be used by the system.

Example:

```
_Min_Heap_Size = 0x200; /* required amount of heap */  
_Min_Stack_Size = 0x400; /* required amount of stack */
```

The values defined here are normally used later in the linker script to make it possible for the linker to test if the heap and stack fit in the memory. The linker can then issue an error if there is not enough memory available.

2.5.6.4 Specify memory regions

The memory regions are specified with names `ORIGIN` and `LENGTH`. It is common also to have an attribute list specifying the usage of a particular memory region, such as `(rx)` with “r” standing for read-only section and “x” for executable section. It is not required to specify any attribute.

Example:

```
/* Memories definition */  
MEMORY  
{  
  RAM      (xrw)  : ORIGIN = 0x20000000,  LENGTH = 96K  
  FLASH    (rx)   : ORIGIN = 0x80000000,  LENGTH = 512K  
}
```

2.5.6.5 Specify output sections (.text and .rodata)

The output sections define where the sections such as '.text', '.data' or others are located in the memory. The example below tells the linker to put all sections such as .text, .rodata and others in the Flash memory region. The glue sections mentioned in the example are used by GCC if there are some mixed code in the program. For instance, the glue code is used if some Arm[®] code makes a call to thumb code or vice versa.

Example:

```

/* Sections */
SECTIONS
{
  /* The startup code into "FLASH" Rom type memory */
  .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector)) /* Startup code */
    . = ALIGN(4);
  } >FLASH

  /* The program code and other data into "FLASH" Rom type memory */
  .text :
  {
    . = ALIGN(4);
    *(.text)           /* .text sections (code) */
    *(.text*)          /* .text* sections (code) */
    *(.glue_7)         /* glue arm to thumb code */
    *(.glue_7t)        /* glue thumb to arm code */
    *(.eh_frame)

    KEEP (*(.init))
    KEEP (*(.fini))

    . = ALIGN(4);
    _etext = .;        /* define a global symbols at end of code */
  } >FLASH

```

2.5.6.6 Specify initialized data (.data)

Initialized data values require extra handling as the initialization values must be placed in the Flash memory and the startup code must be able to initialize the RAM variables with correct values. The example below creates symbols _sidata, _sdata and _edata. The startup code can then use these symbols to copy the values from Flash memory to RAM during program start.

Example:

```

/* Used by the startup to initialize data */
_sidata = LOADADDR(.data);

/* Initialized data sections into "RAM" Ram type memory */
.data :
{
  . = ALIGN(4);
  _sdata = .;        /* create a global symbol at data start */
  *(.data)           /* .data sections */
  *(.data*)          /* .data* sections */
  *(.RamFunc)        /* .RamFunc sections */
  *(.RamFunc*)       /* .RamFunc* sections */

  . = ALIGN(4);
  _edata = .;        /* define a global symbol at data end */
} >RAM AT> FLASH

```

2.5.6.7 Specify uninitialized data (.bss)

Uninitialized data values must be reset to 0 by the startup code: the linker script file must identify the locations of these variables. The example below creates symbols `_sbss` and `_ebss`. The startup code can then use these symbols to set the values of the uninitialized variables to 0.

Example:

```
/* Uninitialized data section into "RAM" Ram type memory */
. = ALIGN(4);
.bss :
{
  /* This is used by the startup in order to initialize the .bss section */
  _sbss = .;          /* define a global symbol at bss start */
  __bss_start__ = _sbss;
  *(.bss)
  *(.bss*)
  *(COMMON)

  . = ALIGN(4);
  _ebss = .;         /* define a global symbol at bss end */
  __bss_end__ = _ebss;
} >RAM
```

2.5.6.8 Check if user heap and stack fit in the RAM

One section of the code is normally dedicated to linker checks about the fact that the needed heap and stack fit into the RAM together with all other data.

Example:

```
/* User_heap_stack section, used to check that there is enough "RAM" Ram type memory left */
*_user_heap_stack :
{
  . = ALIGN(8);
  PROVIDE ( end = . );
  PROVIDE ( _end = . );
  . = . + _Min_Heap_Size;
  . = . + _Min_Stack_Size;
  . = ALIGN(8);
} >RAM
```

Note: *The stack is placed on top of RAM and heap after data with a gap in between. See [Table 3. Memory map layout](#).*

2.5.6.9 Linker map and list files

When building a project generated with [STM32CubeIDE](#), a map and a list file are created in the debug or release build output folders. These files contain detailed information on the final locations of code and data in the program.

The Build Analyzer view can be used to analyse the size and location of a program in detail. Read more about this in [Section 8 Build Analyzer](#).

2.5.7 Modify the linker script

This section presents common use cases requiring to edit the linker script. Editing and managing the script allows for more exact placements of the code and data.

2.5.7.1 Place code in a new memory region

Many devices have more than one memory region. It is possible to use the linker script to specifically place code in different areas. The example below shows how to update a linker script to support code to be placed in a new memory region named `IP_CODE`.

Example:

```
Original MEMORY AREA

/* Memories definition */
MEMORY
{
  RAM      (xrw)   : ORIGIN = 0x20000000,   LENGTH = 96K
  FLASH   (rx)    : ORIGIN = 0x80000000,   LENGTH = 512K
}

Add IP_CODE into MEMORY AREA

/* Memories definition */
MEMORY
{
  RAM      (xrw)   : ORIGIN = 0x20000000,   LENGTH = 96K
  FLASH   (rx)    : ORIGIN = 0x80000000,   LENGTH = 256K
  IP_CODE (rx)    : ORIGIN = 0x80400000,   LENGTH = 256K
}
```

Place the following code a bit further down in the script, between the `.data { ... }` and the `.bss { ... }` section in the linker script file:

Example:

```
.ip_code :
{
  *(.IP_Code*);
} > IP_CODE
```

This tells the linker to place all sections named `.IP_Code*` into the `IP_CODE` memory region, which is specified to start at target memory address `0x804 0000`.

In the C code, tell the compiler which functions must go to this section by adding `__attribute__((section(".IP_Code")))` before the function declaration.

Example:

```
__attribute__((section(".IP_Code"))) int myIP_read()
{
  // Add code here...
  return 1;
}
```

The `myIP_read()` function is now placed in the `IP_CODE` memory region by the linker.

2.5.7.2 Place code in RAM

To place code in the RAM, some modifications of the linker script and startup code are needed. The example below describes the changes to be applied when the internal RAM is split into a few sections and the code is placed and executed in one of the internal RAM sections.

Define a new memory region in the MEMORY {} region in the linker script:

```
Original MEMORY AREA

/* Memories definition */
MEMORY
{
  RAM      (xrw)   : ORIGIN = 0x20000000,   LENGTH = 96K
  FLASH   (rx)    : ORIGIN = 0x80000000,   LENGTH = 512K
}

Split RAM into memory areas RAM1, RAM_CODE, RAM

/* Memories definition */
MEMORY
{
  RAM1     (xrw)   : ORIGIN = 0x20000000,   LENGTH = 16K
  RAM_CODE (xrw)   : ORIGIN = 0x20004000,   LENGTH = 16K
  RAM      (xrw)   : ORIGIN = 0x20008000,   LENGTH = 64K
  FLASH   (rx)    : ORIGIN = 0x80000000,   LENGTH = 512K
}
```

Define an output section for the code in the linker script. This must be placed with a Load Memory Address (LMA) belonging to the Flash memory, and a Virtual Memory Address (VMA) in RAM:

```
/* load code used by the startup code to initialize the ram code */
_siram_code = LOADADDR(.RAM_CODE);
.RAM_CODE :
{
  . = ALIGN(4);
  _sram_code = .; /* create a global symbol at ram_code start */
  *(.RAM_Code) /* .RAM_Code sections */
  *(.RAM_Code*) /* .RAM_Code* sections */
  . = ALIGN(4);
  _eram_code = .; /* define a global symbol at ram_code end */
} >RAM_CODE AT> FLASH
```

The RAM code area must be initialized and code copied from the Flash memory to the RAM code area. The startup code can access the location information symbols `_siram_code`, `_sram_code` and `_eram_code`. Add load address symbols for `RAM_CODE` into the startup file:

```
/* Load address for RAM_CODE */
.word _siram_code;
.word _sram_code;
.word _eram_code;
```

Add a piece of code into the startup code to copy the RAM code from the Flash memory (LMA) to the RAM (VMA):

```
Reset_Handler:
    ldr    sp, =_estack          /* set stack pointer */

/* Copy the ram code from flash to RAM */
    movs  r1, #0
    b     LoopRamCodeInit

RamCodeInit:
    ldr   r3, =_siram_code
    ldr   r3, [r3, r1]
    str   r3, [r0, r1]
    adds  r1, r1, #4

LoopRamCodeInit:
    ldr   r0, =_sram_code
    ldr   r3, =_eram_code
    adds  r2, r0, r1
    cmp   r2, r3
    bcc   RamCodeInit

/* Copy the data segment initializers from flash to SRAM */
    movs  r1, #0
    b     LoopCopyDataInit

CopyDataInit:
```

In the C code, instruct the compiler about which functions must go to this section by adding `__attribute__((section(".RAM_Code")))` before the functions declarations:

```
__attribute__((section(".RAM_Code"))) int myRAM_read()
{
    // Add code here...
    return 2;
}
```

Refer to [\[ST-12\]](#) for information on how to execute application code from CCM SRAM using STM32CubeIDE. It contains examples on how to setup the linker script and startup code to execute a function or an interrupt handler from RAM. The example in the chapter 4 of [\[ST-12\]](#) can be used as an inspiration on how to add other RAM regions and setup code sections to be located in RAM.

2.5.7.3 Place variables at specific addresses

It is possible to place variables at specific addresses in the memory. To achieve this, the linker script must be modified. The example presented in this section places constant variables handling a product `VERSION_NUMBER`, `CRC_NUMBER`, and `BUILD_ID` in memory.

The first step is to create a new memory region in the linker script:

```
Original MEMORY AREA

/* Memories definition */
MEMORY
{
  RAM      (xrw)      : ORIGIN = 0x20000000,    LENGTH = 96K
  FLASH    (rx)       : ORIGIN = 0x80000000,    LENGTH = 512K
}

Add a new 2K FLASH_V memory region at end of flash
/* Memories definition */
MEMORY
{
  RAM      (xrw)      : ORIGIN = 0x20000000,    LENGTH = 96K
  FLASH    (rx)       : ORIGIN = 0x80000000,    LENGTH = 512K-2K
  FLASH_V  (rx)       : ORIGIN = 0x807F800,     LENGTH = 2K
}

```

At this point, the memory section must be added:

```
Place the following a bit further down in the script, between the .data { ... } and the .bss
{ ... } section

.flash_v :
{
*(.flash_v*);
} > FLASH_V

```

This instructs the linker to place all sections named `flash_v*` into the `flash_v` output section in the `FLASH_V` memory region, which is specified to start at target memory address `0x807 F800`.

A section can be called almost anything except some predefined names such as `"data"`.

Now, the variables that must be located into the `FLASH_V` memory must be defined with attributes in the C files:

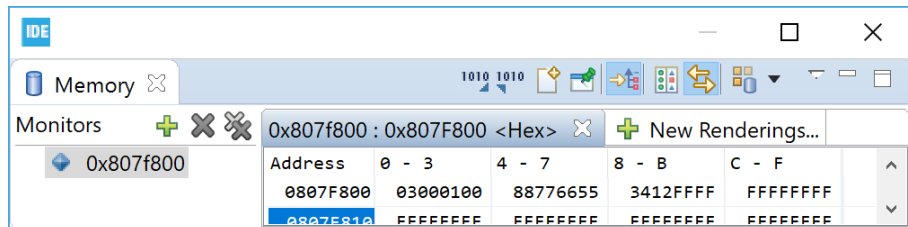
```
__attribute__((section(".flash_v.VERSION"))) const uint32_t VERSION_NUMBER=0x00010003;
__attribute__((section(".flash_v.CRC"))) const uint32_t CRC_NUMBER=0x55667788;
__attribute__((section(".flash_v.BUILD_ID"))) const uint16_t BUILD_ID=0x1234;

```

When debugging this example and examining the memory, it can be observed that:

- Address 0x807 f800 contains VERSION_NUMBER
- Address 0x807 f804 contains CRC_NUMBER
- Address 0x807 f808 contains BUILD_ID

Figure 83. Linker memory output



If the inserted data order in the Flash memory is important, map the order of the variables in the linker script. This makes it possible to define the variables in any file. The linker outputs the variables in the defined order independently on how the files are linked. As a result, if the CRC_NUMBER is calculated in some way after the linker has built the file, the CRC_NUMBER can be inserted into the Flash memory file by another tool:

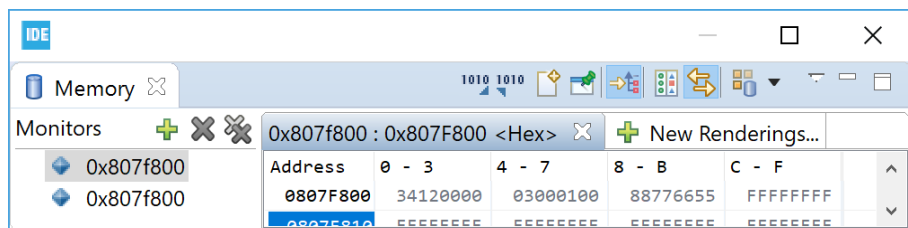
```
Decide the order in the linker script by adding the specially named sections in order
BUILD_ID, VERSION_NUMBER, CRC_NUMBER, and others(*).
```

```
.flash_v :
{
  *(.flash_v.BUILD_ID*);
  *(.flash_v.VERSION*);
  *(.flash_v.CRC*);
  *(.flash_v*);
} > FLASH_V
```

When debugging this example and examining the memory, it can be observed that:

- Address 0x807 f800 contains BUILD_ID
- Address 0x807 f804 contains VERSION_NUMBER
- Address 0x807 f808 contains CRC_NUMBER

Figure 84. Linker memory output specified order



2.5.7.4 Linking in a block of binary data

It is possible to link in a block of binary data into the linked file. The example below describes how to include a . . /readme.txt file.

Example:

```
File: readme.txt
Revision: Version 2
Product news: This release ...
```

One way to include this in the project is to make a reference in a C file to include it using the `incbin` directive and the `allocatable ("a")` option on the section:

```
asm(".section .binary_data,\"a\";"
".incbin \"../readme.txt\";"
);
```

The new section `binary_data` is then added into the linker script with instructions that the section must be put in the Flash memory. The `KEEP()` keyword can be used to surround an input section so that the linker garbage collector does not eliminate the section even if not called:

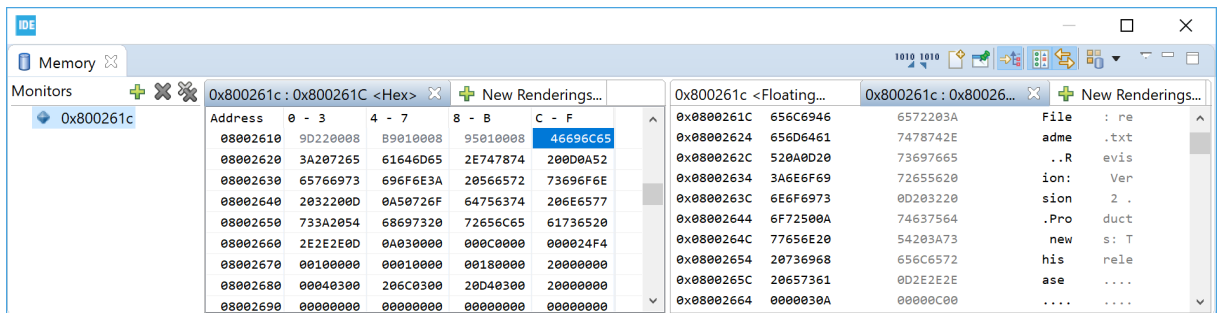
```
.binary_data :
{
  _binary_data_start = .;
  KEEP(*(.binary_data));
  _binary_data_end = .;
} > FLASH
```

This block can then be accessed from the C code:

```
extern int _binary_data_start;
int main(void)
{
  /* USER CODE BEGIN 1 */
  int *bin_area = &_binary_data_start;
```

The binary data, in this case the `readme` file, can be observed in the *Memory* view when the project is debugged.

Figure 85. Linker memory displaying file `readme`



2.5.7.5 Locate uninitialized data in memory (NOLOAD)

There is sometimes a need to have variables located into the Flash, or some other non-volatile memory, which must not be initialized at startup. In such cases, it is possible to create a specific `MEMORY AREA` in the linker script (`FLASH_D`) and use the `NOLOAD` directive in the section using the area.

Example:

```
The MEMORY AREA can be defined like this

/* Memories definition */
MEMORY
{
  RAM      (xrw)      : ORIGIN = 0x20000000,   LENGTH = 96K
  FLASH    (rx)       : ORIGIN = 0x80000000,   LENGTH = 512K-4K
  FLASH_D  (rx)       : ORIGIN = 0x807F000,    LENGTH = 2K
  FLASH_V  (rx)       : ORIGIN = 0x807F800,    LENGTH = 2K
}
```

Add a section for `FLASH_D` using the `NOLOAD` directive. This can be done using the following code a bit further down in the linker script:

```
Place the following a bit further down in the script

.flash_d (NOLOAD) :
{
  *(.flash_d*);
} > FLASH_D
```

Finally, data can be used somewhere in the program by adding a section attribute when declaring the variables that must be located in the `FLASH_D` memory.

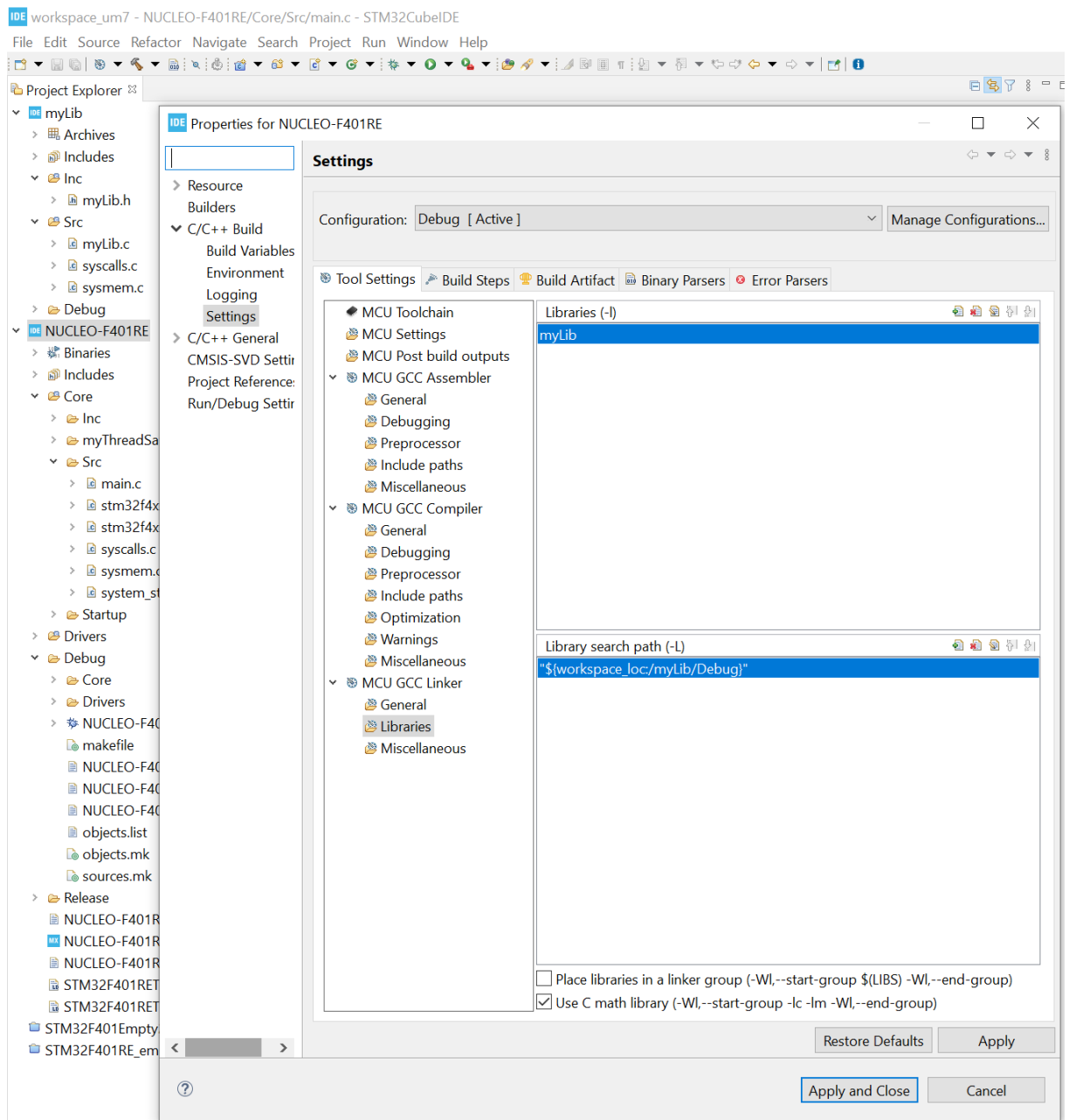
```
__attribute__((section(".flash_d"))) uint32_t Distance;
__attribute__((section(".flash_d"))) uint32_t Seconds;
```

2.5.8 Include libraries

To include a library into a project:

1. Right-click the project where the library must be included in the *Project Explorer* view and select **[Properties]**
2. In the dialog, select **[C/C++ Build]>[Settings]**
3. Select the *Tool Settings* tab in the panel
4. Select **[C Linker]>[Libraries]**
5. Add the library name to the **[Libraries]** field.
 Make sure the library name is added and not the path. According to the GCC convention, the library name is its filename without the *“lib”* prefix and *“.a”* extension.
 Example: for a library file named `libmyLib.a`, add the library name `myLib`.
 If by any chance the library name do not comply with the GCC convention, the full library name can be entered, preceded by a colon *“:”*.
 Example: for a library file named `STemWin524b_CM4_GCC.a`, add the library name `:STemWin524b_CM4_GCC.a`.
6. In the **[Library Paths]** list, set the library location path. Do not include the name of the library in the path.
 Example: `${workspace_loc:/myLib/Debug}` is the path to the archive file of the library project `myLib` residing in the same workspace as the application project.
7. Enable **[Place libraries in a linker group (-WI,--start-group \$(LIBS) -WI,--end-group)]** if libraries need to be linked several times to resolve circular dependencies.

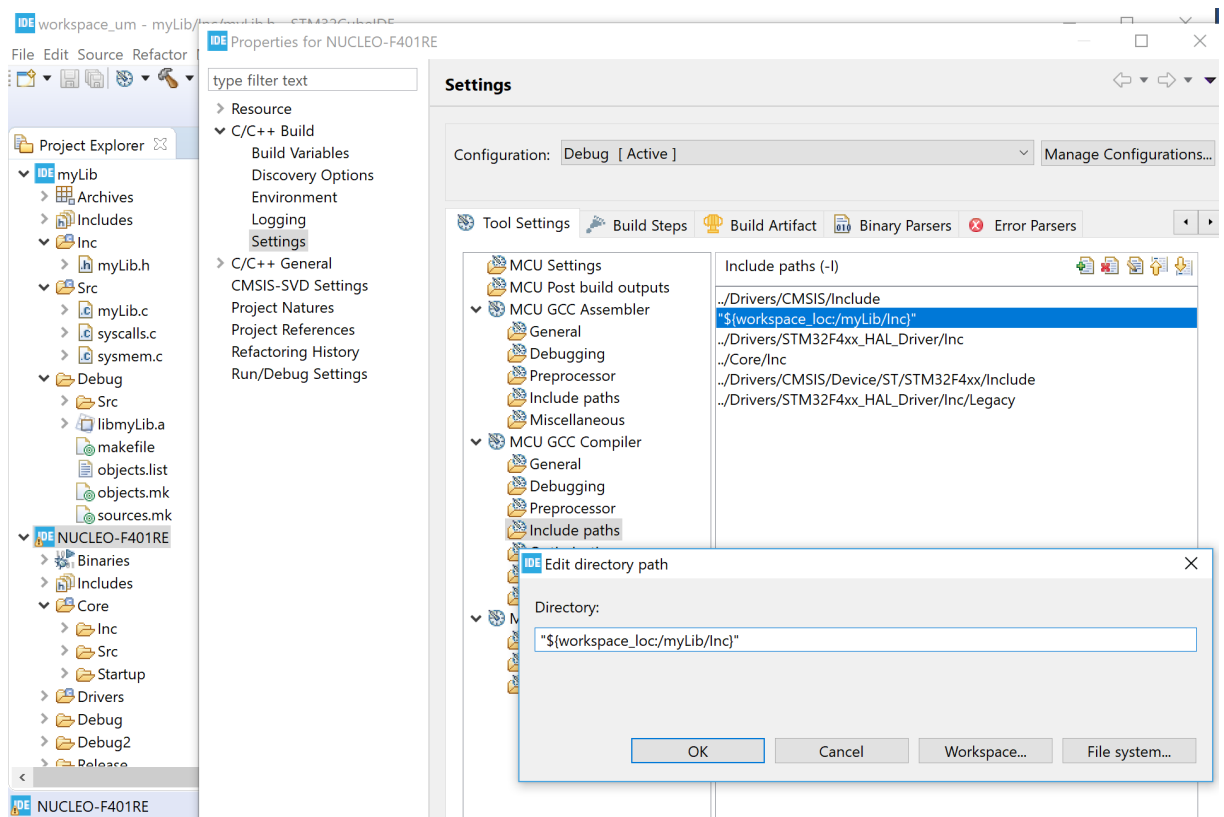
Figure 86. Include a library



The source folders for the header files must also be added to the [Include paths] field:

1. Select [MCU GCC Compiler]>[Include paths]
2. press the [Add...] button and add the paths to the source folders for the header files in the library

Figure 87. Add library header files to the include paths



Note: Libraries added by include paths are considered as static libraries because they are provided by external parties. The header files are not rescanned as the content must not have changed for external header files. If external libraries must be treated as normal source folders, the folders must also be added as source folders to the project.

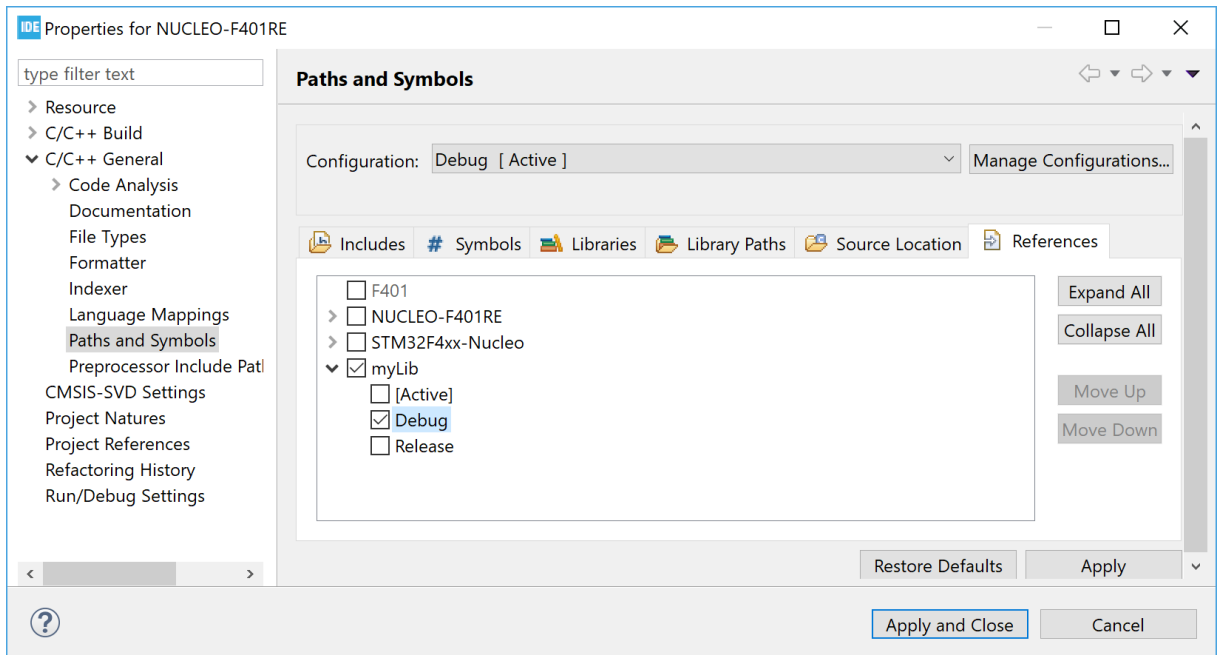
Refer to [Section 2.5.9 Referring to projects](#) for more information if a project is referring to another project, a library or a normal project.

2.5.9 Referring to projects

Whenever a project is using code from another project, both projects must be referring to each other.

For a project to refer to a specific build of another project:

1. Select instead **[Project]>[Properties]**
2. Select **[C/C++ General]>[Paths and Symbols]**
3. Open the *References* tab
4. select the **[Configuration]** that the current project is referring to

Figure 88. Set project references


Note: When multiple projects are used as references, use the **[Move Up]** and **[Move Down]** buttons to setup the priorities.

There are many advantages to set project references correctly:

- The projects involved are not rebuilt more than necessary.
- The indexer is able to find functions from the library and open them. To use this possibility, press the **Ctrl** key and, in the editor, click the library function where it is used to open the library source file in the editor.
- It is possible to create the call hierarchy for the functions in the library. To find the call hierarchy, mark the function name and press **Ctrl+Alt+H** to display the call hierarchy in the *Call Hierarchy* view.

If a library project is added as a reference, all the correct settings in the *Paths and Symbols* property page for the library is set. The tool settings that depend on this property page are adjusted also.

This is the recommended method of adding libraries developed locally. For more information about adding libraries, refer to [Section 2.5.8 Include libraries](#).

Another way to have projects referring to each other is as follows:

1. Select **[Project]>[Properties]**
2. Select **[Project References]**
3. Select and mark the project for reference

With this method, however, it is not possible to refer to different build configurations and libraries are not set up automatically.

2.6 I/O redirection

The C run-time library contains many functions, including some to handle I/Os. The I/O-related run-time functions include `printf()`, `fopen()`, `fclose()`, and many others. It is common practice to redirect the I/O from these functions to the actual embedded platform. For instance, the `printf()` output can be redirected to an LCD display or serial cable while file operations like `fopen()` and `fclose()` can be redirected to a Flash memory file system middleware.

2.6.1 printf() redirection

There are several ways to perform `printf()` redirection, such as using UART or SWV/ITM. Another solution is the Real-Time Transfer technology (RTT) provided by SEGGER.

The three techniques compare as follows:

- The UART output is maybe the most commonly used method, where the output from the embedded system is sent for instance to a terminal using RS-232. It requires some CPU overhead and medium bandwidth.
- The Instrumentation Trace Macrocell (ITM) output is efficient but requires that the Arm® CoreSight™ debugger technology with Serial Wire Viewer (SWV) is supported by the device. This is normally the case for Cortex®-M3, Cortex®-M4, Cortex®-M7, and Cortex®-M33 based devices. However, the SWV signals must be available and connected to the board also. It requires low CPU overhead but limited bandwidth. ITM output is explained in [Section 4 Debug with Serial Wire Viewer tracing \(SWV\)](#).
- The RTT solution is described by SEGGER on their website. RTT is a fast solution but requires SEGGER J-LINK debug probe.

To enable I/O redirection with UART or ITM output, the file `syscalls.c` must be included and built into the project. When `printf()` is used, it calls the `_write()` function, which is implemented in `syscalls.c`.

The `syscalls.c` file is normally created and included in the project when creating a new STM32CubeIDE project. The `_write()` function in this file must be modified to enable `printf()` redirection by modifying the call to `__io_putchar()`. The way to modify `_write()` depends on the hardware and library implementation.

The example below shows how to update `syscalls.c` so that `printf` output is redirected to ITM with an STM32F4 Series device. This is done by adding some header files to access `ITM_SendChar()` and make a call to `ITM_SendChar()`.

```
Original _write() function

__attribute__((weak)) int _write(int file, char *ptr, int len)
{
    int DataIdx;

    for (DataIdx = 0; DataIdx < len; DataIdx++)
    {
        __io_putchar(*ptr++);
    }
    return len;
}

Modified with added header files calling ITM_SendChar(*ptr++);

#include "stm32f4xx.h"
#include "core_cm4.h"

__attribute__((weak)) int _write(int file, char *ptr, int len)
{
    int DataIdx;

    for (DataIdx = 0; DataIdx < len; DataIdx++)
    {
        //__io_putchar(*ptr++);
        ITM_SendChar(*ptr++);
    }
    return len;
}
```

It can be noticed that the `_write` function in `syscalls.c` contains a weak attribute. This means that the `_write` function can be implemented in any C file used by the project.

For instance, the new `_write()` function can be added directly into `main.c`. Omit the weak attribute in that case, as shown in the example below.

```
int _write(int file, char *ptr, int len)
{
    int DataIdx;

    for (DataIdx = 0; DataIdx < len; DataIdx++)
    {
        //__io_putchar(*ptr++);
        ITM_SendChar(*ptr++);
    }
    return len;
}
```

2.7 Thread-safe wizard for empty projects and CDT™ projects

STM32CubeIDE includes a thread-safe wizard to generate files to support the use of resources that can be updated by application code and interrupts or when using a real-time operating system.

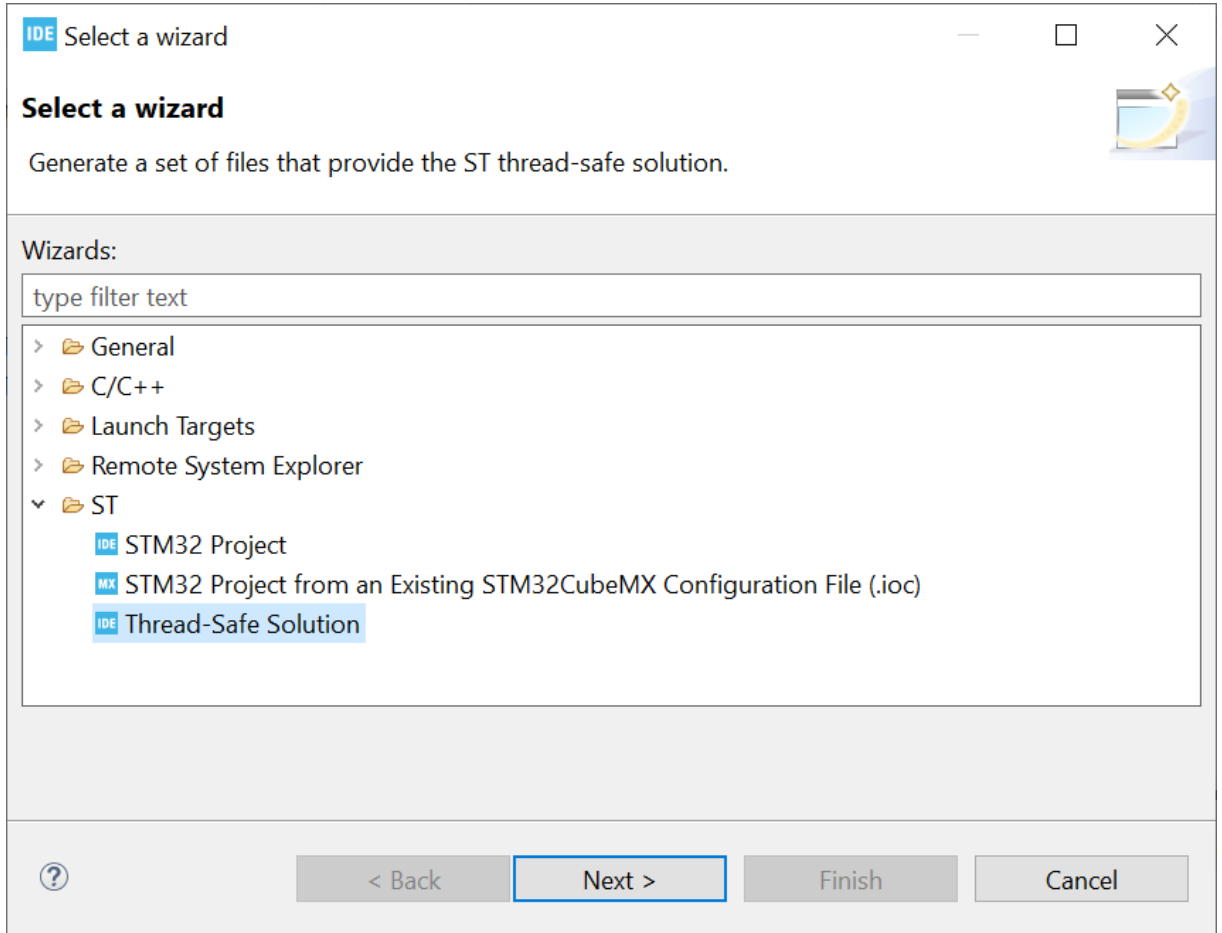
Note: *The thread-safe wizard may only be used for STM32CubeIDE empty projects. For projects managed by STM32CubeMX, the thread-safe implementation configuration must be made using STM32CubeMX dialogs.* The thread-safe wizard creates three files and adds the `STM32_THREAD_SAFE_STRATEGY` define to the project. The files are:

- `newlib_lock_glue.c`
- `stm32_lock_user.h`
- `stm32_lock.h`

First, in the example below, a `myThreadSafe` folder is created in the empty project. This folder is selected in the *Thread-Safe Solution* wizard so that files are generated in this folder.

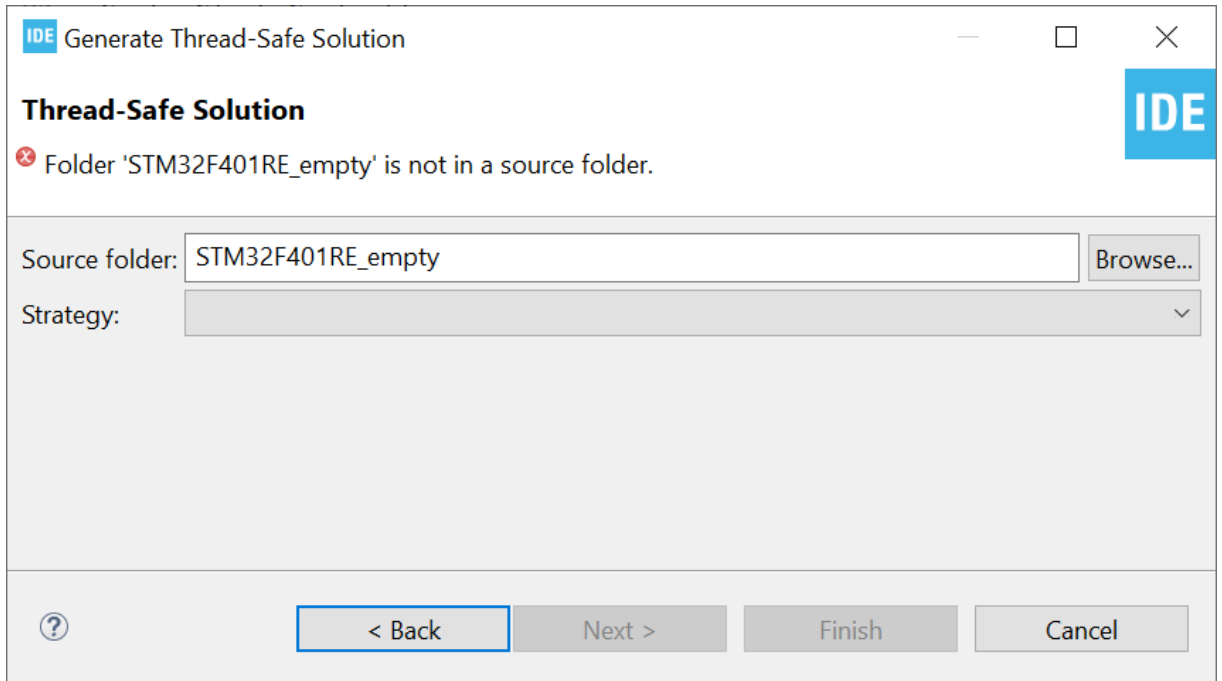
Open the menu [File]>[New]>[Other...] to obtain the wizard selection window shown in Figure 89.

Figure 89. Select a wizard



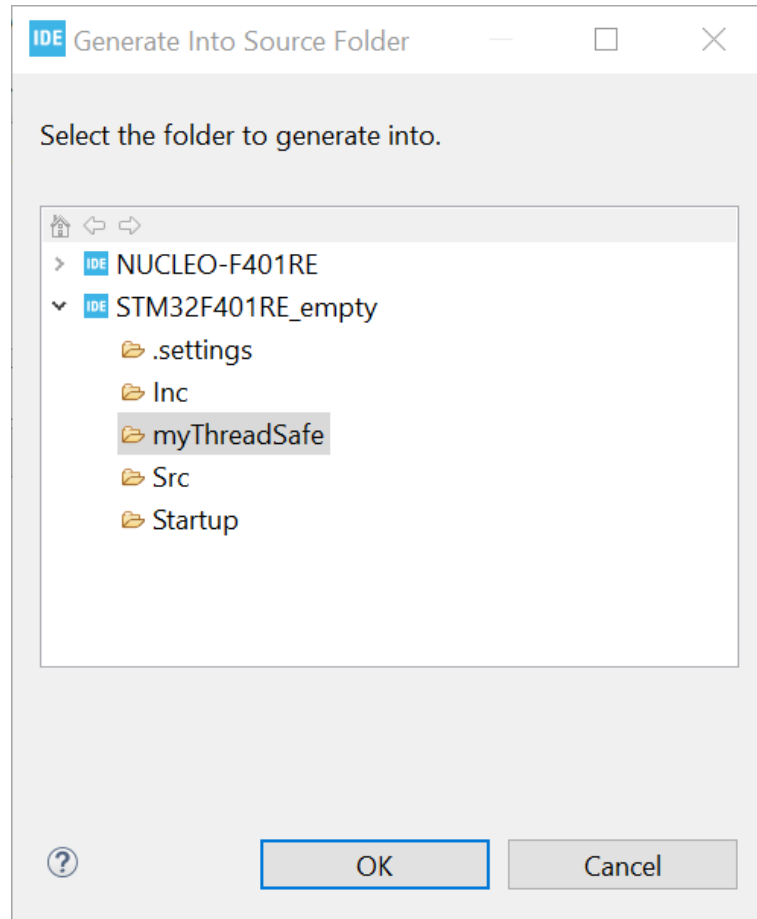
In the [ST] node select [Thread-Safe Solution] and press [Next >] to open the *Thread-Safe Solution* wizard.

Figure 90. Thread-Safe Solution wizard



Press **[Browse]** to open the *Generate Into Source Folder* dialog.

Figure 91. Thread-safe source folder location



Select the source folder to generate the files into and press **[OK]**.

The wizard proposes to select among five different thread-safe strategies:

1. User-defined thread-safe implementation.
2. Allow lock usage from interrupts.
3. Deny lock usage from interrupts.
4. Allow lock usage from interrupts. Implemented using FreeRTOS™ locks.
5. Deny lock usage from interrupts. Implemented using FreeRTOS™ locks.

The different strategies are explained in file `stm32_lock.h`.

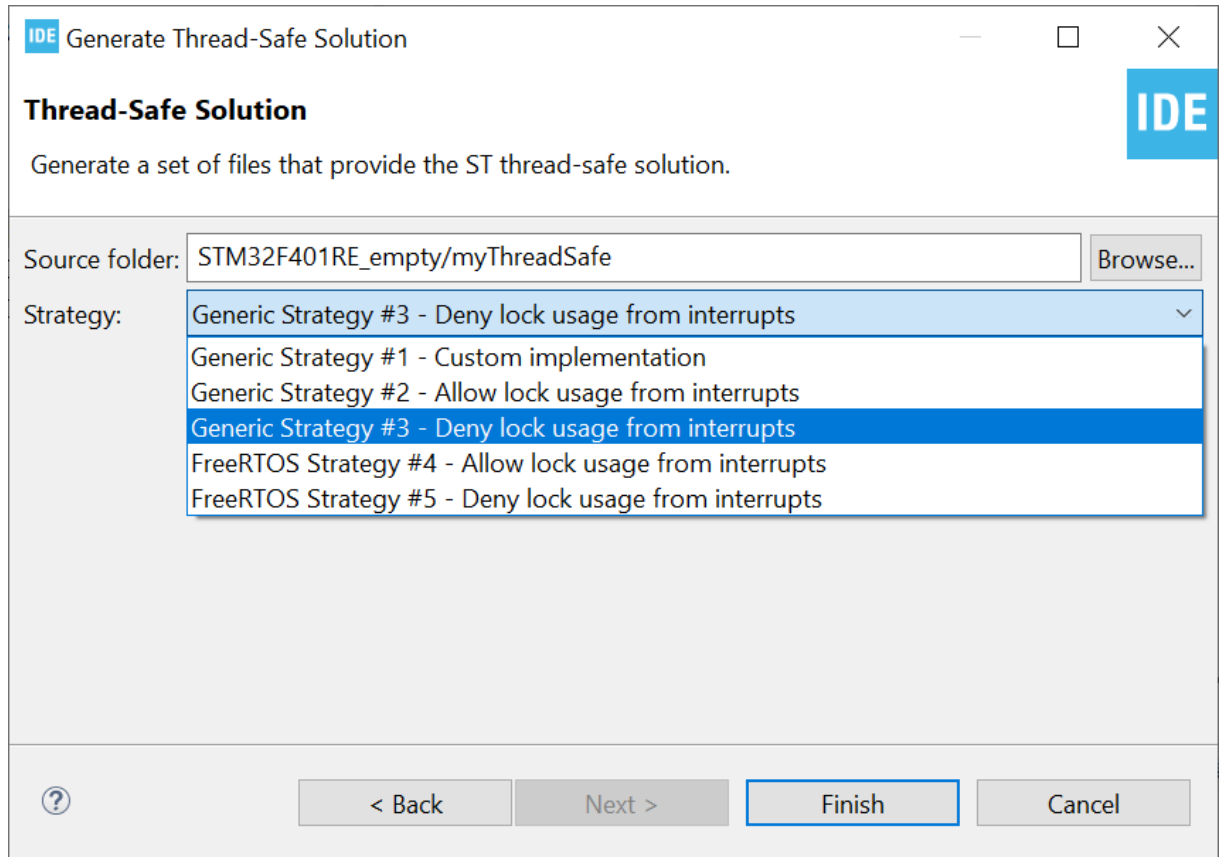
```

* 1. User defined thread-safe implementation.
*   User defined solution for handling thread-safety.
*   NOTE: The stubs in stm32_lock_user.h needs to be implemented to gain
*   thread-safety.
*
* 2. Allow lock usage from interrupts.
*   This implementation will ensure thread-safety by disabling all interrupts
*   during e.g. calls to malloc.
*   NOTE: Disabling all interrupts creates interrupt latency which
*   might not be desired for this application!
*
* 3. Deny lock usage from interrupts.
*   This implementation assumes single thread of execution.
*   Thread-safety dependent functions will enter an infinity loop
*   if used in interrupt context.
*
* 4. Allow lock usage from interrupts. Implemented using FreeRTOS locks.
*   This implementation will ensure thread-safety by entering RTOS ISR capable
*   critical sections during e.g. calls to malloc.
*   By default this implementation supports 2 levels of recursive locking.
*   Adding additional levels requires 4 bytes per lock per level of RAM.
*   NOTE: Interrupts with high priority are not disabled. This implies
*   that the lock is not thread-safe from high priority interrupts!
*
* 5. Deny lock usage from interrupts. Implemented using FreeRTOS locks.
*   This implementation will ensure thread-safety by suspending all tasks
*   during e.g. calls to malloc.
*   NOTE: Thread-safety dependent functions will enter an infinity loop
*   if used in interrupt context.

```

Select a strategy as shown in Figure 92.

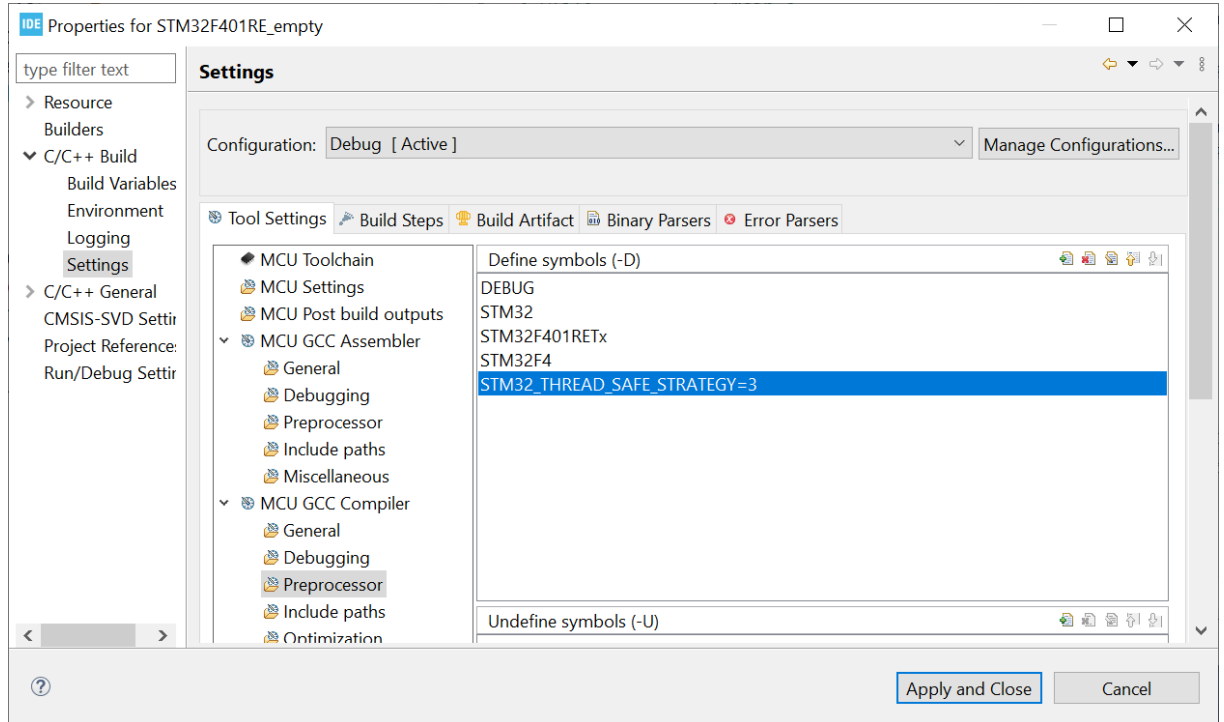
Figure 92. Thread-safe strategy selection



Select a strategy and press [**Finish**] to generate the files into the selected source folder.

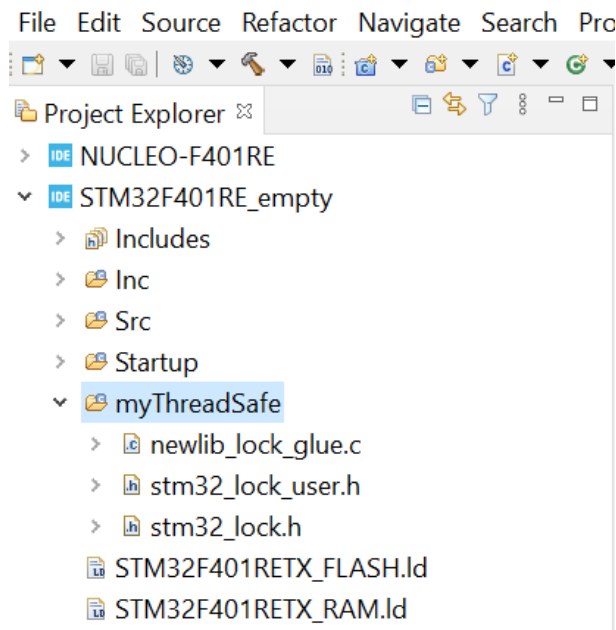
Note: The files generated are the same and contain the same information independently of the selected strategy.

A new define, `STM32_THREAD_SAFE_STRATEGY=3`, is added by the wizard to the project for use by the preprocessor when building the project. The define value is set according to the strategy selected in the wizard. The define can be observed by opening the project properties and looking into the *Tool Settings* tab.

Figure 93. Thread-safe properties


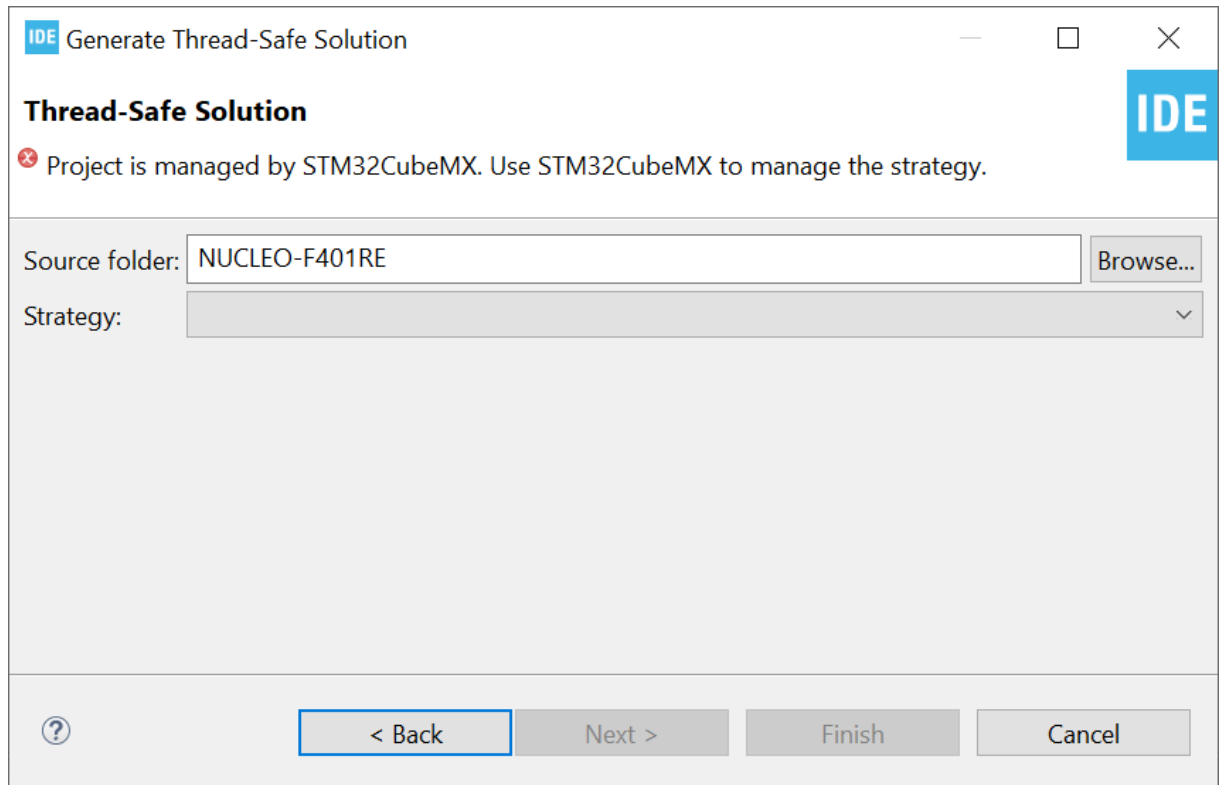
The generated files are shown in the *Project Explorer*.

Note: *The generated files require that some CMSIS header files are included in the project. The files must be copied and added manually into the project if they are missing.*

Figure 94. Thread-safe files


If the wizard is started while the project is managed by STM32CubeMX, an error is displayed stating that STM32CubeMX must be used to manage the thread-safe strategy.

Figure 95. Thread-safe error dialog



2.8 Position-independent code

This section is of interest to users working on applications where the final address location is not defined in the system. This occurs for instance when using a bootloader: the system designer must be able to define the final location of the application. In such case, position-independent code (PIC) can be used. The `-fPIE` compiler option enables the compiler/linker to generate position-independent executable.

Compiling with option `-fPIE` generates position-independent executable so that if the application is linked for address `0x800 0000` but placed at `0x800 1000`, it still runs.

However, the information in this section is not complete. The solution it describes works when using global data initialized to zero (`.bss`) but it does not work when using initialized data and has several other limitations. One such limitation is that run-time libraries included in the STM32 toolchain cannot be used as these libraries are built without the `-fPIE` option for optimization. Instead of using position-independent code in a system, it is worth considering other solutions.

Alternate solution example:

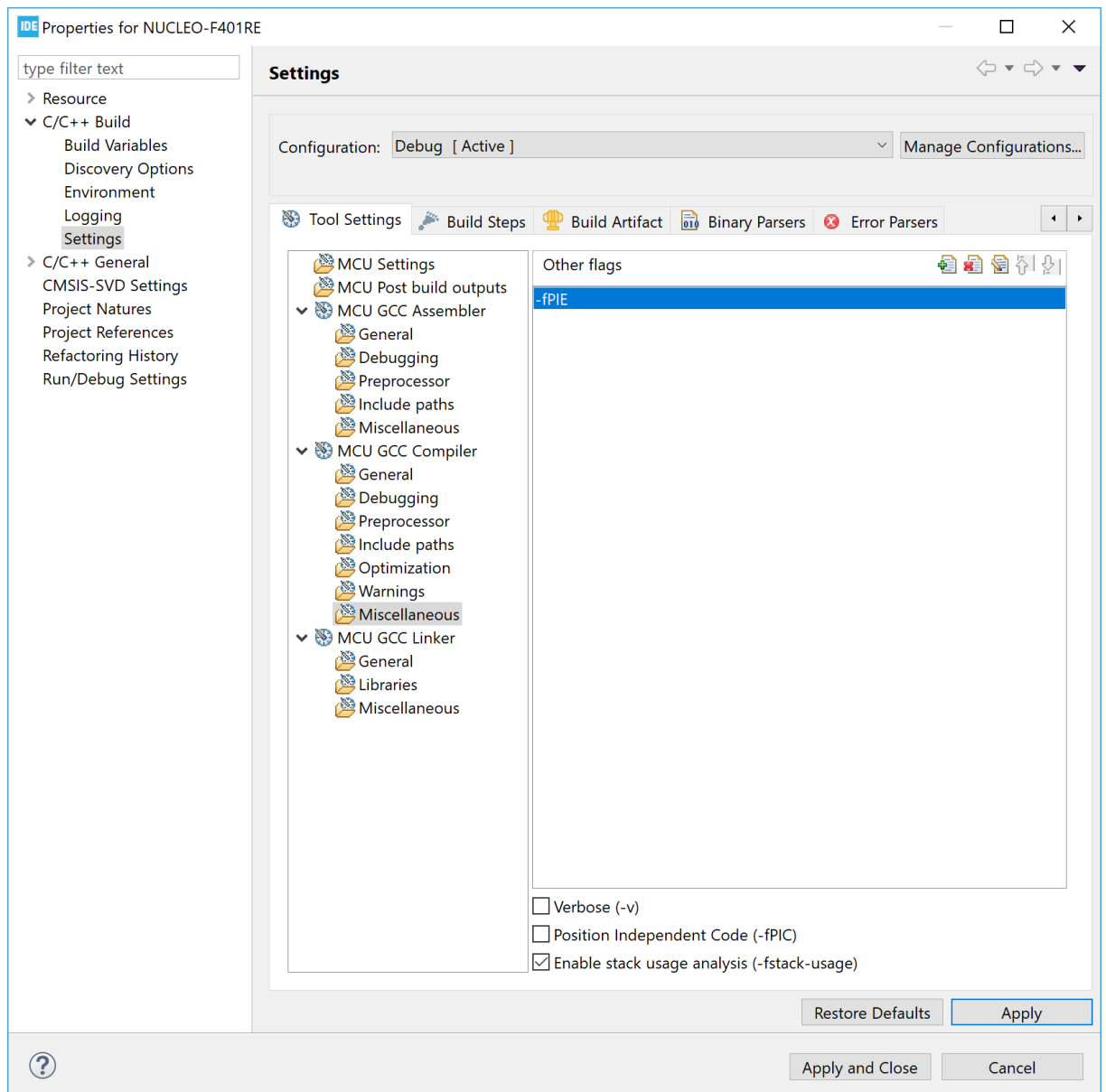
If a system is designed to have a bootloader and multiple versions of an application in Flash memory at different slots, it may be easier to setup multiple build configurations for the application. Each build configuration is based on its own linker script file. In this case, there is no need to use position-independent code since run-time libraries can be used. Each build configuration links the application to a unique slot in Flash memory, generating one single `elf` file per slot. When downloading a new version of the application into a slot, the correct `elf` file must be used. The bootloader can be designed to validate the addresses in the `elf` file and generate an error if it contains addresses that are out of the slot. The application can copy the interrupt vector table to RAM and update vector copies depending on the slot where the application is stored.

2.8.1 Adding the `-fPIE` option

To add the `-fPIE` option into the tool settings:

1. Right-click the project in the *Project Explorer* view and select **[Properties]**
2. In the dialog, select **[C/C++ Build]>[Settings]**
3. Select the *Tool Settings* tab in the panel
4. Select **[MCU GCC Compiler]>[Miscellaneous]**
5. Add `-fPIE` to the **[Other flags]** field.

Figure 96. Position independent code, `-fPIE`



2.8.2 Run-time library

The C run-time library is compiled without using the `-fPIE` option. So any call to the library must be avoided when generating position-independent executable. The startup code normally has a call to `__libc_init_array`. This call must be removed as in the example below:

```
/* Call static constructors */
/*   bl __libc_init_array */
```

2.8.3 Stack pointer configuration

Make sure that the stack pointer is set up correctly. The stack pointer must be set in the `Reset_Handler` in the startup file as shown in the example below. It must not be assumed that the stack pointer is set by a reset reading it from the vector table.

```
Reset_Handler:
    ldr    sp, =_estack          /* set stack pointer */
```

2.8.4 Interrupt vector table

The vectors in the vector table must be updated if the program is loaded to an offset address. If a program needs to add the offset to each vector in the table, it can copy the interrupt vector table to the RAM and add the offset to this vector table.

The vector base register must also be changed so that it points to the new located vector table as shown in the example below:

```
/* Set Vector Base Address */
SCB->VTOR=RAM_VectorTable;
```

2.8.5 Global offset table

The global offset table (GOT) is a table of addresses normally stored in the data section when building and using the `-fPIE` option. It is used by the executed program to find, during run-time, addresses of global variables, unknown at compile time. If no global variable location change is needed, the variables can be located at same place as located when linking the program. Then the GOT table can be placed in the `.text` section in the Flash memory area instead.

The example below shows how to update the linker script with the `.got*` section. In this case the `GOT_START` and `GOT_END` symbols are added also so that the tools are able to know the GOT location and size.

```
/* The program code and other data into "ROM" Rom type memory */
.text :
{
    . = ALIGN(4);
    *(.text)          /* .text sections (code) */
    *(.text*)        /* .text* sections (code) */
    GOT_START = .;
    *(.got*)
    GOT_END = .;
    *(.glue_7)       /* glue arm to thumb code */
    *(.glue_7t)      /* glue thumb to arm code */
    *(.eh_frame)

    KEEP (*( .init))
    KEEP (*( .fini))

    . = ALIGN(4);
    _etext = .;      /* define a global symbols at end of code */
} >ROM
```

2.8.6 Interrupt vector table and symbols

When debugging the code with an offset, both the load offset and the new symbol address must be specified. The symbol address to provide is the `.text` section address. The linker script can be updated by defining `.isr_vector` to be located into `.text`. This avoids the issue of finding the location of `.text`.

Remove the following

```
.isr_vector :
{
  . = ALIGN(4);
  KEEP(*(.isr_vector)) /* Startup code */
  . = ALIGN(4);
} >FLASH
```

Add `KEEP(*(.isr_vector))` instead to first location of `.text`

`/* The program code and other data into "FLASH" Rom type memory */`

```
.text :
{
  . = ALIGN(4);
  KEEP(*(.isr_vector)) /* Startup code */
  *(.text) /* .text sections (code) */
  *(.text*) /* .text* sections (code) */
  GOT_START = .;
  *(.got*)
  GOT_END = .;
  *(.glue_7) /* glue arm to thumb code */
  *(.glue_7t) /* glue thumb to arm code */
  *(.eh_frame)

  KEEP (*(.init))
  KEEP (*(.fini))

  . = ALIGN(4);
  _etext = .; /* define a global symbols at end of code */
} >FLASH
```

2.8.7 Debugging position-independent code

When debugging position-independent code located at an offset, the download offset and new symbol address must be set.

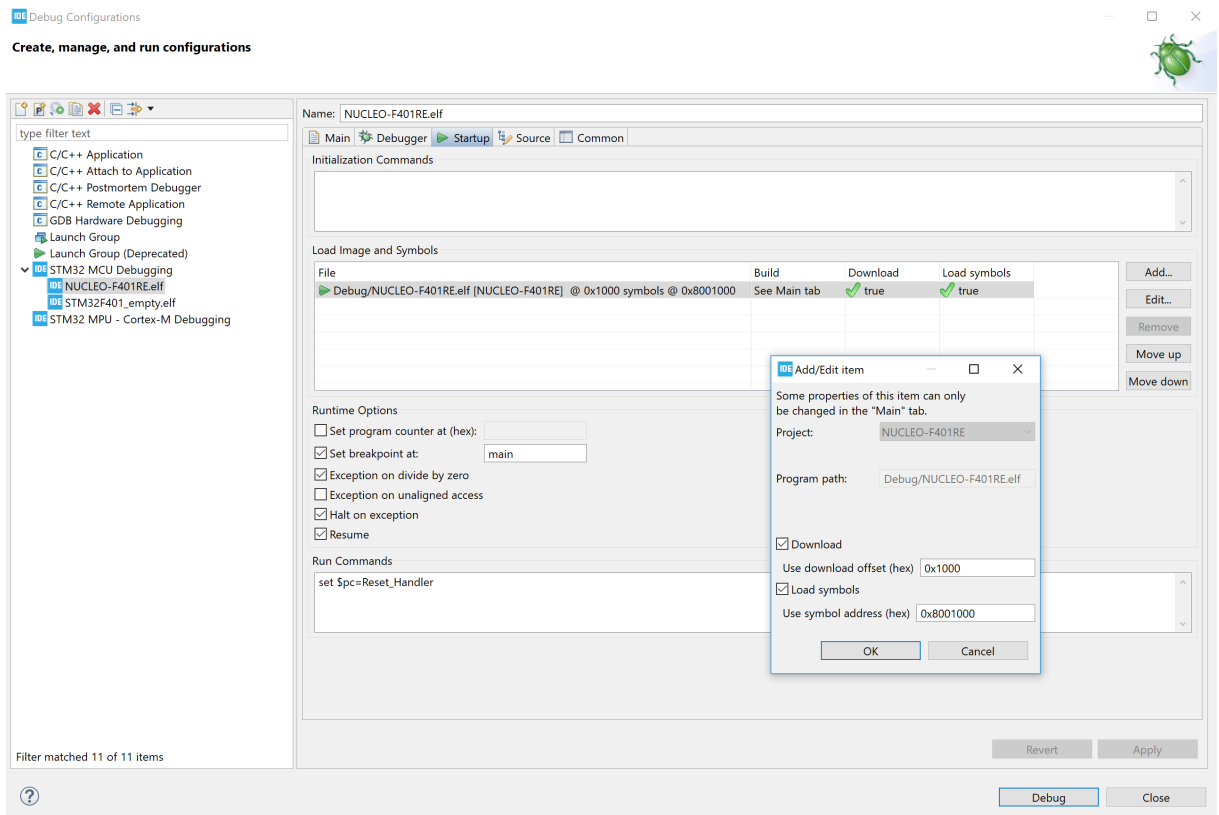
Figure 97. Debugging position independent code


Figure 97 illustrates an example where the download offset is 0x1000 and the symbol address is 0x800 1000. It is possible to set the symbol address to 0x800 1000 in this case because the `.isr_vector` is added into the `.text` section as described in Section 2.8.6 [Interrupt vector table and symbols](#).

If instead the `.isr_vector` is located in another section outside `.text`, the start address of the `.text` section must be used with the offset added. For instance, if the map file states that `.text` starts at 0x0000 0000 0800 0194, the symbol address in this case must be set to 0x800 1194.

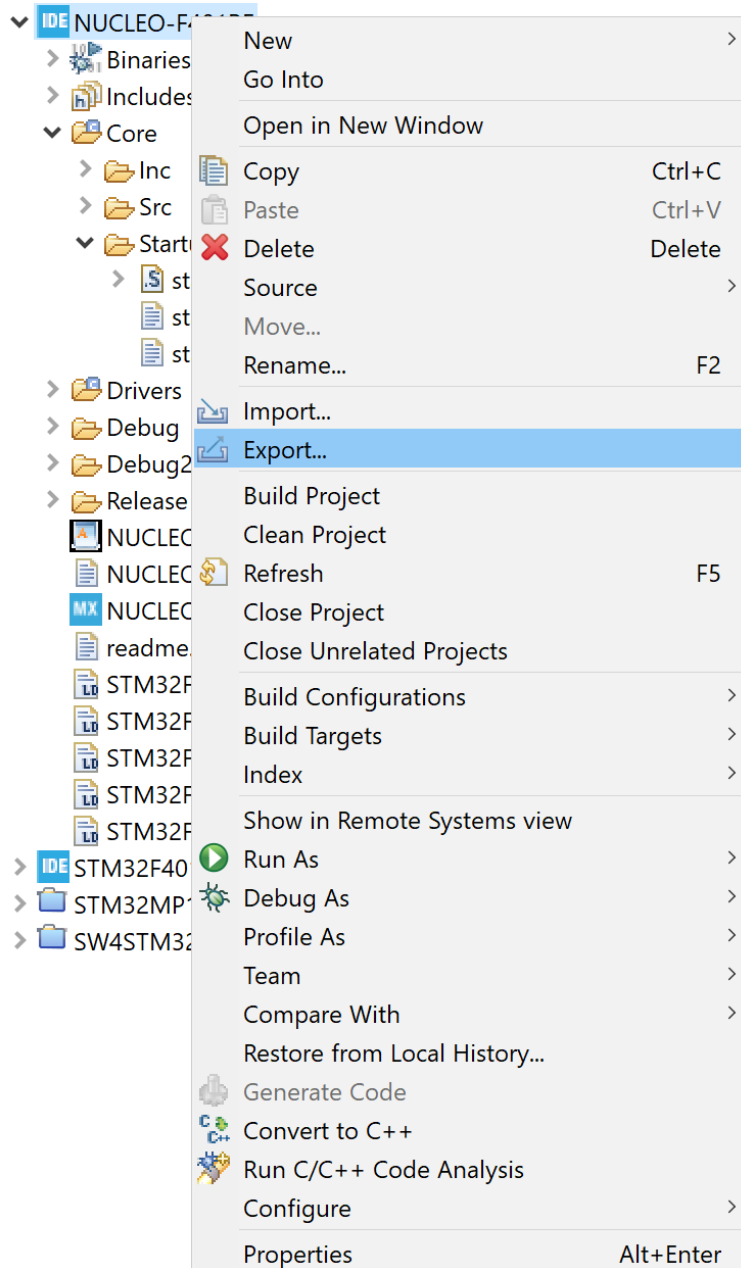
Figure 97 shows that the breakpoint is set at `main` and that the program counter (`$pc`) is set to the `Reset_Handler` symbol into [Run Commands]. This symbol contains the correct address to the `Reset_Handler` because `gdb` uses the base symbol address 0x800 1000. If `$pc` is not setup during this debug configuration, the [Resume] checkbox must be disabled to make the program stop after load. In this case, the program counter must be set manually in the *Registers* view before starting the program.

2.9 Exporting projects

A project can be exported in many different ways. This section shows how to export a project as a compressed zip file.

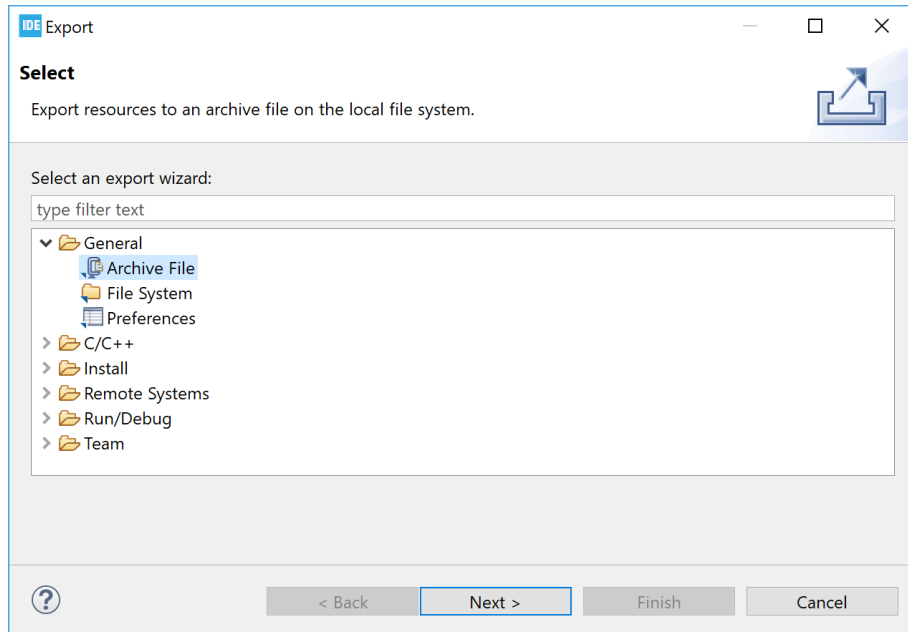
Right-click the project in the *Project Explorer* view and select [**Export...**].

Figure 98. Export project



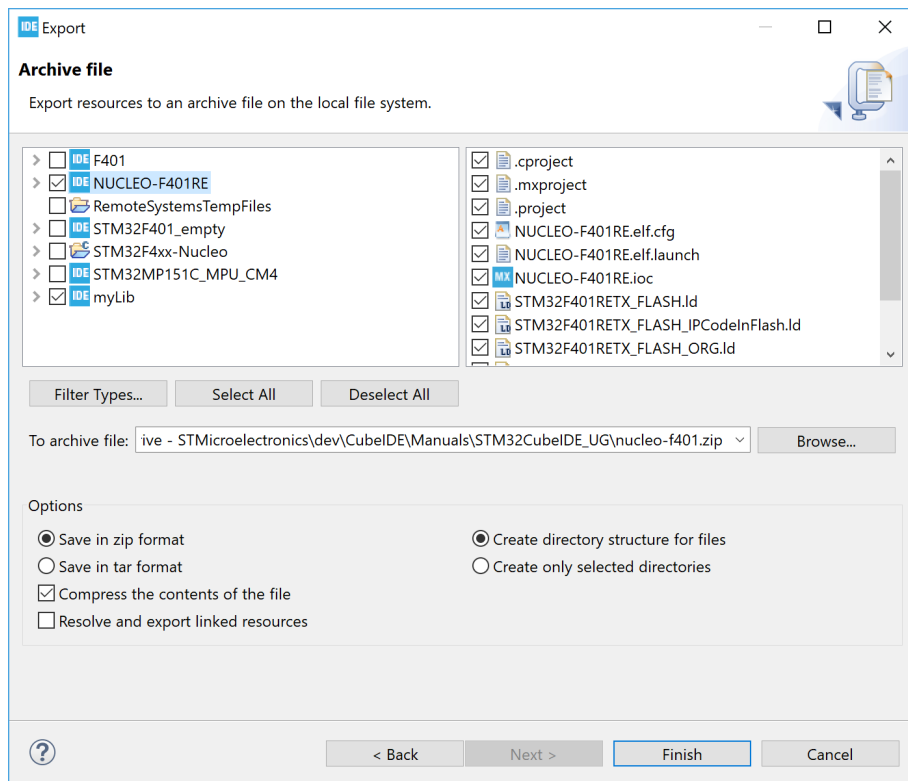
The *Export* dialog opens. Select **[General]>[Archive File]** and press **[Next >]**.

Figure 99. Export dialog



The *Export* dialog is updated. Select the project to be exported. It is possible to exclude some project files from the export. In the example in [Figure 100](#), all project and library files are included. A file name must be entered into the **[To archive file]** field, possibly browsing to a folder location for the file with the **[Browse...]** button. In the example, the default options values are kept unchanged. Press **[Finish]** to export the project and create the zip file.

Figure 100. Export archive



2.10 Importing existing projects

This section describes different ways to import existing projects into an STM32CubeIDE workspace. The standard Eclipse® importer is capable of importing Eclipse® projects. This is used to import projects created with STM32CubeIDE. The project importer is also extended to support the import of ac6 System Workbench for STM32 projects and Atollic® TrueSTUDIO® projects. Such projects are converted during the import phase to STM32CubeIDE projects.

It is possible to import and debug an existing `elf` file developed by another IDE or toolchain. More information on how this is done is available in [Section 3.8](#).

2.10.1 Importing an STM32CubeIDE project

A project can be imported in many different ways. This section shows how to import a project that was exported as a compressed zip file.

- One way to open the *Import* dialog is to use the menu **[File]>[Import...]**
- Another way is to right-click the *Project Explorer* view and select **[Import...]**

Figure 101. Import project

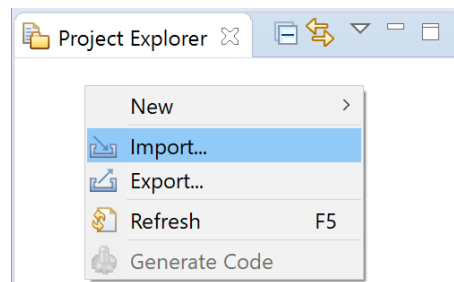


Figure 102. Import dialog

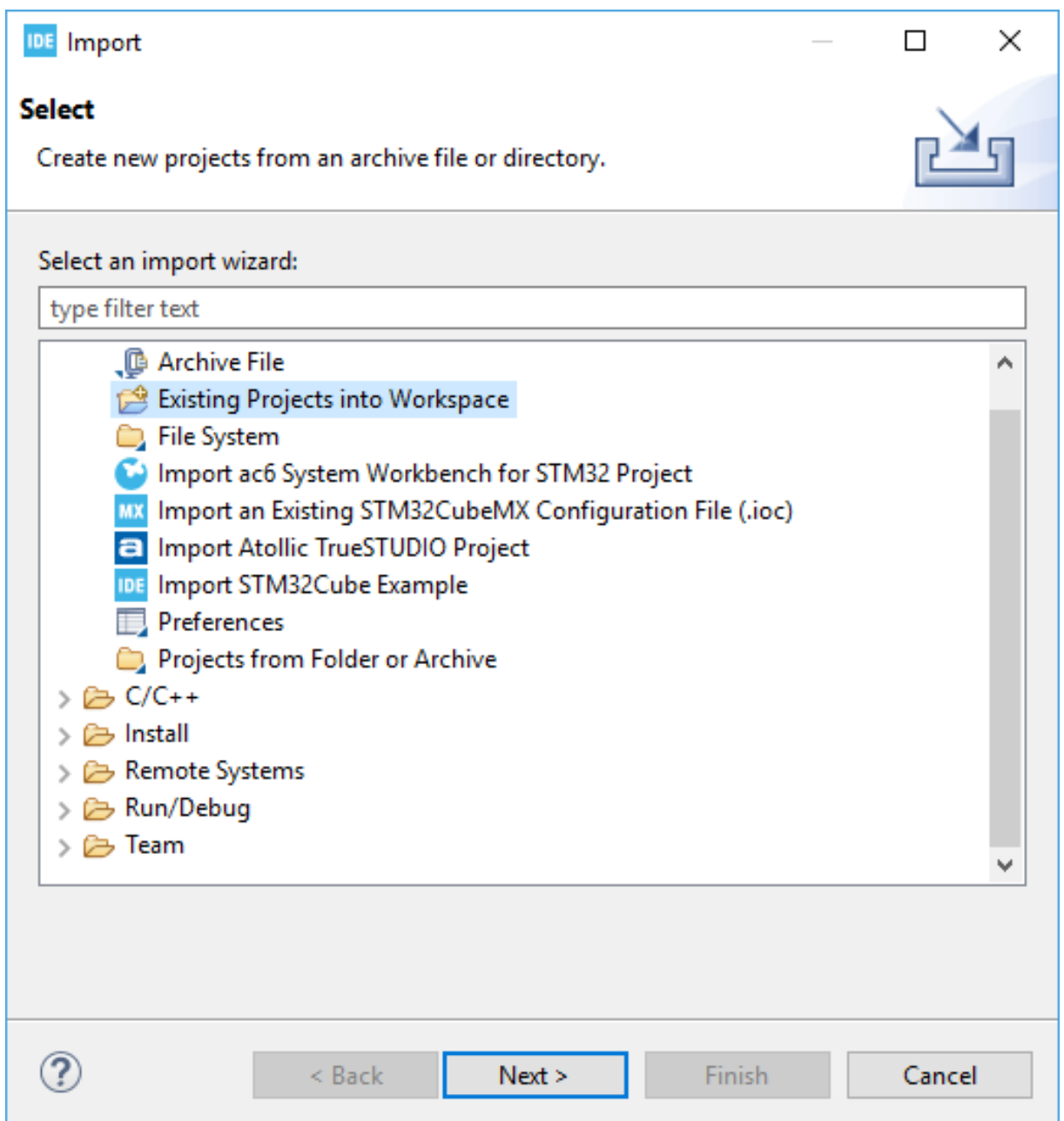
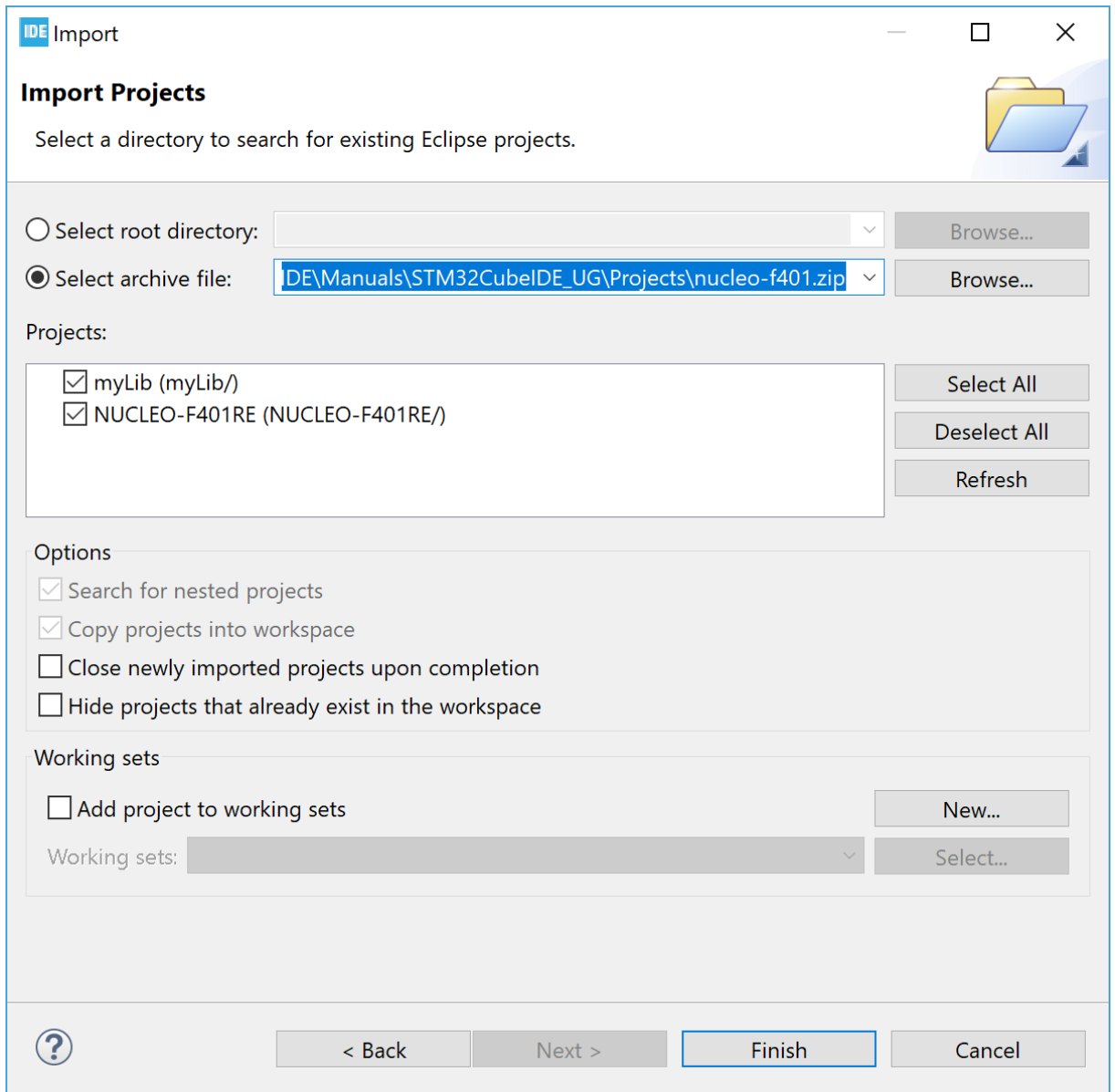


Figure 103. Import projects



2.10.2 Importing System Workbench and TrueSTUDIO® projects

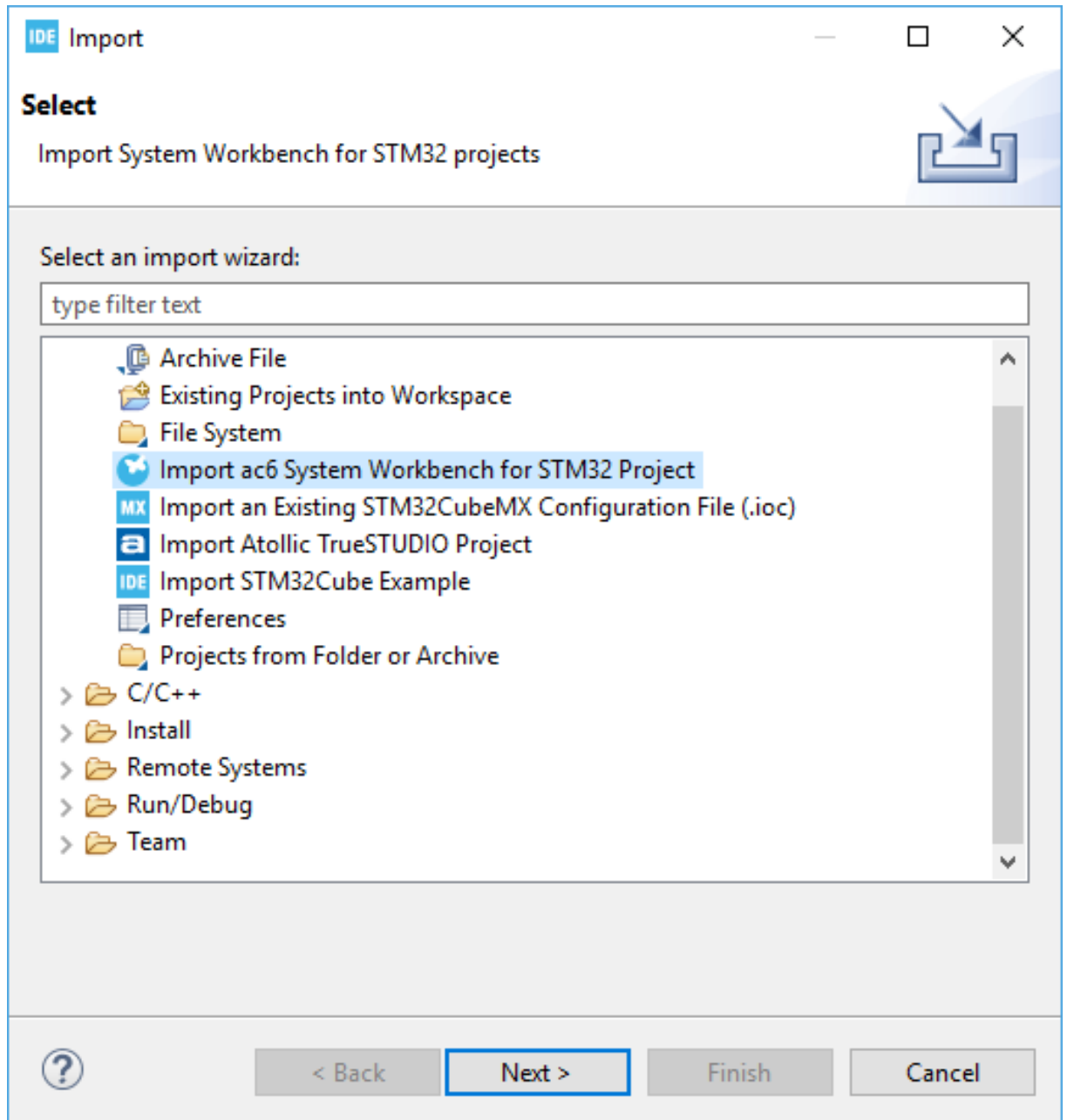
To import an ac6 System Workbench for STM32 project or an Atollic® TrueSTUDIO® project into STM32CubeIDE, it is advised to work on a project copy:

1. Create a copy of the project, either as a copy of the project folder or an export of the project in a zip file
2. Use the copied project for the import into STM32CubeIDE

The way to import the copied project is to open the *Import* dialog by means of the menu [File]>[Import...] or by right-clicking the *Project Explorer* view.

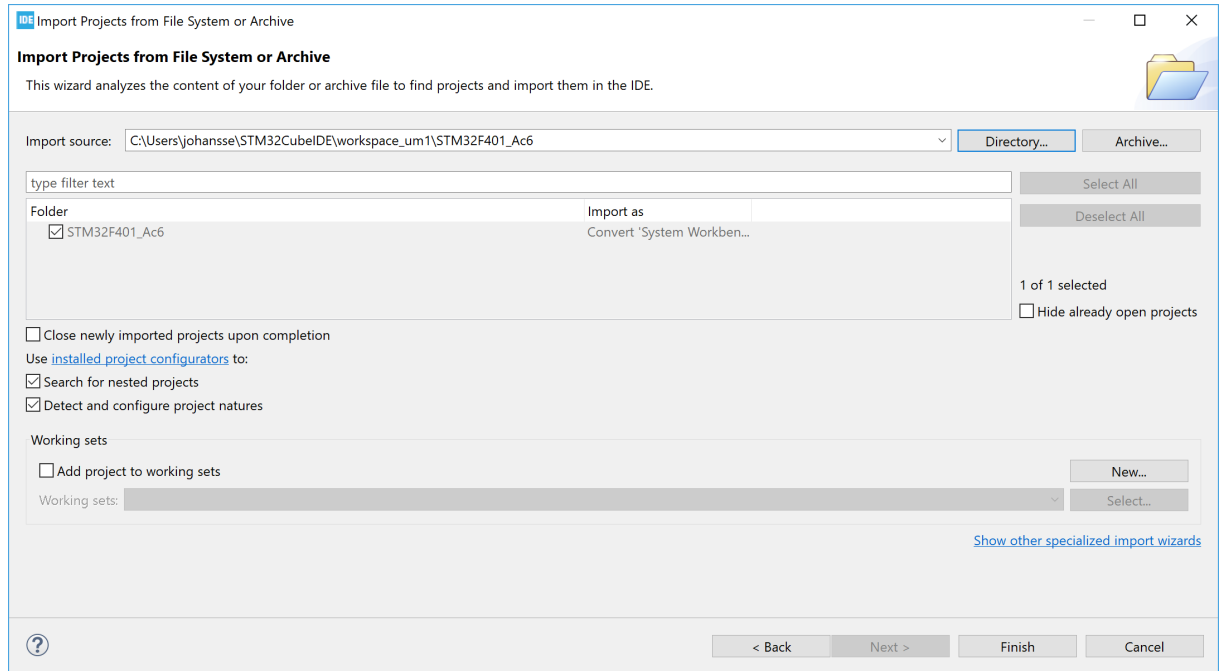
Select [**Import ac6 System Workbench for STM32 project**] or [**Import Atollic TrueSTUDIO project**] depending on the original tool used to create the project and press [**Next >**].

Figure 104. Import System Workbench projects (1 of 3)



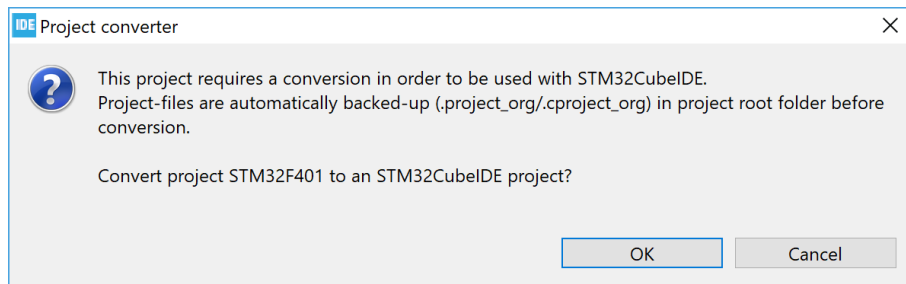
In this example, the ac6 project is copied into the STM32CubeIDE workspace, hence the **[Directory...]** button is used and project STM32F401_Ac6 is selected. The import wizard detects that this is a System Workbench project.

Figure 105. Import System Workbench projects (2 of 3)



Press **[Finish]** to open the *Project converter* dialog.

Figure 106. Import System Workbench projects (3 of 3)



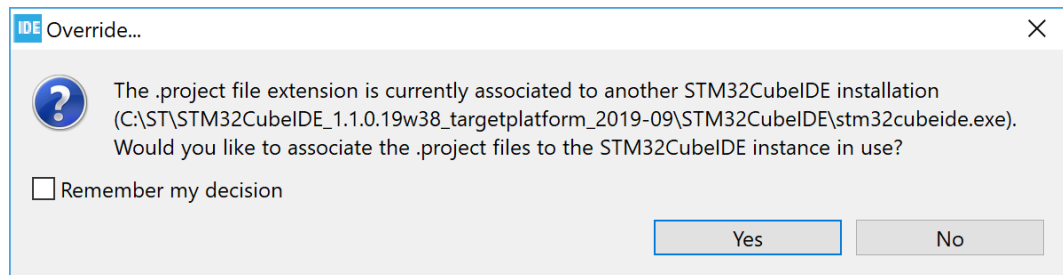
Press **[OK]** to convert the project to an STM32CubeIDE project.

There are two migration guides explaining how to migrate from ac6 System Workbench for STM32 ([ST-06]) and Atollic® TrueSTUDIO® to STM32CubeIDE ([ST-05]). These guides can be opened from the *Technical Documentation* page in the *Information Center*.

2.10.3 Importing using project files association

When STM32CubeIDE is started, a pop-up window asks if `.cproject` and `.project` files must be associated to the program.

Figure 107. Import using project files association



If the association is selected, double-clicking on a `.project` file in the personal computer file browser triggers the project import by STM32CubeIDE into the current workspace. The project converter investigates the project, which is imported directly if made for STM32CubeIDE. If the project comes from another tool, the project converter tries to identify if it is a known project format and, in such case, converts the project to an STM32CubeIDE project as described in [Section 2.10.2 Importing System Workbench and TrueSTUDIO projects](#).

2.10.4 Prevent “GCC not found in path” error

When importing old projects, an error in the *Problems* view can state “Program “gcc” not found in PATH”. The error is caused by the project use of a deprecated discovery method setting. The error can be removed by updating the *Window Preferences* and *Project Properties* settings.

1. Open **[Window]>[Preferences]**. In the *Preferences* dialog, select **[C/C++]>[Property Pages Settings]** and enable checkbox **[Display “Discovery Options” page]**.
2. Open **[Project Properties]>[C/C++ Build]>[Discovery Options]** and disable checkbox **[Automate discovery of paths and symbols]**.

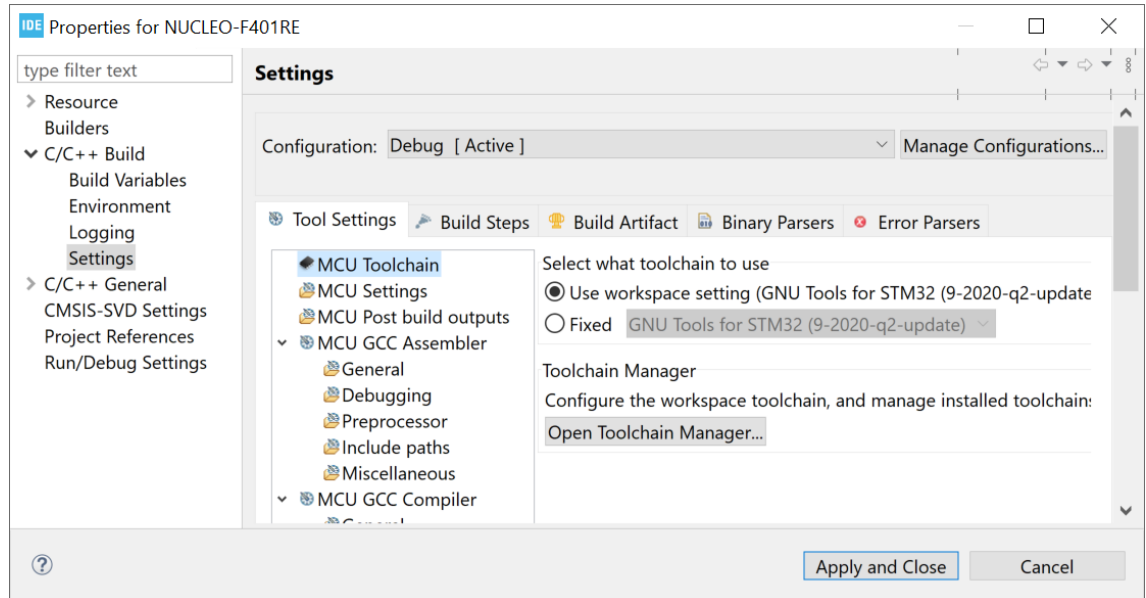
2.11 Toolchain Manager

The *Toolchain Manager* is used to install toolchains, uninstall toolchains and select the default workspace toolchain when building a project.

To open the *Toolchain Manager* from the *Tool Settings* tab in project properties:

1. Select the **[MCU Toolchain]** node

Figure 108. Open *Toolchain Manager*

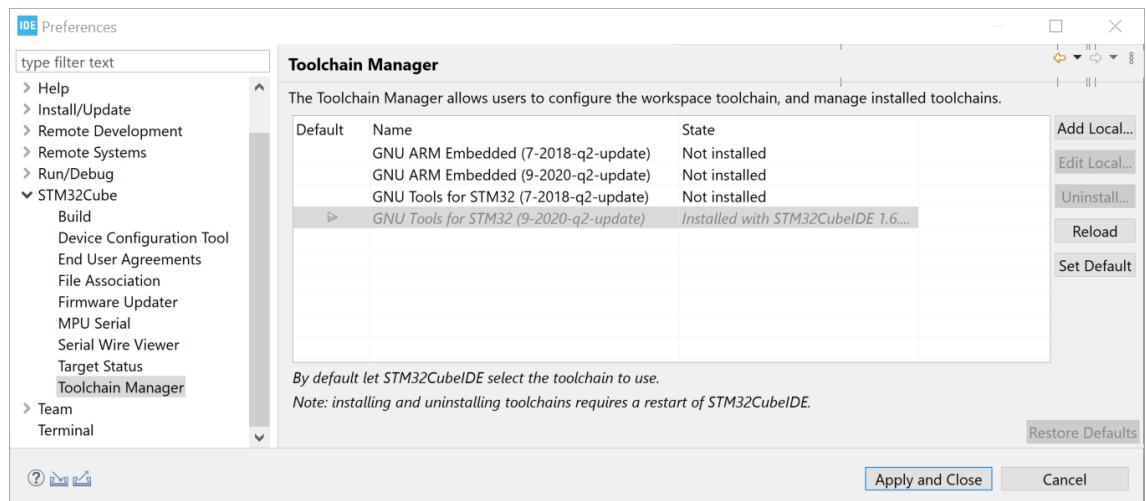


2. Click on **[Open Toolchain Manager...]**

The *Toolchain Manager* can also be opened from the **[Window]>[Preferences]** menu:

1. Select **[STM32Cube]>[Toolchain Manager]**

Figure 109. *Toolchain Manager*



The columns in the *Toolchain Manager* are described in [Table 4](#).

Table 4. *Toolchain Manager* column details

Name	Description
Default	A green/grey arrow symbol indicates the default workspace toolchain. The arrow color is: <ul style="list-style-type: none"> • green when the toolchain is manually set as default by the user • grey when the toolchain is selected as default by STM32CubeIDE logic
Name	The name of the toolchain.
State	The state of the toolchain. Toolchains available for download from STMicroelectronics online repository are listed as <i>“installed”</i> or <i>“not installed”</i> . Local toolchains added by the user are listed as <i>“local”</i> .

The buttons in the *Toolchain Manager* are described in [Table 5](#).

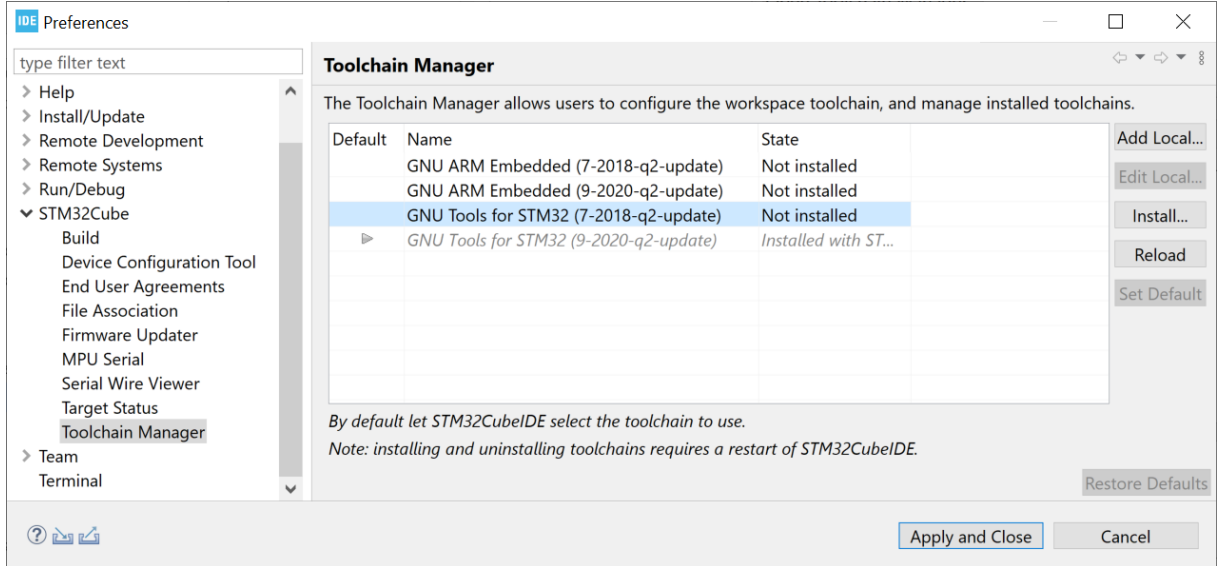
Table 5. *Toolchain Manager* button information

Name	Description
Add Local...	Add reference to local toolchain.
Edit Local...	Edit reference to local toolchain.
Install... Uninstall... Remove...	The button text depends on the type of the selected toolchain. It is used to: <ul style="list-style-type: none"> • Install / Uninstall the selected toolchain provided by the repository • Remove the selected local toolchain
Reload	Reload the toolchain list from the repository.
Set Default	Set selected toolchain to be used by default.
Restore Defaults	Restore and use the default toolchain.
Apply and Close	Apply selection and close dialog.
Cancel	Cancel dialog.

2.11.1 Install new toolchain

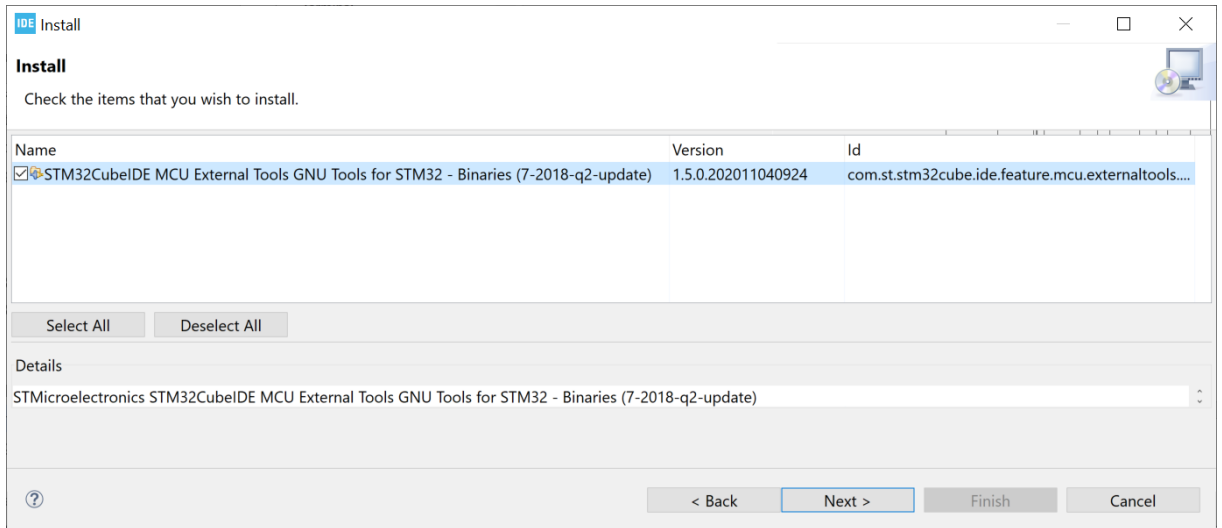
Open the *Toolchain Manager* to install a new toolchain.

Figure 110. Install toolchain



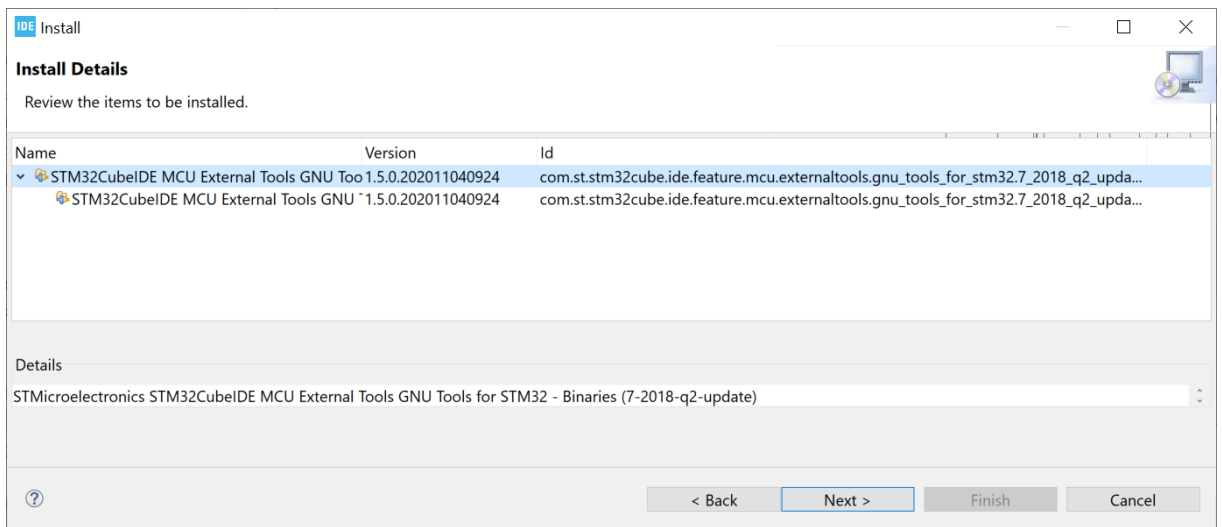
Select the toolchain to install and click on **[Install...]**. The *Install* dialog opens and displays the items to be installed.

Figure 111. Check items to install



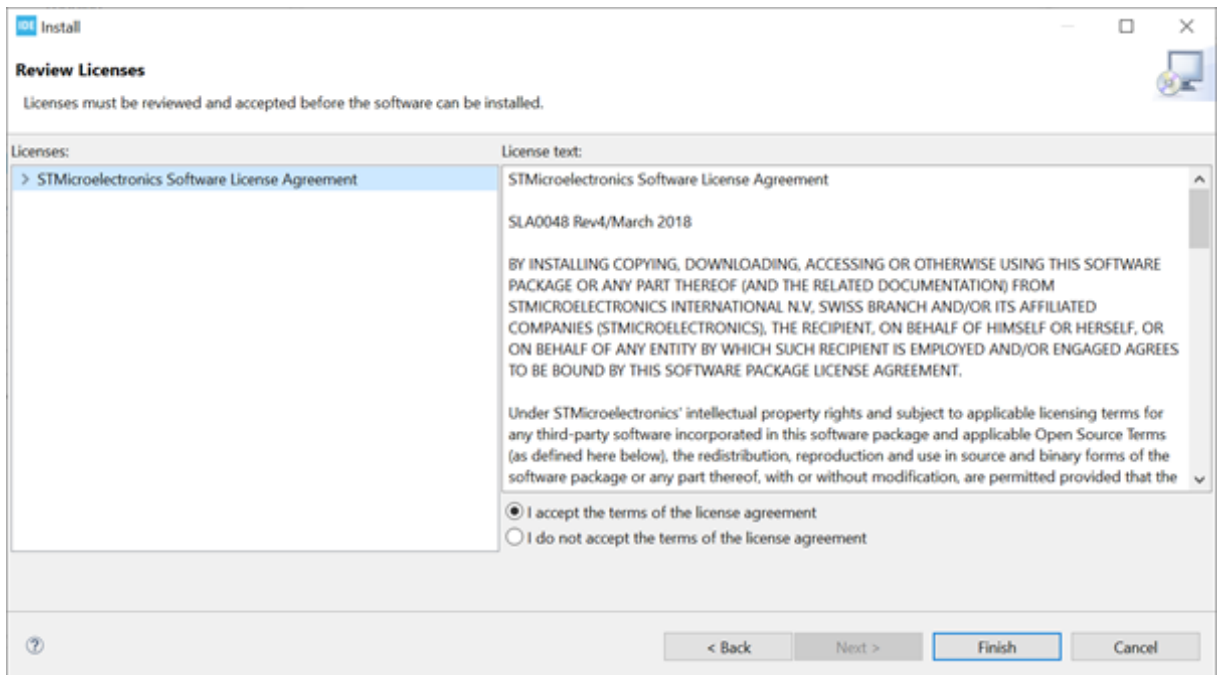
Check the items to install and click on **[Next]**.

Figure 112. Review items to install



Review the items and click on **[Next]**.

Figure 113. Review and accept licenses

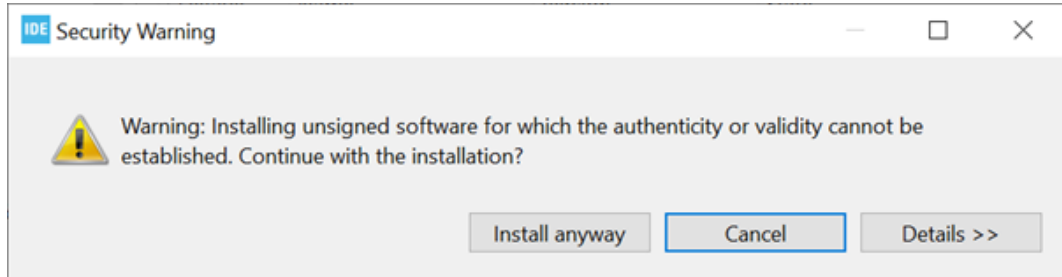


Review the licenses, select **[I accept the terms of the license agreements]** and click on **[Finish]**.

At this point, the software installation starts. The progress bar displayed at the bottom of the STM32CubeIDE window shows the installation completion rate. Wait until the installation is completed.

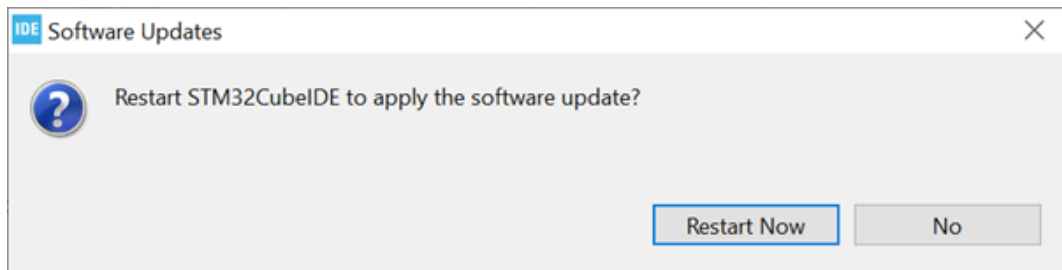
The following warning may appear before the installation is finished.

Figure 114. Security warning



In this case, to finalize the installation, click on **[Install anyway]**. After some time, the following dialog is displayed.

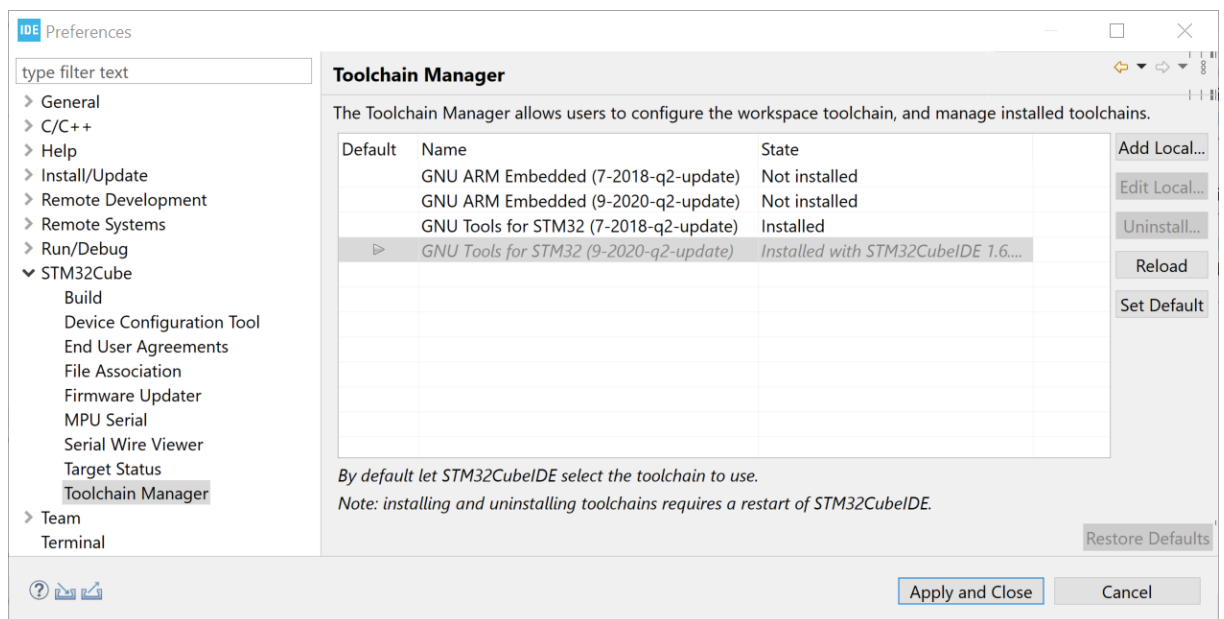
Figure 115. Restart to apply software update



Click on **[Restart Now]** to be able to use the installed toolchain in STM32CubeIDE. STM32CubeIDE is restarted and the new toolchain can be used.

Open the *Toolchain Manager* to verify the installation.

Figure 116. Toolchain installed



In this case, Figure 116 shows that two versions of *GNU Tools for STM32* are installed.

2.11.2 Manage default toolchain

The *Toolchain Manager* highlights the default workspace toolchain with an arrow in the *Default* column.

Figure 117. Default toolchain

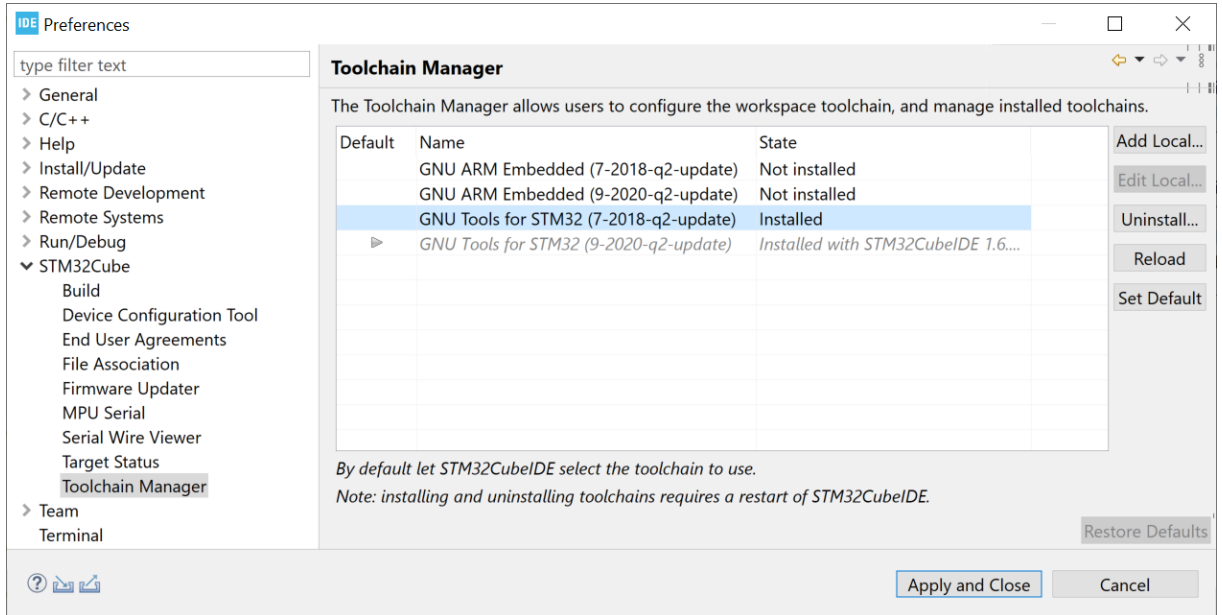
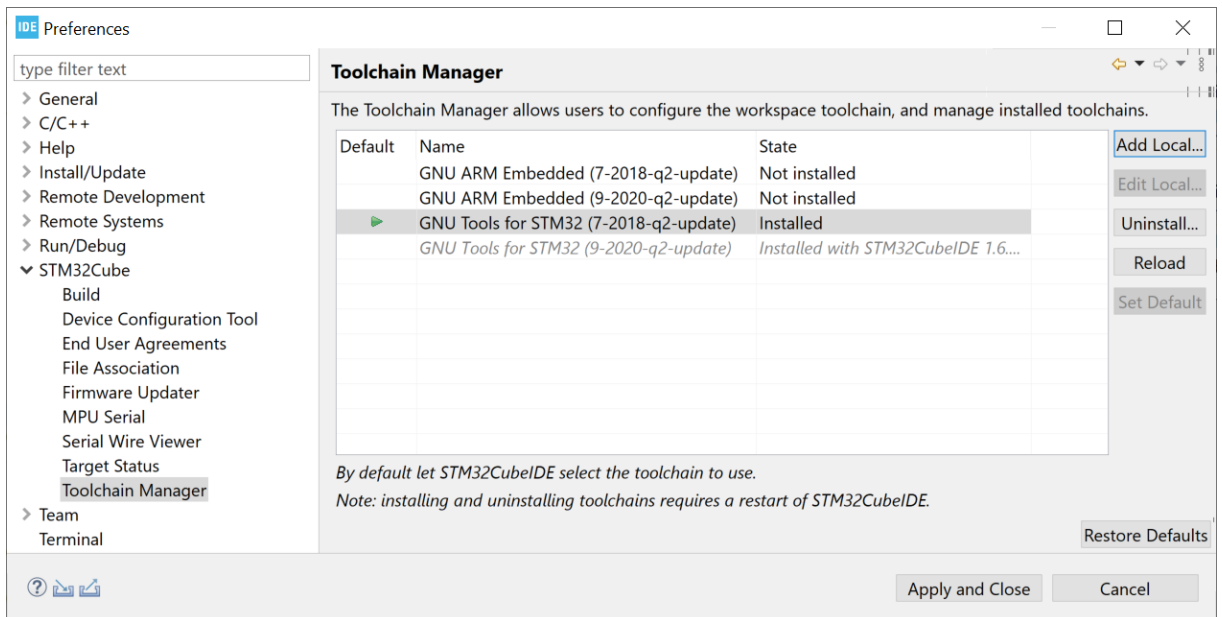


Figure 117 shows that *GNU Tools for STM32 version 9-2020-q2-update* is the default workspace toolchain. The *GNU Tools for STM32 version 7-2018-q2-update* line is marked in blue, which indicates that this toolchain selected. Any line in the table can be selected with the mouse.

Click on **[Set default]**: the selected toolchain to be used as the default workspace toolchain is highlighted with an arrow symbol in the *Default* column of the *Toolchain Manager*.

Figure 118. Default toolchain updated

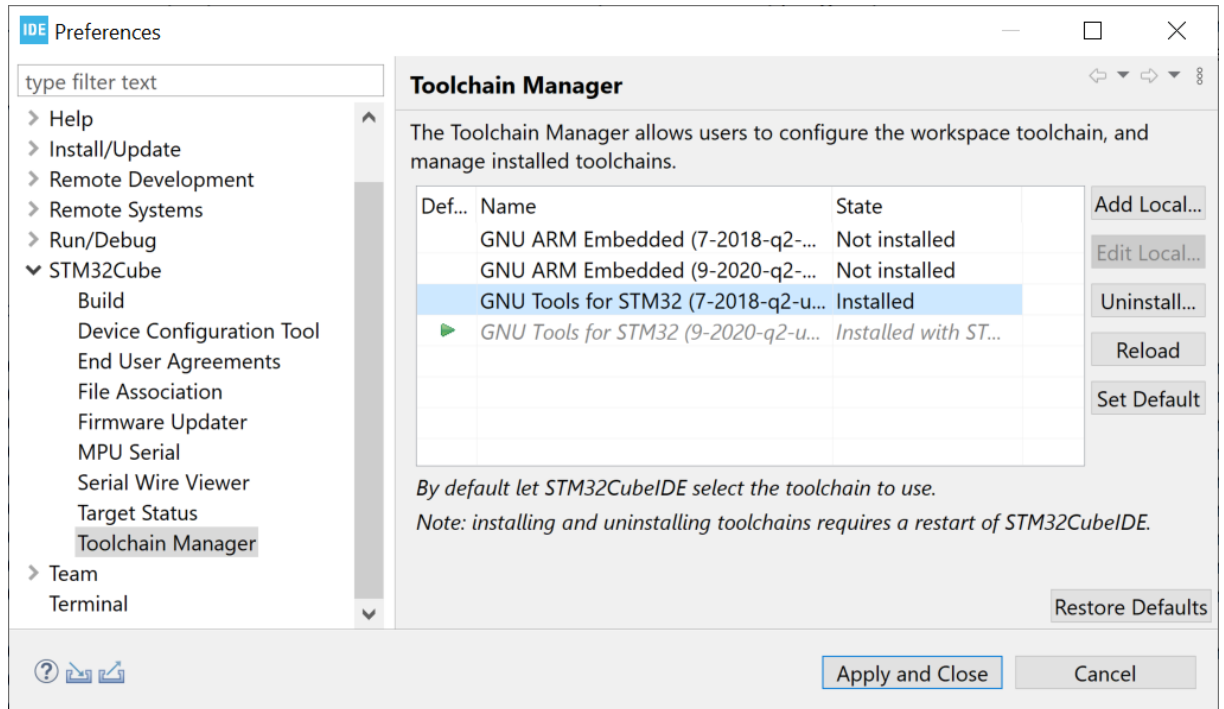


Click on **[Apply and Close]** to apply the setting and update which toolchain is set to be the default workspace toolchain.

2.11.3 Uninstall toolchain

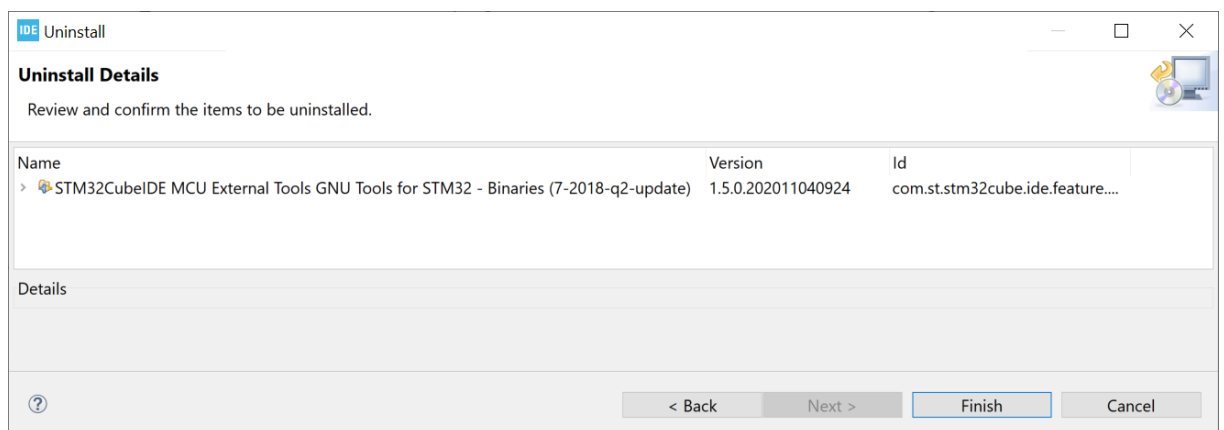
It is not possible to uninstall the *GNU Tools for STM32* toolchain, which is installed by default with STM32CubeIDE. Any other installed toolchain can be uninstalled.

Figure 119. Uninstall toolchain



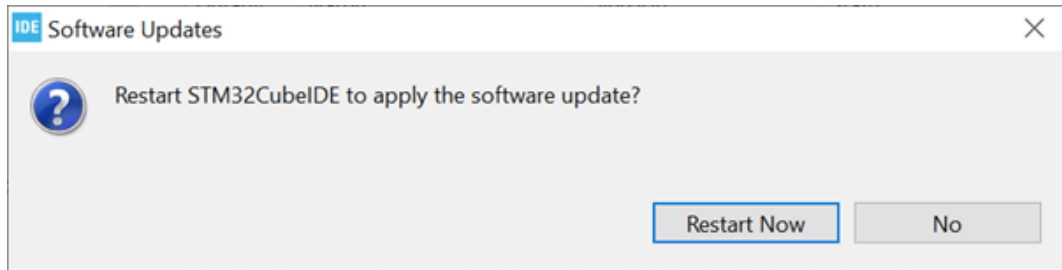
To uninstall a toolchain, select it in the *Toolchain Manager* and click on **[Uninstall...]**. This opens the *Uninstall* dialog.

Figure 120. Uninstall details



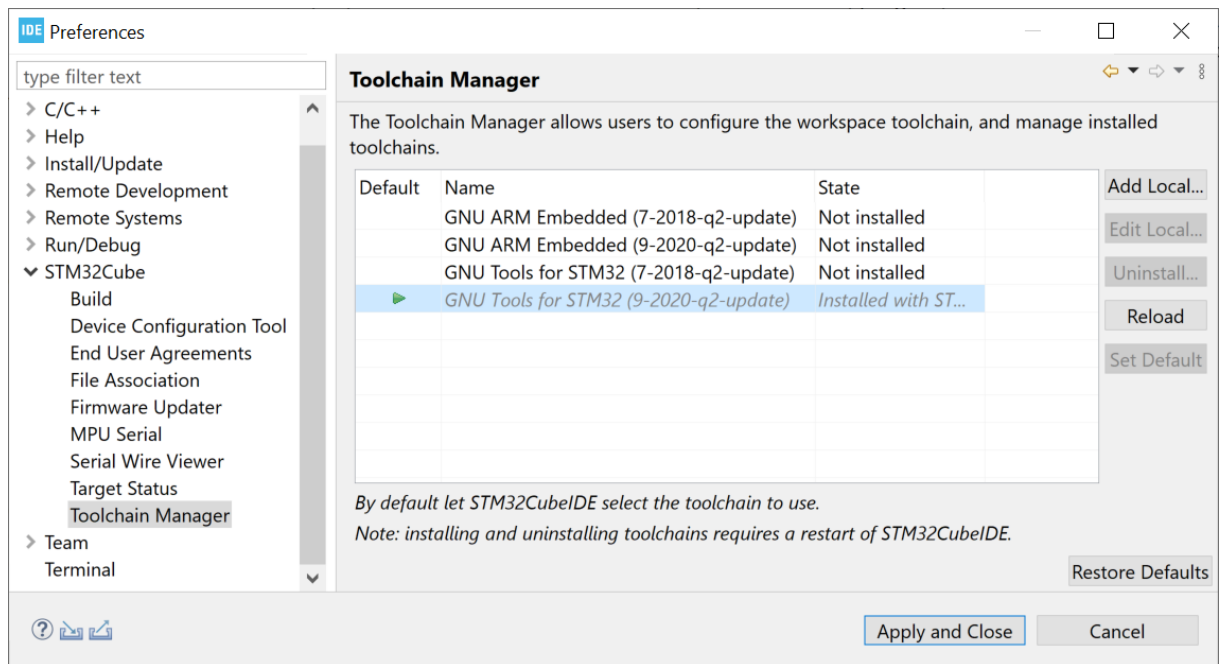
Click on **[Finish]** to start the software uninstallation. The *Software Updates* dialog is displayed.

Figure 121. Software updates



Click on **[Restart Now]** to apply the software update.
The product is restarted.
Open the *Toolchain Manager* to verify the installation.

Figure 122. Toolchain uninstalled



In this case, [Figure 122](#) shows that only one version of *GNU Tools for STM32* is installed.

2.11.4 Using local toolchain

It is possible to add and use an already installed local GNU ARM toolchain. To add a local toolchain, follow the steps below:

1. Open *Toolchain Manager* and press the [Add Local...] button.

Figure 123. Add local toolchain

IDE

Add local toolchain

✖ Give the toolchain a name

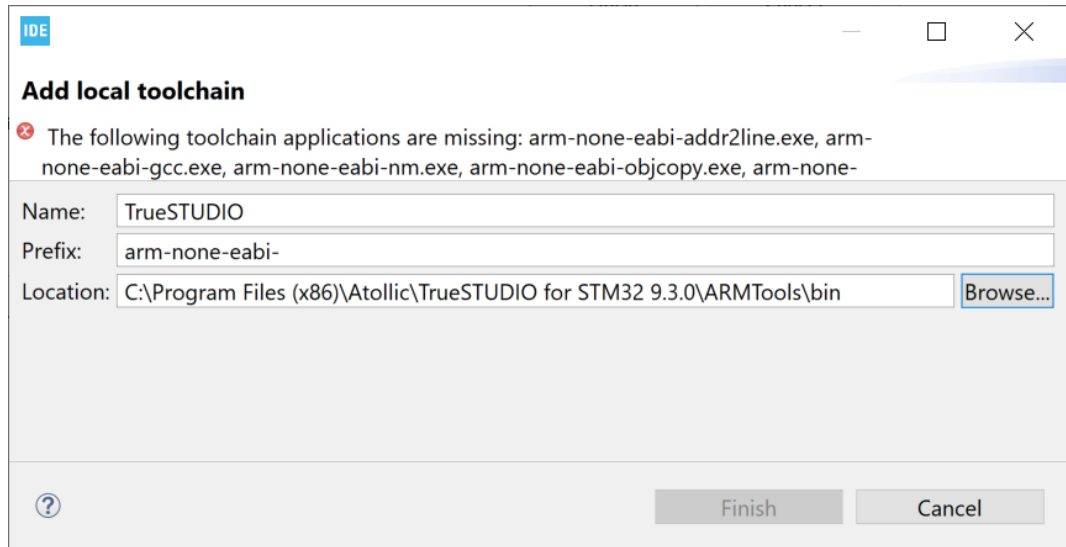
Name:

Prefix:

Location:

2. Add a name and specify location.

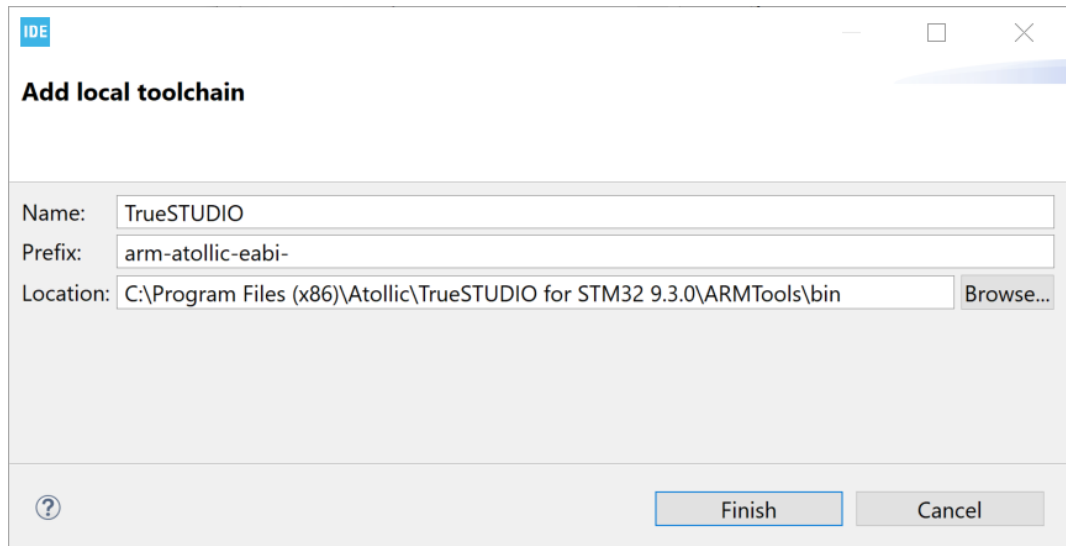
Figure 124. Specify local toolchain location



As seen in Figure 124, some naming problems can occur. In this case, the problem results from a wrong prefix that prevents the toolchain application validation.

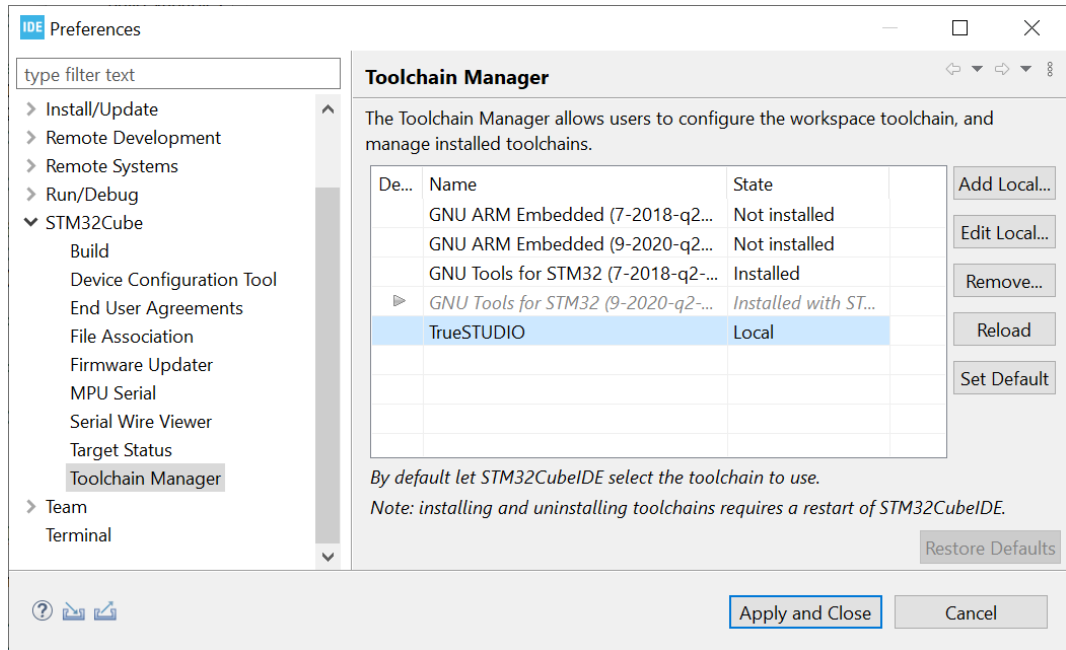
Update the toolchain prefix. The prefix must end with a dash (-).

Figure 125. Specify local toolchain prefix



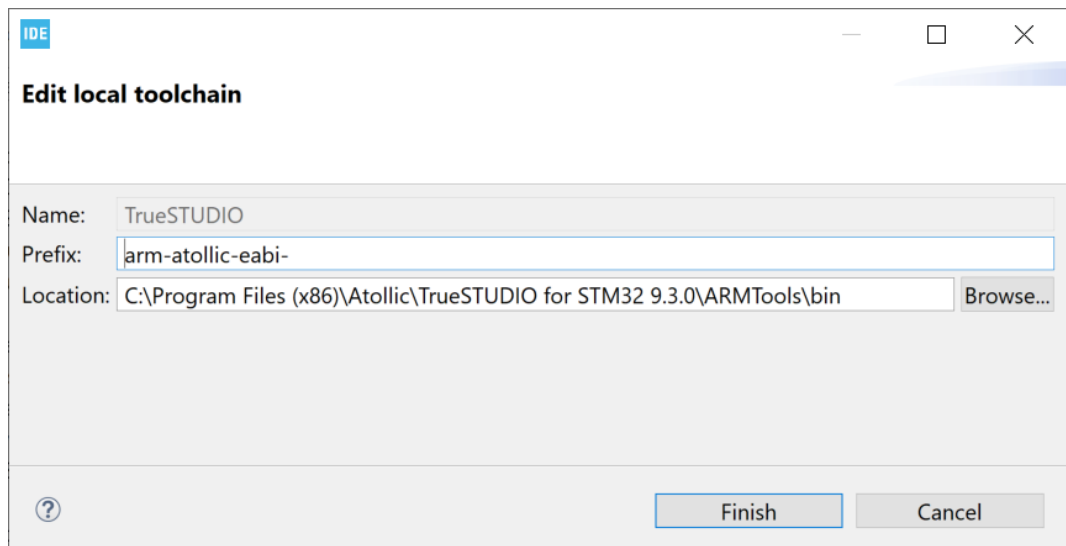
3. Press **[Finish]**.

Figure 126. Local toolchain added



4. Use the **[Edit Local...]** button to edit local toolchain. The *Edit local toolchain* dialog opens, and it is possible to update *Prefix* and *Location*.

Figure 127. Edit local toolchain

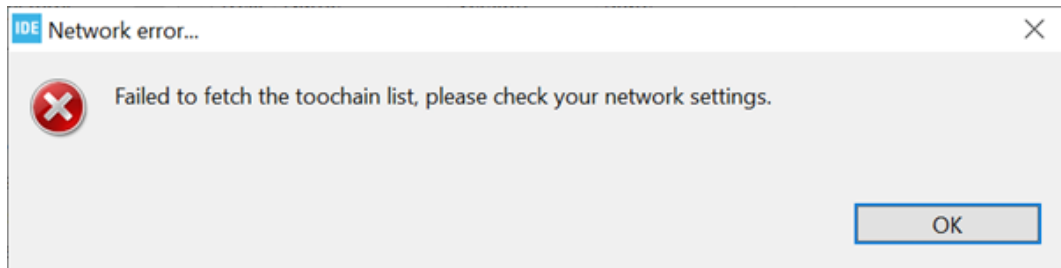


5. Update *Prefix* or *Location* and press **[Finish]** to update local toolchain settings.

2.11.5 Network error

In case of problem to access the update site, the *Network error...* dialog is displayed.

Figure 128. Toolchain network error



Check the network settings. Information on how to configure network proxy settings are described in [Section 1.5.3 Preferences - Network proxy settings](#).

3 Debugging

3.1 Introduction to debugging

STM32CubeIDE includes a powerful graphical debugger based on the GDB command-line debugger. It also bundles GDB servers for the ST-LINK and SEGGER J-Link JTAG probes.

The GDB server is a program that connects GDB on the PC to a target system. The STM32CubeIDE debug session can autostart a local GDB server or connect to a remote GDB server.

The remote GDB server can be running on the same PC, or on a PC that is accessible via the network and specified with *Host name* or *IP address* and a *Port number*. When connecting to a remote GDB server, this GDB server must be started first before a debug session is started in STM32CubeIDE.

When autostart local debugging is selected, STM32CubeIDE automatically starts and stops the GDB server as required during debugging, thus integrating the GDB server seamlessly.

Note: *It is recommended to use compiler optimization level `-O0` when building a project that must be debugged. Debugging with optimization level `-Og` may work but higher optimization level is hard to debug because of compiler code optimization.*

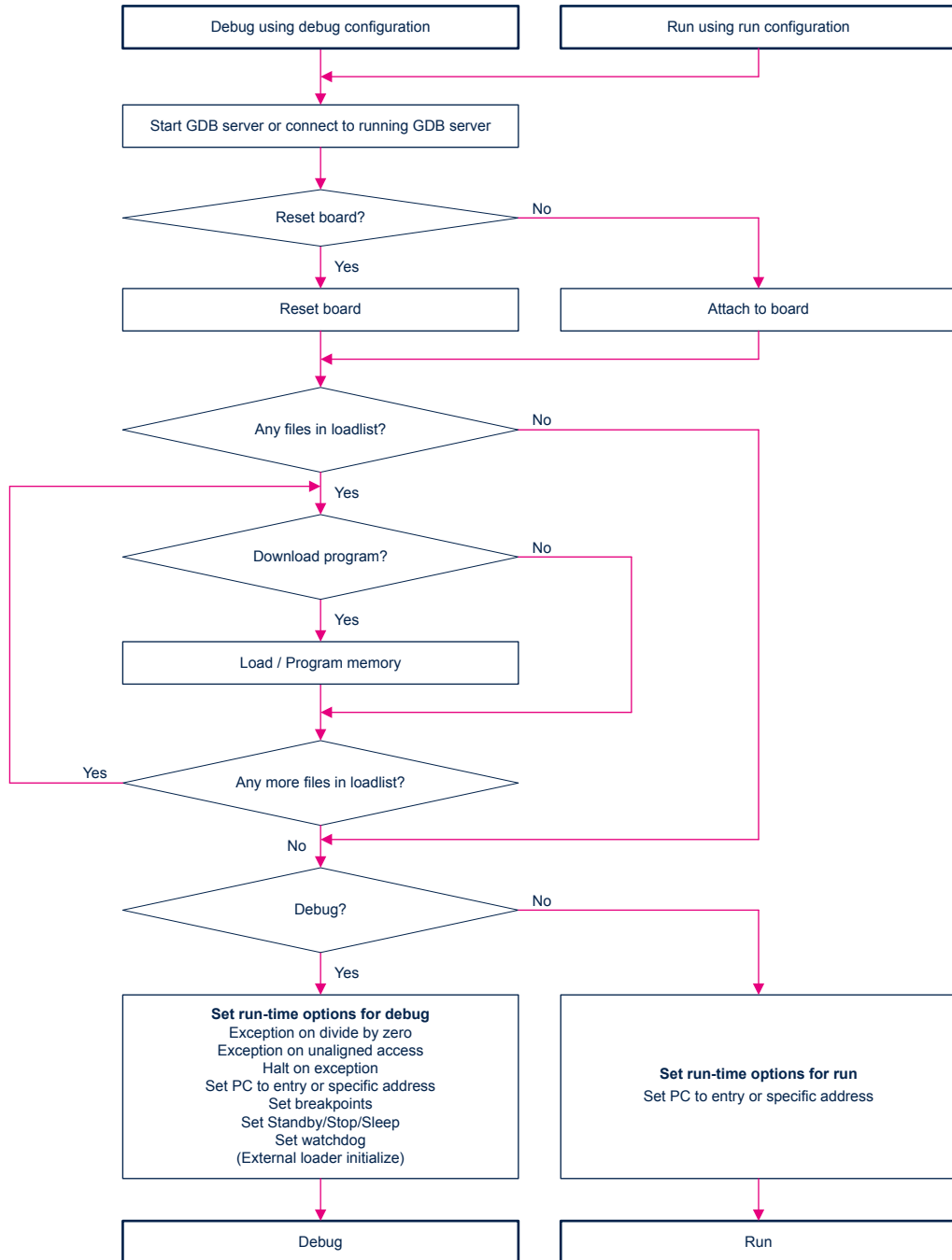
It is also possible to use the GDB server only to download the application into the target system and run it without starting a debug session. This is performed by creating run configurations, which is described later in this chapter (refer to [Section 3.7 Run configurations](#)).

STM32CubeIDE can be used to debug an existing `elf` file developed with another IDE or toolchain by importing the `elf` file using the import of STM32 Cortex[®]-M executable. This is described in [Section 3.8 Import STM32 Cortex-M executable](#).

3.1.1 General debug and run launch flow

Debug configurations are used to debug an STM32 program. Run configurations are used to flash a new program into the STM32 and start it. The flowchart in Figure 129 presents the order of starting the GDB server, reset the device, load the program, set run-time options, exceptions, program counter, breakpoints, Standby/Stop/Sleep, watchdog, and external loader initialization when starting a debug session. It also displays the differences between debug and run sessions.

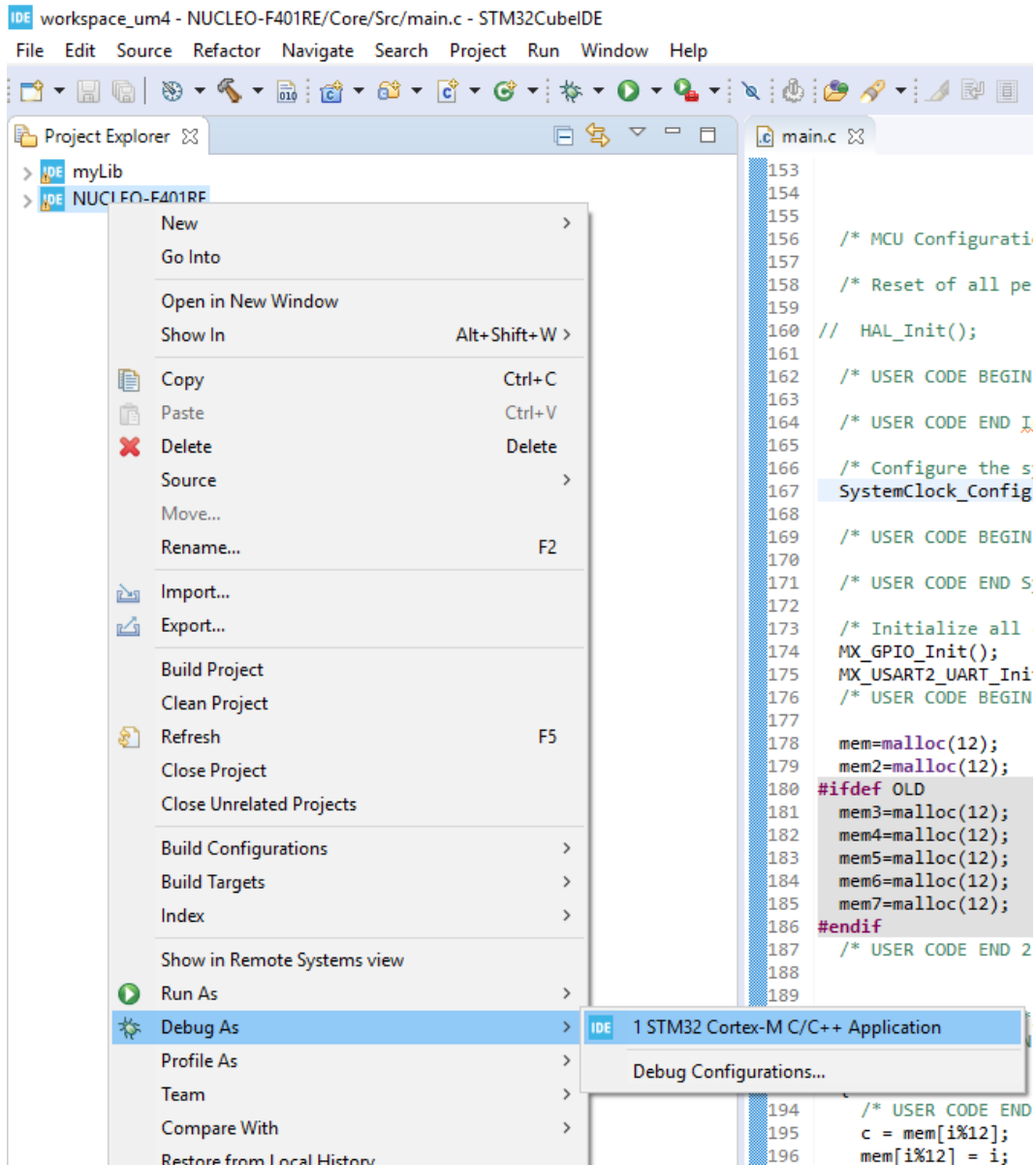
Figure 129. General debug and run launch flowchart



3.2 Debug configurations

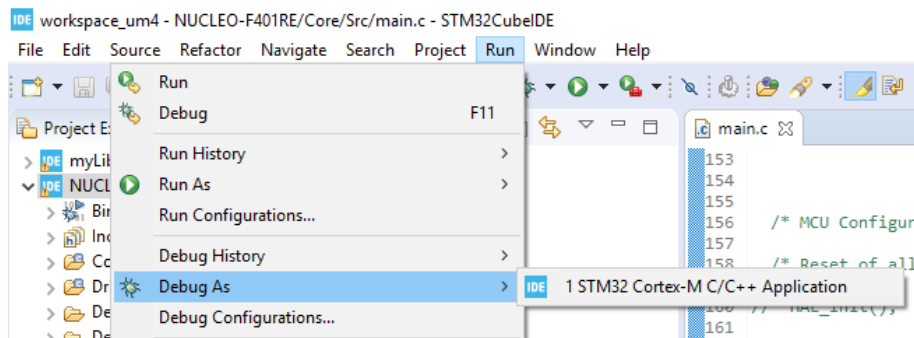
A debug configuration for the project is needed before a debug session can be started. To create the first debug configuration for the project, right click on the project name in the *Project Explorer* view and select [Debug As]>[STM32 Cortex-M C/C++ Application].

Figure 130. Debug as STM32 MCU



Another way to create a new debug configuration is to select the project name in the *Project Explorer* view and use the menu [Run]>[Debug As]>[STM32 Cortex-M C/C++ Application].

Figure 131. Debug as STM32 MCU menu



A third way to create a new debug configuration is to select project name in the *Project Explorer* view and press [F11].

All three different ways open the *Debug Configuration* dialog.

3.2.1 Debug configuration

The *Debug Configuration* dialog contains the following tabs:

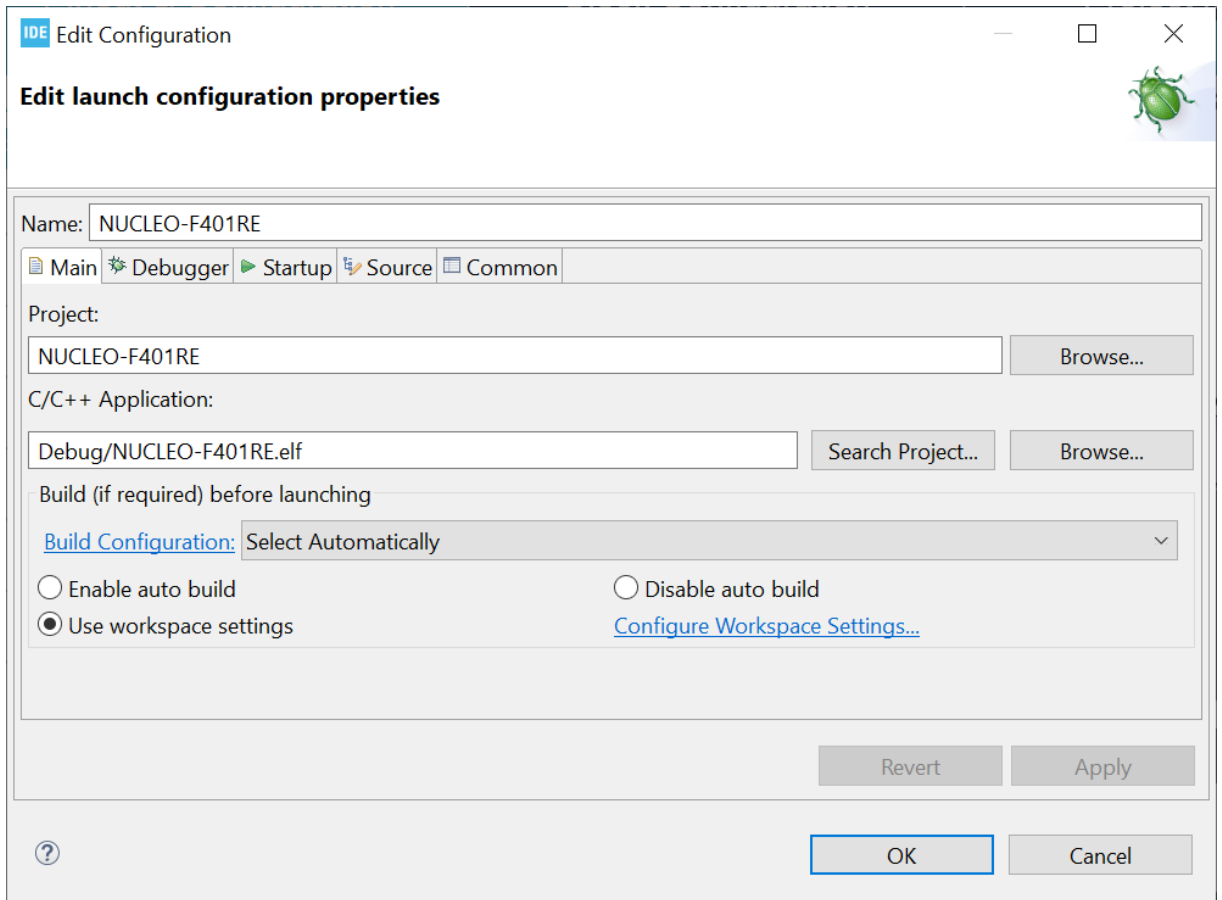
- *Main*
- *Debugger*
- *Startup*
- *Source*
- *Common*

The *Debugger* and *Startup* tabs must be updated when creating a new debug configuration while the others do not require update.

3.2.2 Main tab

The *Main* tab contains the configuration of the C/C++ application to debug. Usually, when creating a debug configuration using the sequence described earlier in this chapter, there is no need to make any change in the *Main* tab. Make sure the correct `elf` file and project are selected.

Figure 132. Debug configuration main tab



Note: It is possible in the Main tab to define if a build must be made before the debug session is started.

3.2.3 Debugger tab

The *Debugger* tab configures how to start the GDB server and connect to it. It also defines which GDB server must be used if [Autostart local GDB server] is selected.

Figure 133. Debug configuration debugger tab

Edit Configuration

Edit launch configuration properties

Name: NUCLEO-F401RE

Main Debugger Startup Source Common

GDB Connection Settings

Autostart local GDB server Host name or IP address localhost

Connect to remote GDB server Port number 61234

Debug probe ST-LINK (ST-LINK GDB server)

GDB Server Command Line Options

Show Command Line

Interface

SWD JTAG

ST-LINK S/N Scan

Frequency (kHz): Auto

Access port: 0 - Cortex-M4

Reset behaviour

Type: Connect under reset

Device settings

Debug in low power modes: Enable

Suspend watchdog counters while halted: No configuration

Serial Wire Viewer (SWV)

Enable

Core Clock (MHz): 16.0

Limit SWO clock

Maximum SWO clock (kHz): auto detect

Port number: 61235

RTOS Kernel Awareness

Enable RTOS Proxy

Driver settings

Driver: ThreadX

Port: cortex_m0

Port number: 60000

Misc

Verify flash download

Enable live expressions

Log to file: C:\Users\johansse\STM32CubeIDE\workspace_um8\NUCLEO-F401RE\ Browse...

External Loader: Scan Initialize

Shared ST-LINK

Max halt timeout(s): 2

Revert Apply

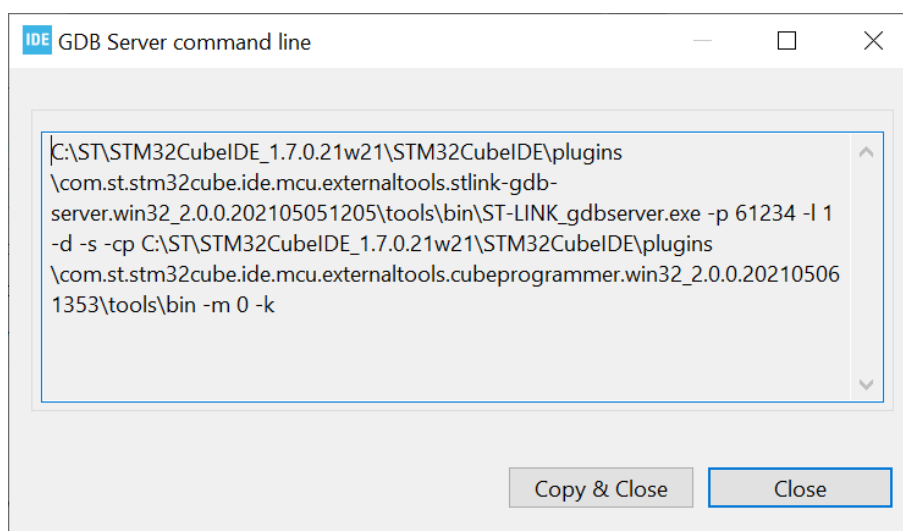
OK Cancel

The [Port number] edit field contains the default value used by the GDB server selected in field [Debug probe]. Field [Host name or IP address] must be set when [Connect to remote GDB server] is selected.

Field **[Debug probe]** selects the probe and GDB server to be used for debugging. When using an ST-LINK debug probe, *ST-LINK GDB server* or *OpenOCD* can be used. When using a SEGGER J-LINK probe, use the *SEGGER J-LINK GDB server*.

Pressing the **[Show Command Line]** button opens the *GDB Server command line* dialog. The dialog displays how the GDB server is started according to the current **[GDB Server Command Line options]** settings.

Figure 134. GDB server command line dialog



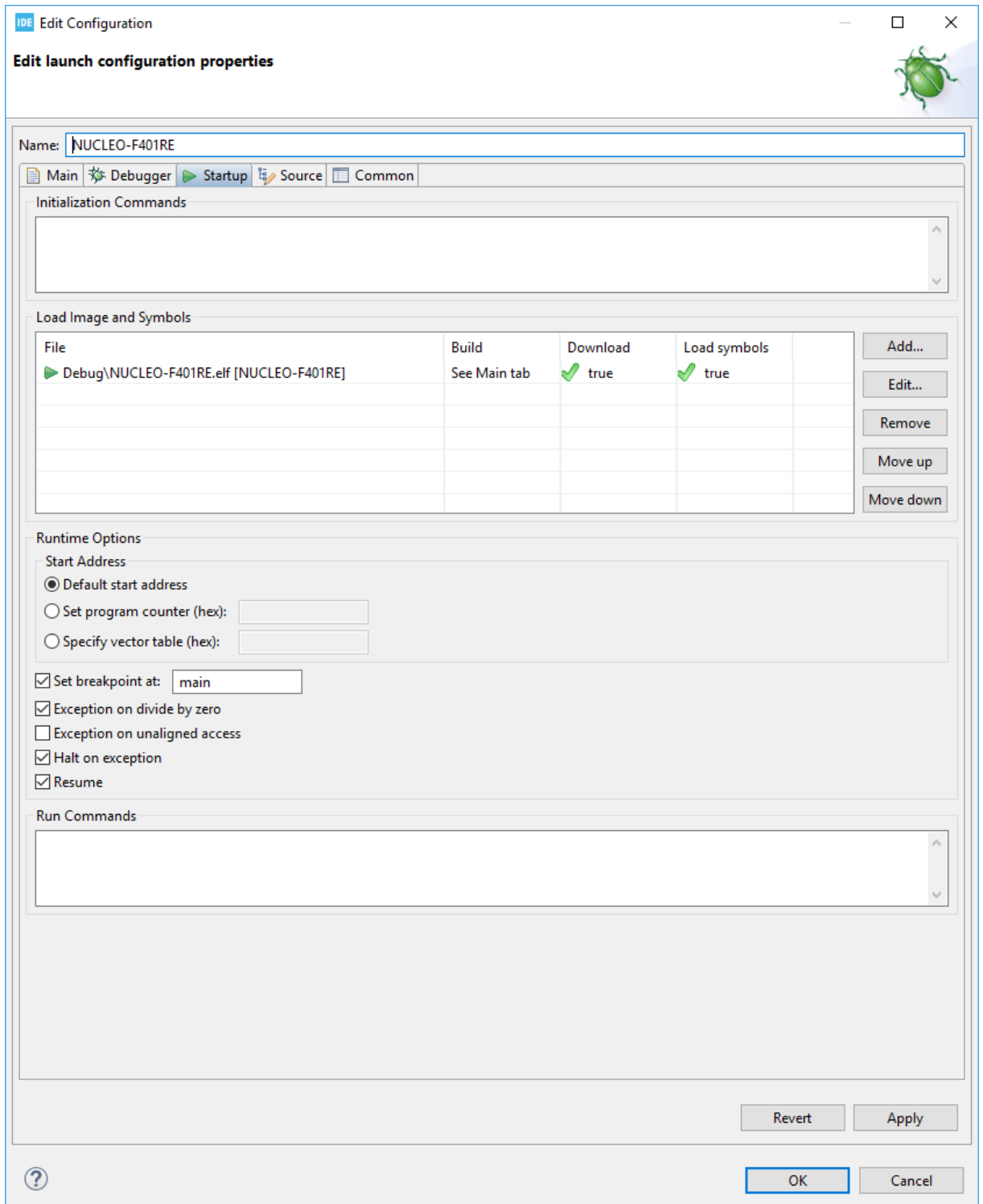
Use the **[Copy & Close]** button to copy the current command line settings to the clipboard, for instance to start the GDB server manually in a command line window by pasting the command.

The **[GDB Server Command Line options]** selections are updated as a function of the **[Debug probe]** selected. Detailed information about these settings are available in [Section 3.4 Debug using different GDB servers and sub-sections](#).

3.2.4 Startup tab

The *Startup* tab configures how to start a debug session.

Figure 135. Debug configuration startup tab



The **[Initialization Commands]** edit field can be updated with any kind of GDB or GDB server monitor commands if there is any special need to send some commands to the GDB server before load commands are sent. For instance, when using ST-LINK GDB server a `monitor flash mass_erase` command can be entered here if a Flash memory erase is needed before load.

The **[Load Image and Symbols]** list box must contain the file(s) to debug. This list is associated with the following command buttons:

- **[Add...]**: Add new lines for files for download and/or load symbols
- **[Edit...]**: Edit the selected line
- **[Remove]**: Remove the selected line from the list
- **[Move up]**: Move the selected line upwards
- **[Move down]**: Move selected line downwards

The **[Runtime Options]** section contains checkboxes to set the start address and breakpoint, and enable exception handling and resume.

The start address can be selected as:

- **[Default start address]**: `$pc` is set to the start address found in the last loaded `elf` file
- **[Set program counter (hex)]**: `$pc` is set to the hex value specified in the edit field
- **[Specify vector table (hex)]**: `$pc` is updated with the value found in memory using specified address + offset of 4. This is similar to how `$pc` is set by a reset using vector table in a Cortex[®]-M device

The **[Set breakpoint at:]** checkbox is enabled by default and the edit field displays `main`. It means that, by default, a breakpoint is set at `main` when the program is debugged.

Three exception checkboxes, **[Exception on divide by zero]**, **[Exception on unaligned access]** and **[Halt on exception]**, are used to make it easier to find problems when debugging an application.

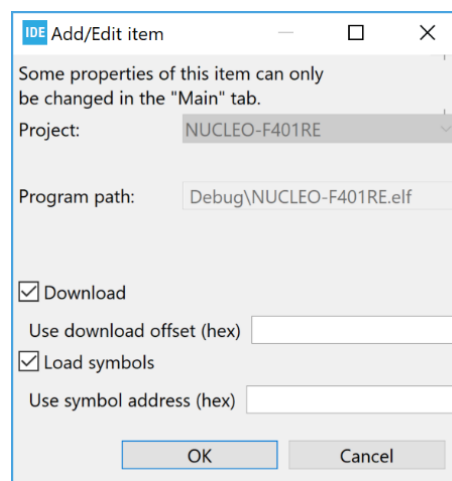
- **[Exception on divide by zero]** is enabled by default to make it easier to trap a divide-by-zero error when debugging
- **[Exception on unaligned access]** can be enabled to get exceptions if there are any unaligned accesses
- **[Halt on exception]** is enabled by default so that program execution halts when an exception error occurs during debugging. If an exception occurs, the *Fault Analyzer* view can be used to find the location of the problem

Note: The application software needs to enable **Exception on divide by zero** and **Exception on unaligned access** if they must be issued also when running the application and not only during debugging. The CMSIS Cortex[®]-M header files contain defines to update the SCB Configuration Control Register. For instance, `core_cm4.h` contains the `SCB->CCR` register, and `SCB_CCR_DIV_0_TRP` and `SCB_CCR_UNALIGN_TRP` defines.

When the **[Resume]** selection is enabled, a `continue` command is issued to GDB after load to start the program. Usually, in this case, the program breaks at `main` if a breakpoint at `main` is set. Otherwise, when the **[Resume]** selection is disabled, the program stays at the `ENTRY` location specified in the linker script, normally the `Reset_Handler` function. A step may be needed in this case to display the `Reset_Handler` function in the editor.

When a line in the listbox is selected and **[Edit...]** is pressed, the following dialog appears for selecting if the file must be downloaded and if symbols must be loaded.

Figure 136. Add/Edit item

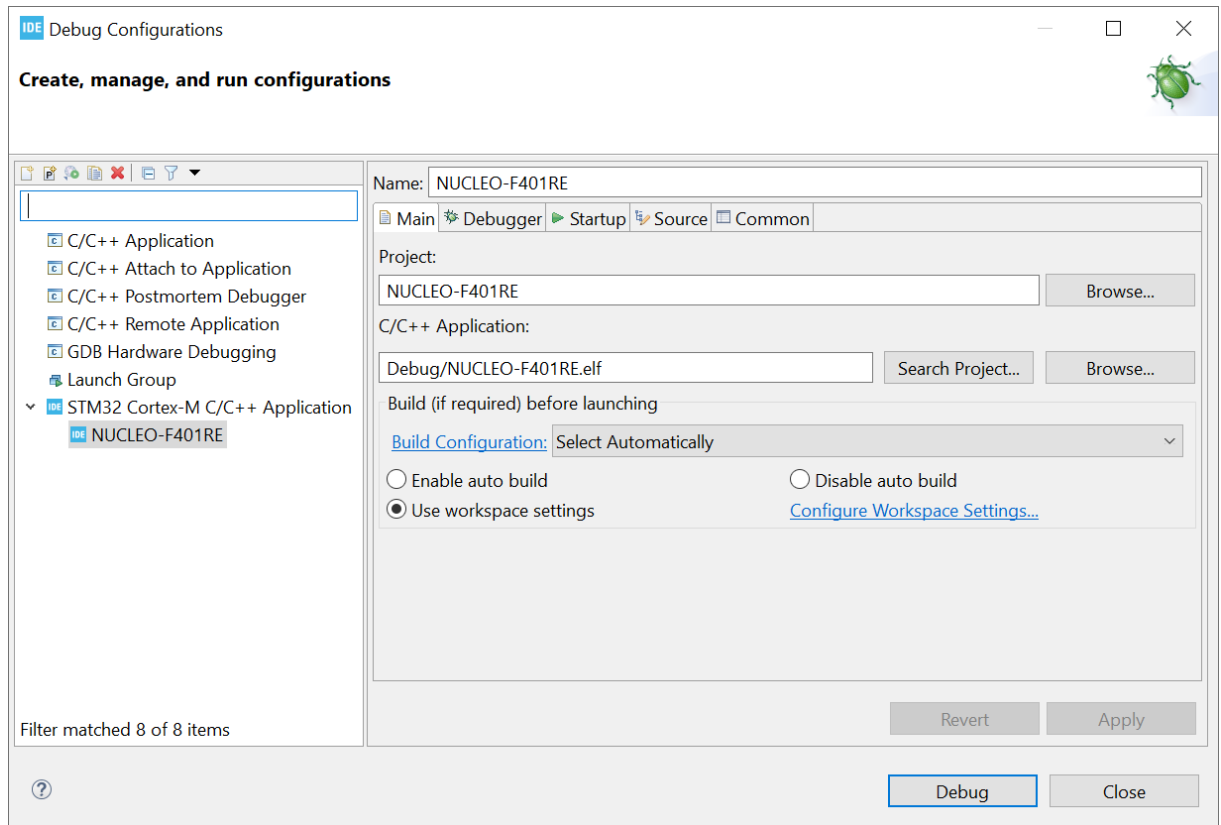


3.3 Manage debug configurations

Each project can have several debug configurations. It is easy to create a copy of an existing debug configuration and update it with some changes. For instance, one configuration may contain Flash memory loading of new programs while another does not load any program.

When opening debug configurations from the menu [Run]>[Debug Configurations...], the *Debug Configurations* dialog opens. This dialog contains a navigation window on the left side with a toolbar, and the debug configuration on the right side with the tabs and fields described in Section 3.2 *Debug configurations*.

Figure 137. Manage debug configurations



The [Name] field on top of the right pane can be edited using a name for the debug configuration which reflects the configuration. This name then appears in the navigation window under the [STM32 Cortex-M C/C++ Application] node to the left when pressing [Apply].

The toolbar left of the navigation window contains icons to manage configurations, for instance to duplicate or delete a selected configuration.

Figure 138. Manage debug configurations toolbar



These icons are used for the following purpose, from left to right:

- Create new launch configuration
- New launch of configuration prototype
- Export launch configuration
- Duplicate currently selected launch configuration
- Delete selected launch configuration(s)
- Collapse all expanded launch configurations
- Filter launch configurations

3.4 Debug using different GDB servers

STM32CubeIDE includes the following GDB servers:

- ST-LINK GDB server
- OpenOCD GDB server
- SEGGER J-Link GDB server

All three GDB servers support normal debug, live expressions and SWV.

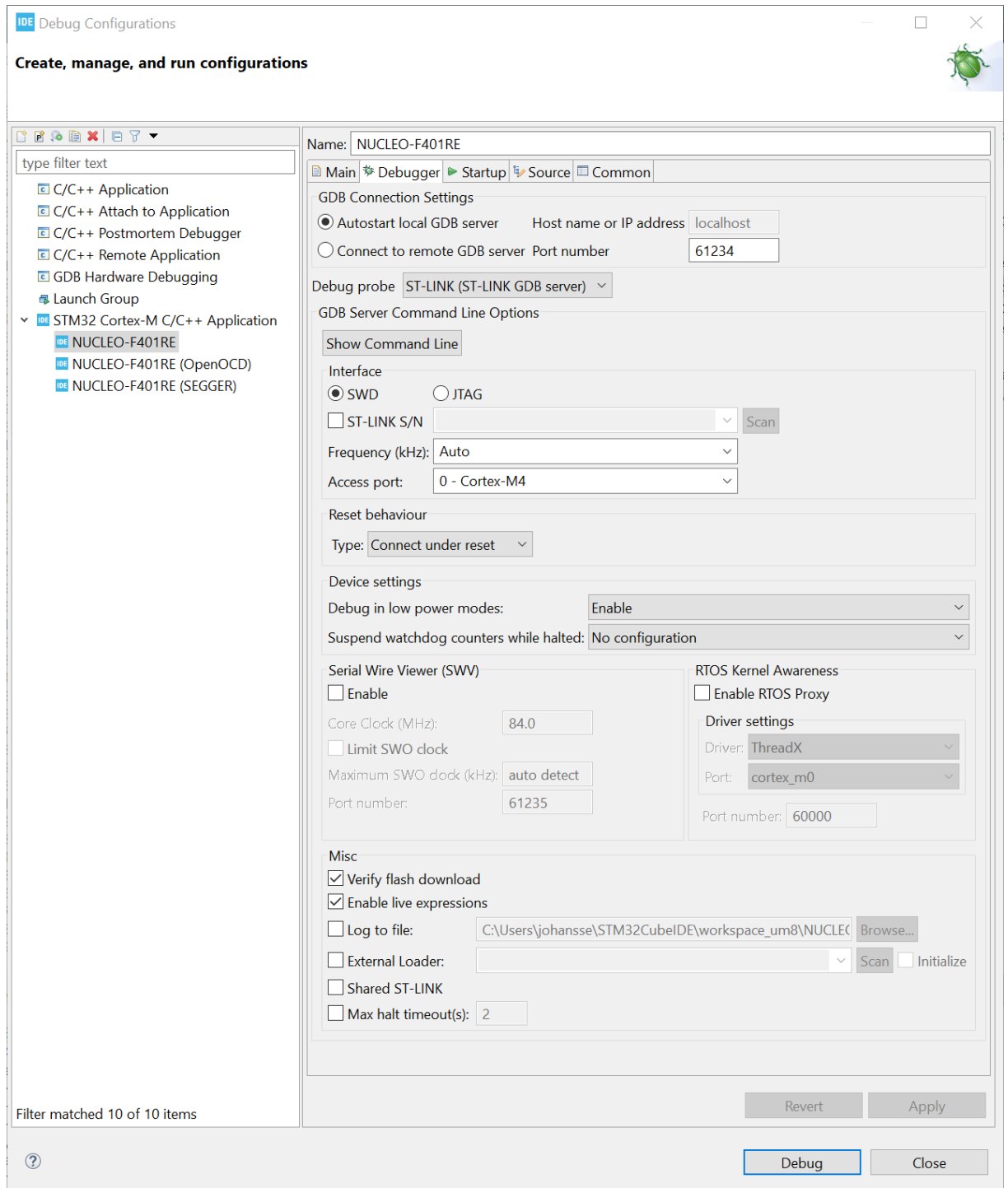
All GDB servers also support RTOS Kernel Awareness debugging for Microsoft® Azure® RTOS ThreadX and FreeRTOS™ operating systems using an RTOS proxy. The RTOS proxy is included in STM32CubeIDE.

Different command-line options are used when starting these GDB servers. Therefore the *Debugger* tab in the *Debug Configurations* dialog displays different settings depending on the GDB server selected. This section describes the individual settings for each server.

3.4.1 Debug using the ST-LINK GDB server

Usually, when the ST-LINK GDB server is used for debugging, there is no need to update the [**GDB Server Command Line Options**] in the Debugger tab. In some cases, the default configuration must be updated, for instance if SWV is intended to be used or if several STM32 boards are connected to the PC.

Figure 139. ST-LINK GDB server debugger tab



Select **[SWD]** or **[JTAG]** in **[Interface]** to define how the ST-LINK probe must connect with the microcontroller. The SWD interface is usually the preferred choice. It must be selected if SWV is to be used.

When **[ST-LINK S/N]** is enabled, the serial number of the ST-LINK probe to be used must be entered in the edit/list field. The **[Scan]** button can be used to scan and list all detected ST-LINK devices connected to the PC. After a scan, the S/N of these ST-LINK devices are listed in the list box from which the desired ST-LINK can be selected. When **[Use specific ST-LINK S/N]** is enabled, the ST-LINK GDB server is started and connects only to the ST-LINK with the selected S/N.

The **[Frequency (kHz)]** selection defines the communication speed between the ST-LINK and STM32 device. When **[Auto]** is selected, the maximum speed provided by ST-LINK is used. Reduce the frequency in case of hardware limitations.

The **[Access port]** selection is used only when debugging a multi-core STM32 device. In such case, the ST-LINK is connected to the device and the ST-LINK GDB server must be informed of the core to debug.

The **[Reset behaviour]** contains selections for **[Type]** and **[Halt all cores]**. The **[Halt all cores]** selection is only visible for multi-core devices.

The **[Type]** can be set as follows:

- **[Connect under reset]** (default): ST-LINK reset line is activated and ST-LINK connects in the SWD or JTAG mode while reset is active. Then the reset line is deactivated.
- **[Software system reset]**: System reset is activated by software writing in a register. This resets the core and peripherals, and can reset the whole system as the reset pin of the target is asserted by itself.
- **[Hardware reset]**: ST-LINK reset line is activated and deactivated (pulse on reset line), then ST-LINK connects in the SWD or JTAG mode.
- **[Core reset]**: Core reset is activated by software writing in a register (not possible on Cortex[®]-M0, Cortex[®]-M0+ and Cortex[®]-M33 cores). This only resets the core, not the peripherals nor the reset pin.
- **[None]**: For attachment to a running target where the program is downloaded into the device already. There must not be any file program command in the *Startup* tab.

Note:

The selected reset behaviour is overridden if the debug configuration includes Flash programming, in which case the ST-LINK GDB server uses the STM32CubeProgrammer (STM32CubeProg) command-line program STM32_Programmer_CLI to program the Flash memory. This program is always started by the ST-LINK GDB server with mode=UR reset=hwRst so that a device reset is done when loading a new program, disregarding the selection of the [None] option. This ensures that device programming is made correctly.

[Halt all cores] can be used only when debugging multi-core devices. The **[Halt all cores]** selection is not visible for single core devices.

[Device settings] contains selections for **[Debug in low power modes]** and **[Suspend watchdog counters while halted]**. These can be defined as:

- **[No configuration]**
- **[Enable]**
- **[Disable]**

The **[Serial Wire Viewer (SWV)]** selections can be used only when the **[SWD]** interface is selected. When **[SWV]** is enabled, it is required to configure the **[Clock Settings]**. The **[Core Clock]** must be set to the device speed. More information about SWV configuration is available in [Section 4.2.1 SWV debug configuration](#).

The **[RTOS Kernel Awareness]** selections are used to enable RTOS-kernel-aware debugging with the ThreadX and FreeRTOS[™] operating systems. When RTOS-kernel-aware debugging is enabled and a debug session is started, all threads are listed in the *Debug* view. By selecting a thread in the *Debug* view the current line executed by the thread is displayed in the editor. More information about RTOS-kernel-aware debugging is available in [Section 6.3](#).

The **[Misc]** selections contains:

- **[Verify flash download]**
- **[Enable live expressions]** (To be able to use the *Live Expressions* view during debugging, the live expression mechanism must be enabled during startup. It is enabled by default.)
- **[Log to file]** (Enable in case of debugging problems. It starts the ST-LINK GDB server with a higher log level and saves the log into a file.)
- **[External Loader]** (Enable if loading must be made to non-internal STM32 Flash memory). A **[Scan]** button is available to access STM32CubeProgrammer external Flash loader files.

When **[External Loader]** is enabled, there is also an **[Initialize]** selection. When it is enabled, the `Init()` function in STM32CubeProgrammer is called after reset. It can be used to configure the device for external memory access. Usually, initialization must be done by the debugged application.

- **[Shared ST-LINK]** (Shared ST-LINK must be enabled if other programs must be able to connect to the same ST-LINK during a debug session.). Refer to [Section 3.6.2 Shared ST-LINK](#) for details.

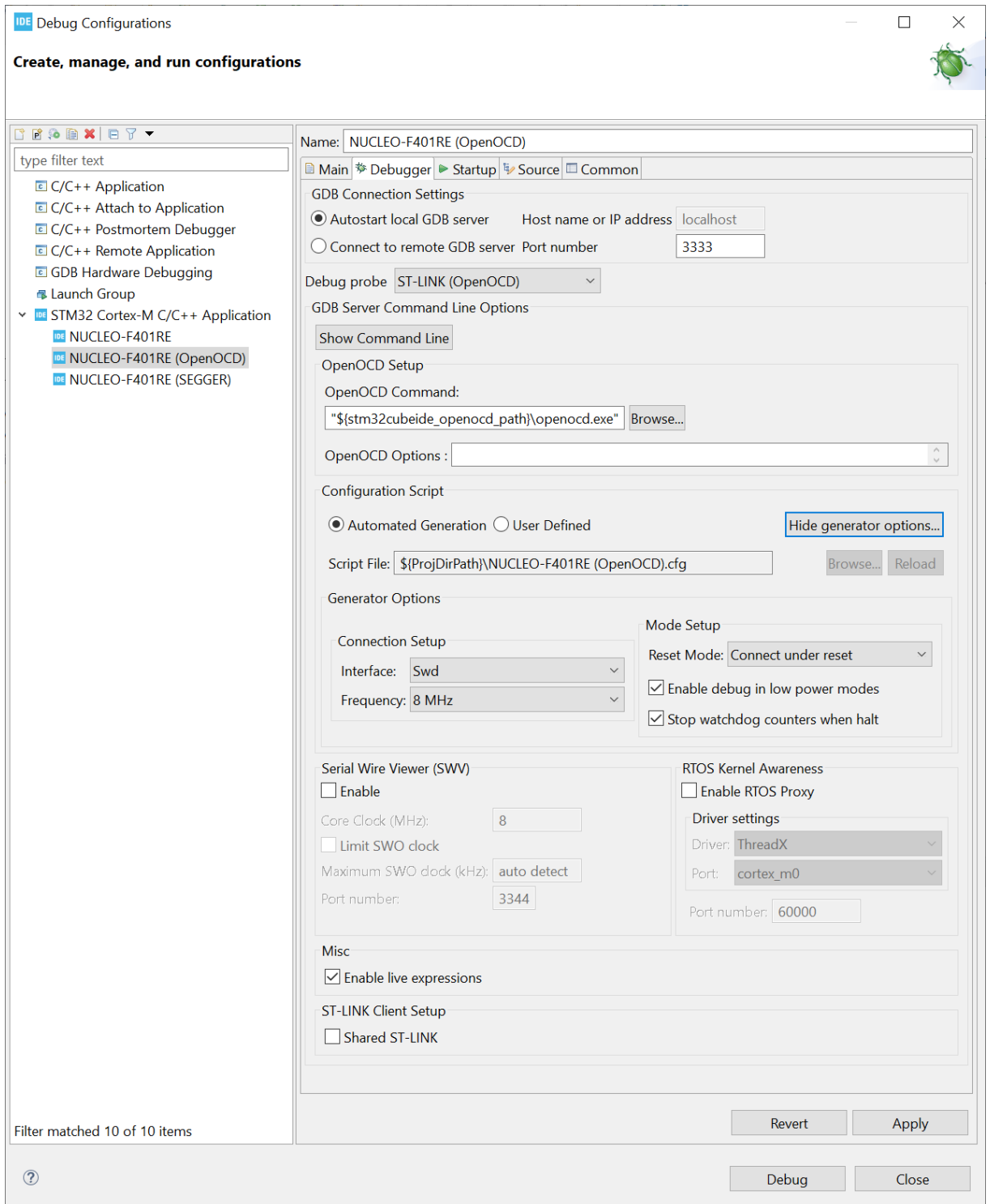
A detailed description of the ST-LINK GDB server is available in the ST-LINK GDB server manual ([\[ST-07\]](#)), which is available from the *Information Center*.

Note: *STM32_Programmer_CLI* is used by the ST-LINK GDB server to program the STM32 or external Flash memory. In this case, such external Flash memory programming is automatically done using the external loader.

3.4.2 Debug using OpenOCD and ST-LINK

When OpenOCD is used, the [GDB Server Command Line Options] in the *Debugger* tab contains a generator options toggle field, which alternates between [Show generator options...] and [Hide generator options...]. When the field is set to [Hide generator options...], the dialog displays additional [GDB Server Command Line Options] as shown in [Figure 140](#).

Figure 140. OpenOCD debugger tab



The **[OpenOCD Command]** edit field contains the `openocd.exe` file to be used when debugging. The **[Browse]** button can be used to select another version of OpenOCD.

The **[OpenOCD Options]** edit field can be used to add additional command-line parameters to be used when starting OpenOCD.

The **[Configuration Script]** selections can be **[Automated Generation]** or **[User Defined]**. When **[Automated Generation]** is selected, an `openocd.cfg` file is created automatically based on the selections made in the *Debugger* tab. When **[User Defined]** is selected, the file must be specified in the **[Script File]** edit field.

The **[Interface]** selection **[Swd]** or **[Jtag]** selects how the ST-LINK probe must connect with the microcontroller. **[Swd]** is usually the preferred choice.

The **Frequency** selection configures the communication speed between the ST-LINK and STM32 device.

The **[Reset Mode]** selection contains:

- **[Connect under reset]** (default): ST-LINK reset line is activated and ST-LINK connects in the SWD or JTAG mode while reset is active. Then the reset line is deactivated.
- **[Hardware reset]**: ST-LINK reset line is activated and deactivated (pulse on reset line), then ST-LINK connects in the SWD or JTAG mode.
- **[Software system reset]**: System reset is activated by software writing in a register. This is resetting the core and peripherals, and can reset the whole system as the reset pin of the target is asserted by itself.
- **[Core reset]**: Core reset is activated by software writing in a register (not possible on Cortex[®]-M0, Cortex[®]-M0+ and Cortex[®]-M33 cores). This is only resetting the core, not the peripherals nor the reset pin.
- **[None]**: For attachment to a running target where the program is downloaded into the device already. There must not be any file program command in the *Startup* tab.

[Enable debug in low power modes] enables debug also with the STM32 device in low-power mode.

[Stop watchdog counters when halt] stops the watchdog when the debug session halts the STM32 device. Otherwise, a watchdog interrupt may be triggered.

The **[Serial Wire Viewer (SWV)]** selections can be used only when the **[SWD]** interface is selected. When **[SWV]** is enabled, it is required to configure the **[Clock Settings]**. The **[Core Clock]** must be set to the device speed. More information about SWV configuration is available in [Section 4.2.1 SWV debug configuration](#).

The **[RTOS Kernel Awareness]** selections are used to enable RTOS-kernel-aware debugging with the ThreadX and FreeRTOS™ operating systems. When RTOS-kernel-aware debugging is enabled and a debug session is started, all threads are listed in the *Debug* view. By selecting a thread in the *Debug* view the current line executed by the thread is displayed in the editor. More information about RTOS-kernel-aware debugging is available in [Section 6.3](#).

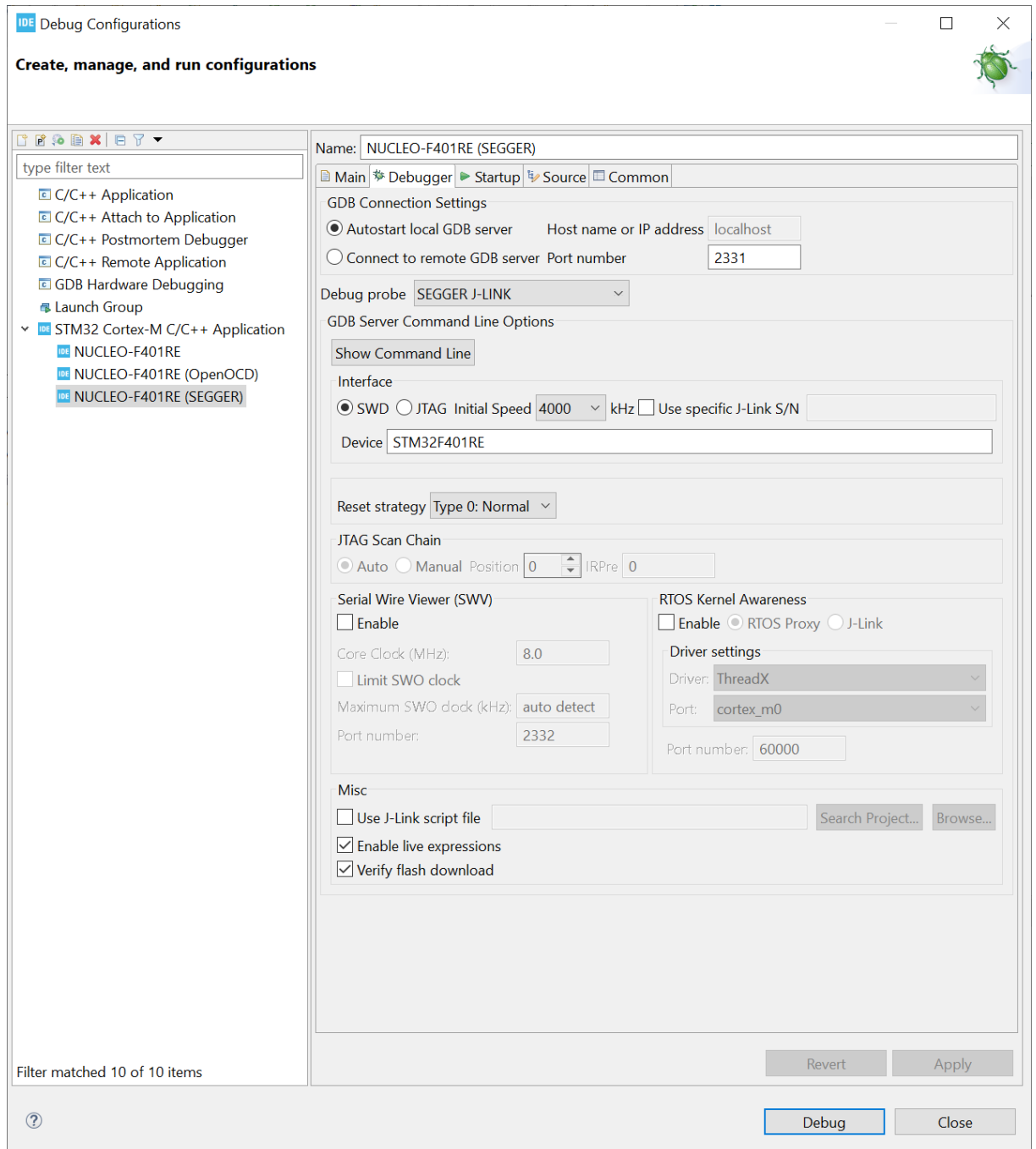
[Enable live expressions] must be enabled if the *Live Expressions* view is meant to be used during debugging.

[Shared ST-LINK] must be enabled if other programs have to connect to the same ST-LINK during a debug session. Refer to [Section 3.6.2 Shared ST-LINK](#) for details.

3.4.3 Debug using SEGGER J-Link

When **[SEGGER J-LINK]** is selected in the *Debugger* tab, the **[GDB Server Command Line Options]** corresponds to SEGGER J-Link GDB server.

Figure 141. SEGGER debugger tab



The **[Interface]** selection **[SWD]** or **[JTAG]** selects how the SEGGER J-Link probe must connect with the microcontroller. The **[SWD]** interface is usually the preferred choice; it is required if SWV is used.

The **[Initial Speed]** selection configures the communication speed used between SEGGER J-Link and the STM32 device.

When **[Use specific J-Link S/N]** is enabled, enter the S/N of the J-Link to be used when debugging in the edit/list field. When **[Use specific J-Link S/N]** is enabled, the SEGGER J-Link GDB server is started and connects only to the J-Link with the selected S/N.

The **Device** edit field is used if it contains an entry. This field can be used if there is a problem to start the SEGGER J-Link GDB server with the default device name used in [STM32CubeIDE](#). In such case, enter the device name used by the SEGGER GDB server in the edit field.

The **[Reset strategy]** selection contains:

- **[Type 0: Normal]** - Default.
- **[None]** - Intended to be used for attaching to the running target. In this case, the program must already be downloaded into the device. There must not be any file program command in the *Startup* tab.

The **[JTAG Scan Chain]** selections can be used only when the **[JTAG]** interface is selected.

The **[Serial Wire Viewer (SWV)]** selections can be used only when the **[SWD]** interface is selected. When **[SWV]** is enabled, it is required to configure the **[Clock Settings]**. The **[Core Clock]** must be set to the device speed. More information about SWV configuration is available in [Section 4.2.1 SWV debug configuration](#).

The **[RTOS Kernel Awareness]** selections are used to enable RTOS-kernel-aware debugging with the ThreadX and FreeRTOS™ operating systems. When RTOS-kernel-aware debugging is enabled and a debug session is started, all threads are listed in the *Debug* view. By selecting a thread in the *Debug* view the current line executed by the thread is displayed in the editor. More information about RTOS-kernel-aware debugging is available in [Section 6.3](#).

The **[Misc]** selections contains:

- **[Use J-Link script file]**
- **[Enable live expressions]**
To be able to use the *Live Expressions* view during debug, the live expression mechanism must be enabled during startup.
- **[Verify flash download]**
- **[Select RTOS variant]** list box can be used if **[Thread-aware RTOS support]** is used with **[FreeRTOS]** and **[embOS]**.

When **[Thread-aware RTOS support]** is used, update the *Startup* tab: disable **[Resume]** and **[in Run Commands]**, add `thread 2` and `continue`. This forces a thread context switch before the `continue` command is sent.

Note: A detailed description of SEGGER J-Link GDB server is available in the SEGGER J-Link manual, which can be accessed from the “Information Center”.

3.5 Start and stop debugging

When a debug configuration is created for the project with the preferred JTAG probe, it is ready for debugging. In the following sections, the ST-LINK GDB server is used. However, the way to debug the STM32 project is quite independent of the choice among ST-LINK GDB server, OpenOCD or SEGGER J-Link.

Perform the following steps to prepare for debug:

1. Determine whether the board supports the JTAG debug, SWD debug, or both.
SWD-mode debug is usually the preferred choice.
2. Connect the JTAG cable between the JTAG probe and the target board.
When using STMicroelectronics STM32 Nucleo and Discovery boards, the ST-LINK is usually integrated on the board. Also, most STMicroelectronics STM32 Evaluation boards contain an embedded ST-LINK.
3. Connect the USB cable between the PC and the JTAG probe.
4. Make sure that the target board has a proper power supply attached.

Once the steps above are performed, a debug session can be started.

3.5.1 Start debugging

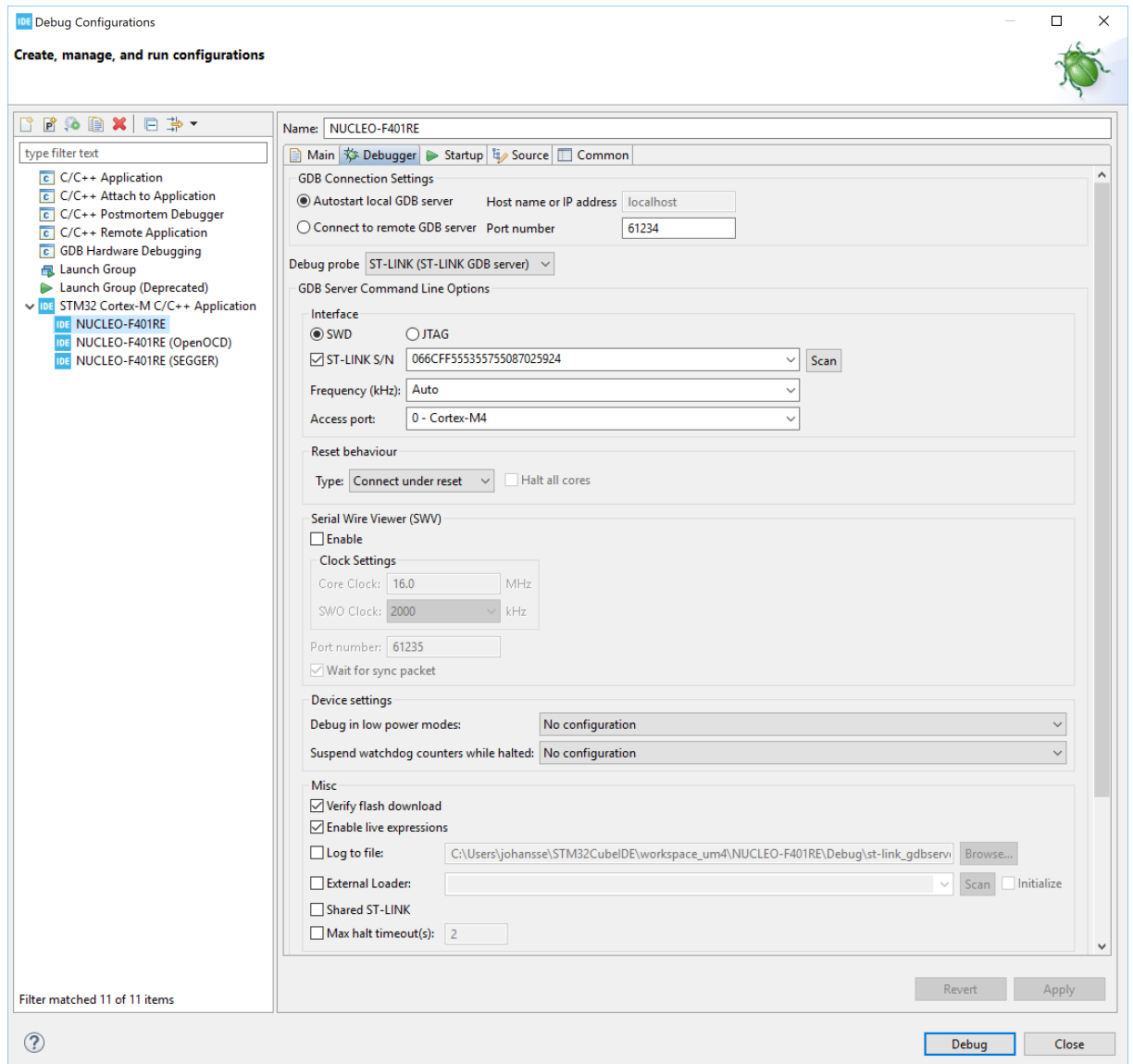
Open the *Debug Configurations* dialog with a right click on the project name in the *Project Explorer* view and select **[Debug As]>[Debug Configurations...]**.

It is also possible to open the dialog using the menu **[Run]>[Debug Configurations...]**.

This opens the *Debug Configurations* dialog.

Note: It is possible to select the project in the “Project Explorer” view and press **[F11]** to restart a debug session after it has been closed.

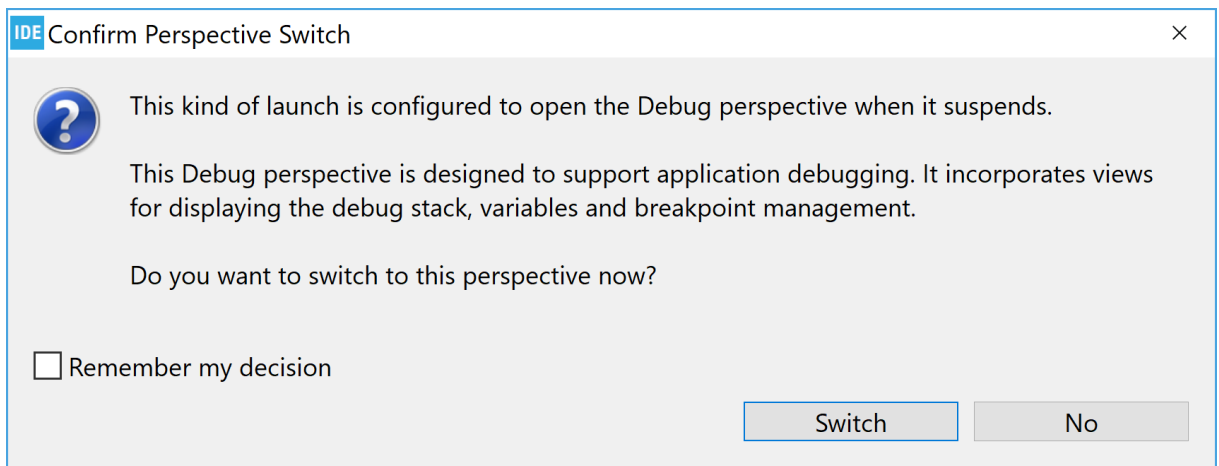
Figure 142. Debug configurations



Select in the left pane the debug configuration to use. Press the **[Debug]** button to start a debug session if all debug configurations have been made. The project is built if file updates are made, but the building depends on the debug configuration.

STM32CubeIDE launches the debugger and the following dialog is opened.

Figure 143. Confirm perspective switch

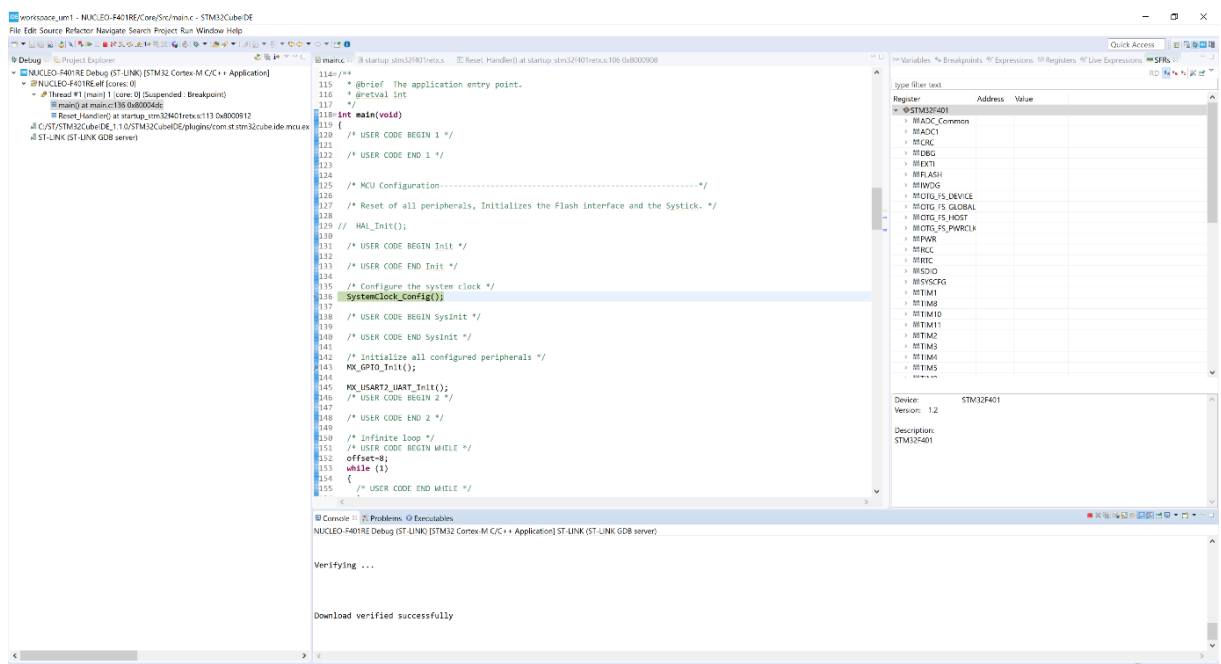


It is recommended to enable **[Remember my decision]** and press **[Switch]**. It opens the *Debug* perspective, which provides a number of views and windows suitable for debugging.

3.5.2 Debug perspective and views

The *Debug* perspective contains menus, toolbars and views frequently used during debugging.

Figure 144. Debug perspective



The most important views opened by default in the *Debug* perspective are:

- The *Debug* view, which displays the program under debug. It also lists threads and offers the possibility to navigate in the thread by selecting a line in threads.
- The *Editor* view, which displays the program file. It is possible to set break points and follow program execution in the file. It is also possible to hover the cursor over a variable to display its current value. The features available during file edition are available also during debug, such as opening the declaration of a function and others.

- The *Variables* view, which displays local variables automatically with their current value when the program is not running.
- The *Breakpoints* view, which displays current breakpoints. It is possible to disable and enable breakpoints in the list. The *Breakpoints* view also contains a toolbar, which, for instance, enables to remove breakpoints, and skip breakpoints with one click on the **[Skip All Breakpoints]** icon.
- The *Expressions* view, which is used to add and view expressions. An expression may be a single global variable, structure, or an expression calculating some variables. The values are only updated when the program is stopped. It is possible to select a global variable in the *Editor* and drag it over to the *Expressions* view instead of entering the variable name.
- The *Registers* view, which displays the debugged device current values. The values are only updated when the program is stopped.
- The *Live Expressions* view, which displays expression values sampled and updated regularly during program execution. The view allows the creation of mathematical expressions that are evaluated automatically, such as $(Index*4+Offset)$. The *Live Expressions* view update requires that live expressions are enabled in the debug configuration. Refer to [Section 3.6.1 Live Expressions view](#) for details.
- The *SFRs* view, which displays the Special Function Registers in the debugged device. Refer to [Section 5 Special Function Registers \(SFRs\)](#) for details.
- The *Console* view, which displays different console outputs. By default, the console output from the GDB server log is displayed. It is possible to change the console log by pressing the **[Display Selected Console]** icon to the right of the *Console* view.

Other views are also useful during debug, among which:

- The *Debugger Console* view, which can be used if there is a need to manually enter GDB commands. The easiest way to open the *Debugger Console* view is to use the **[Quick Access]** field and enter `Debugger` in this field. It lists choices containing the *Debugger Console* view. Select it to open the view. GDB can be entered in the *Debugger Console* view.

For instance, to display 16 words of memory from address `0x800 0000`, enter the GDB command `x /16 0x8000000`.

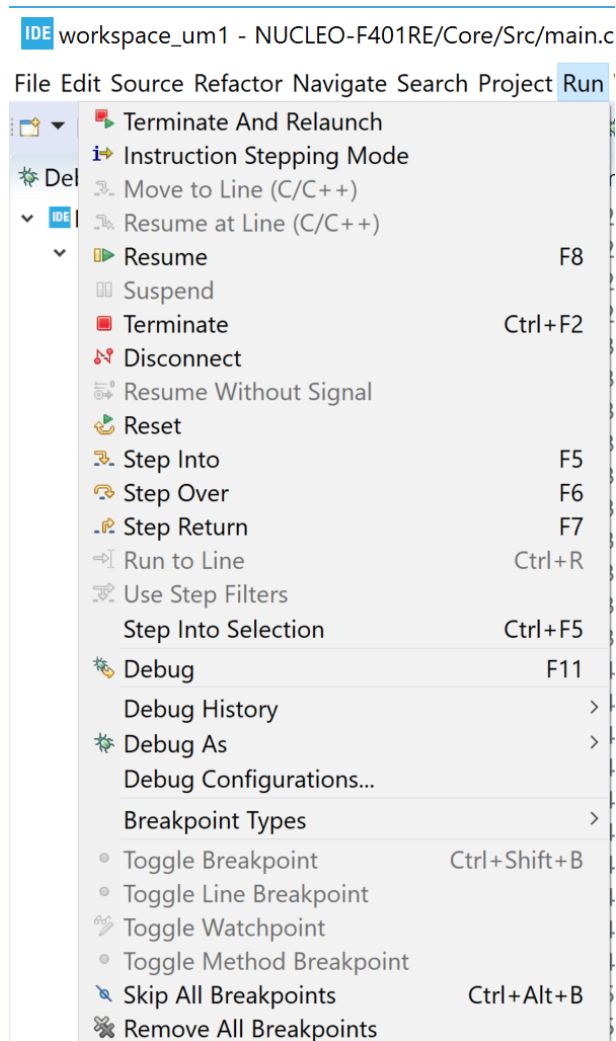
```
x /16 0x8000000
0x8000000: 0x20018000 0x080008b1 0x080007e9 0x080007f7
0x8000010: 0x080007fd 0x08000803 0x08000809 0x00000000
0x8000020: 0x00000000 0x00000000 0x00000000 0x0800080f
0x8000030: 0x0800081d 0x00000000 0x0800082b 0x08000839
```

- The *Memory* and *Memory Browser* views, which can be used to display and update memory data.
- The *Disassembly* view, which is used to view and step in the assembly code.
- The *SWV* views. Refer to [Section 4 Debug with Serial Wire Viewer tracing \(SWV\)](#) for details.
- The *Fault Analyzer* view. Refer to [Section 7 Fault Analyzer](#) for details.

3.5.3 Main controls for debugging

The **[Run]** menu in the *Debug* perspective contains a number of execution control functions.

Figure 145. [Run] menu



Alternatively, the *Debug* perspective toolbar has the following main debug control icons.

Figure 146. *Debug* toolbar



These icons are used for the following purpose, from left to right:

- Reset the device and restart the debug session
- Skip all breakpoints (**Ctrl+Alt+B**)
- Terminate and relaunch
- Resume (**F8**)
- Suspend
- Terminate (**Ctrl+F2**)
- Disconnect
- Step into (**F5**)
- Step over (**F6**)
- Step return (**F7**)
- Instruction stepping mode (assembler stepping)

Press [**Terminate and relaunch**] to terminate the current debug session, build a new program if the source code is modified, and relaunch the debug session.

When pressing [**Instruction stepping mode**], the *Disassembly* view is opened and further stepping uses assembler instruction stepping level. Press [**Instruction stepping mode**] again to toggle back to C/C++ level stepping.

3.5.4 Run, start and stop a program

Use the toolbar icons as follows to run, step, or stop the program:

- Run the program with the [**Resume**] toolbar icon ([**F8**])
- Step into a function with the [**Step into**] toolbar icon ([**F5**])
- Step over a function with the [**Step over**] toolbar icon ([**F6**])
- Step until return from a function with the [**Step return**] toolbar icon ([**F7**])
- Abort running program with the [**Suspend**] toolbar icon

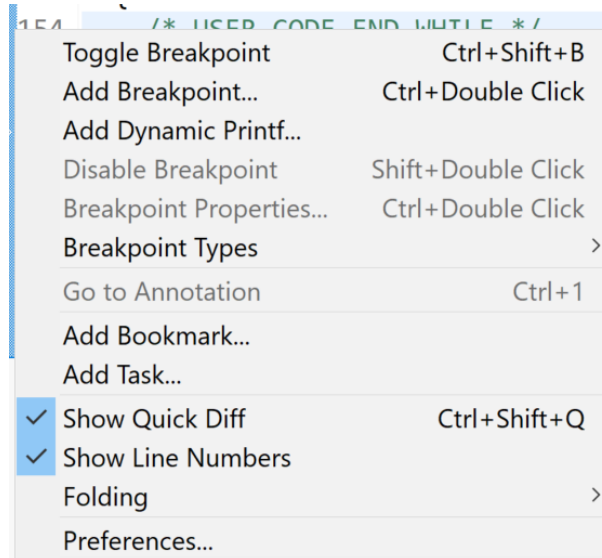
3.5.5 Set breakpoints

It is common during a debug session to set breakpoints and let the code execute until it reaches a breakpoint.

3.5.5.1 Standard breakpoint

A standard code breakpoint at a source code line can easily be inserted by double-clicking in the left editor margin, or by right-clicking in the left margin of the C/C++ source code editor. A context menu is proposed in the latter case.

Figure 147. Debug breakpoint



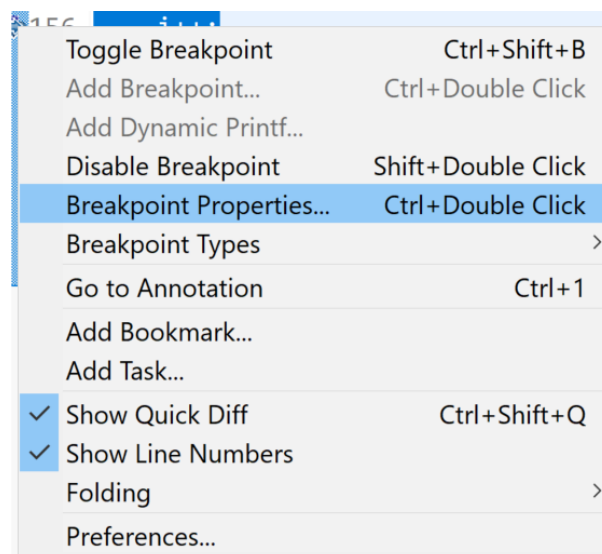
Select the **[Toggle Breakpoint]** menu command to set or remove a breakpoint at the corresponding source code line.

3.5.5.2 Conditional breakpoint

When setting a standard breakpoint at a source code line, the program breaks each time it reaches this line. If that is not the desired behaviour, a condition can be set on the breakpoint that regulates if the program should actually break or not on that breakpoint.

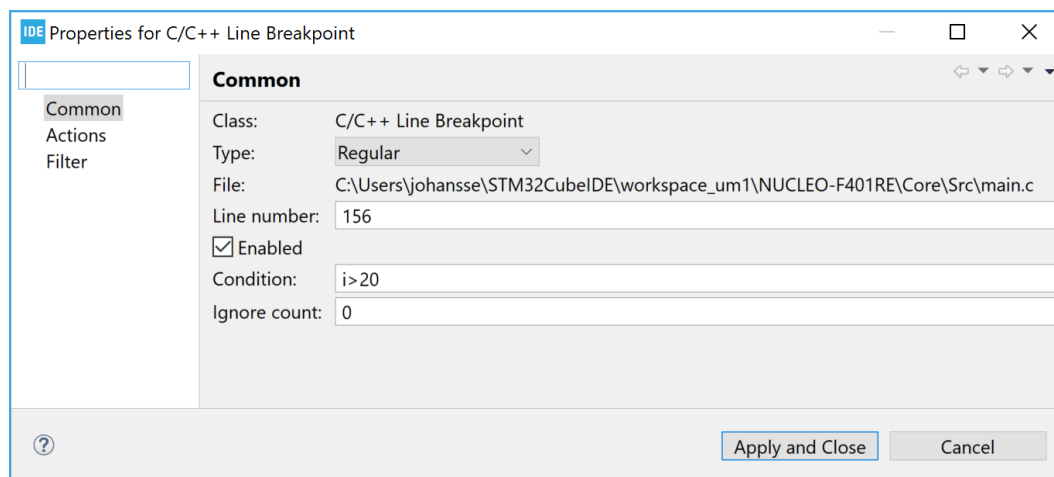
Update breakpoint properties with a right-click on the breakpoint icon visible left of the editor on a line with breakpoint set. The **[Breakpoint Properties]** can also be opened from the *Breakpoints* view.

Figure 148. Breakpoint properties



Select [**Breakpoint Properties...**]. The following window opens. In the example illustrated below, `i>20` is entered as a condition.

Figure 149. Conditional breakpoint



With the condition above set, the program breaks each time the line is executed, then GDB tests the condition and restarts execution if the variable `i` is not greater than `20`. It takes some time for GDB to evaluate the condition.

The conditions are written in C-style. It is therefore possible to write expressions such as `i%2==0` to set more complex conditions.

3.5.6 Attach to running target

It is possible to connect *STM32CubeIDE* and a debugger via JTAG/SWD to the embedded target without performing a reset. This approach is useful when trying to resolve problems that occur at rare occasions. Finding the root cause of the problem in case of a CPU crash is further simplified by learning how to use the *Fault Analyzer* view (refer to [Section 7 Fault Analyzer](#)).

Before trying this approach, consider whether halting the application in the wrong state could potentially harm the hardware (for instance in the case of a motor controller application). This is because when GDB connects to the target, the CPU is halted. This behaviour cannot be modified.

The following three or four steps are needed to update the debug configuration and to attach to running target:

1. Modify the debug configuration to attach to the running target
2. Connect the debug probe to the embedded target
3. Start a debug session using the modified debug configuration
4. Optionally, analyze the CPU fault condition with the *Fault Analyzer* tool (refer to [Fault Analyzer](#))

Step 1: Modify the debug configuration

The default generated debug configurations in *STM32CubeIDE* contains settings to reset the device and download new program, and sets a breakpoint at `main`. This is not of any use when connecting to a running system which may, or may not, have crashed.

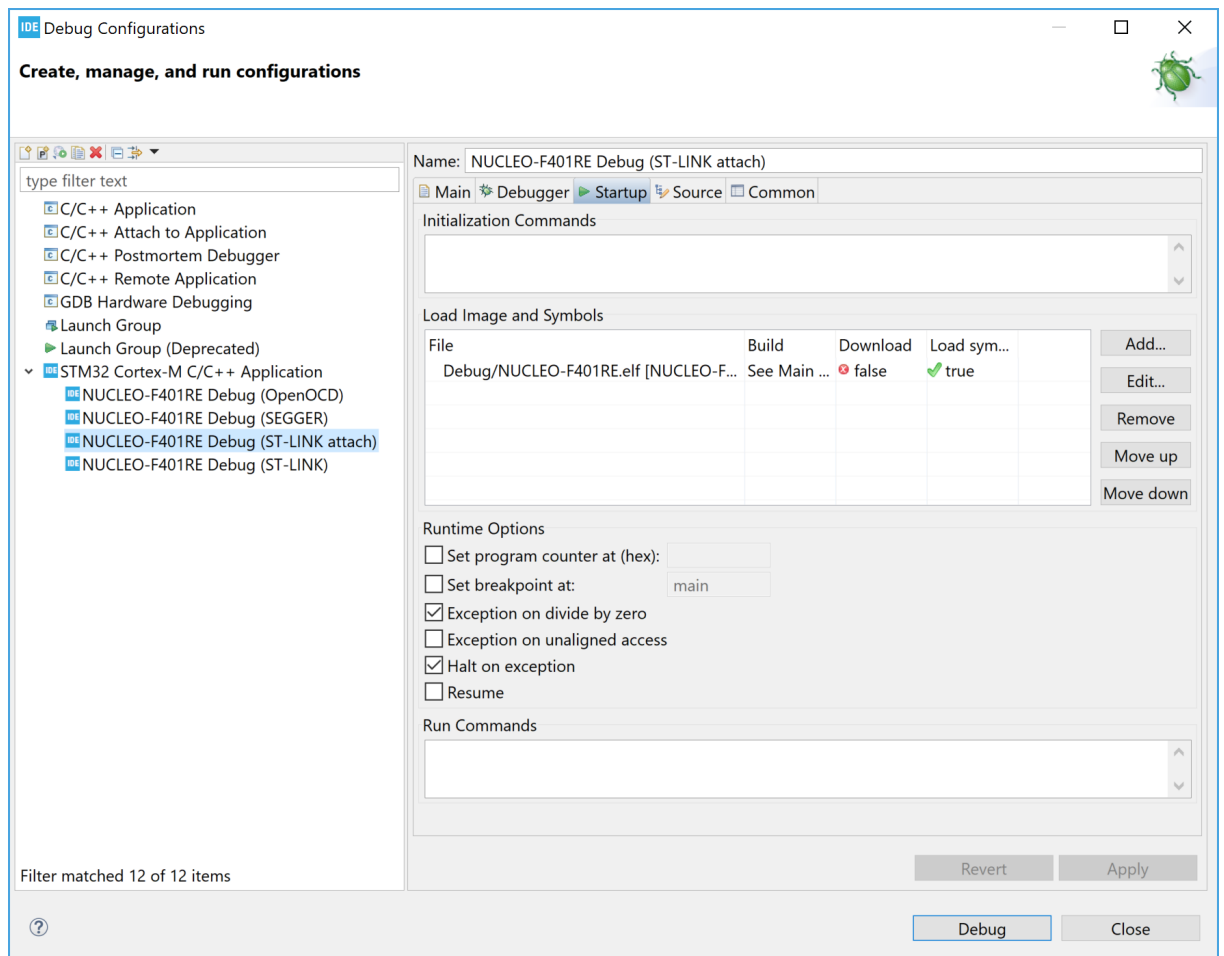
In order to create a modified debug configuration, perform these steps:

1. Open the *Debug Configurations* dialog.
2. In the left frame of the *Debug Configurations* dialog, select the debug configuration associated to the project to debug and make a copy of this by right-clicking it and selecting [**Duplicate**].
3. Give the duplicate debug configuration a name.
4. Update the *Debugger* tab in *Debug Configurations*:
 - When using ST-LINK GDB server and OpenOCD, select [**None**] as [**Reset behaviour**].
 - When using SEGGER J-Link GDB server, select [**None**] as [**Reset strategy**].

5. Change needed/recommended in the *Startup* tab of *Debug Configurations* for both ST-LINK GDB server and SEGGER J-Link GDB server:
 - Disable file **[Download]** in **[Load Image and Symbols]**.
 - Disable **[Set program counter at (hex)]**.
 - Disable **[Set breakpoint at]**.
 - **[Exception on divide by zero]** and **[Exception on unaligned access]** can be disabled or enabled.
 - Disable **[Resume]**.

If the **[Resume]** is enabled, the debugger stops the target during connection and, after a short period of time, sends a `continue` command.

Figure 150. Startup tab attach



Step 2: Connect ST-LINK or SEGGER J-Link to the embedded target

Connect first ST-LINK or the SEGGER J-link to the computer. Then connect it to the embedded target. No reset is issued.

Step 3: Start a debug session using the modified debug configuration

Important:

Do not launch the debug session using the wrong debug configuration, which may reprogram and reset the target. Use [Run]>[Debug Configurations...], select the modified debug configuration in the left frame, and click [Debug]. This is the safest way to launch a debug session with full control of the debug configuration applied and prevents from a potential reset.

The debugger is now connected to the embedded target, which is automatically halted. At this point, different status registers and variables can be investigated in the application. If the CPU has crashed, the *Fault Analyzer* can be used to get a better understanding of the root causes.

3.5.7 Restart or terminate debugging

This section presents various ways to restart and stop a debug session.

3.5.7.1 Restart

During debugging, it is sometimes needed to restart the program to examine more carefully problems observed during debug. In such case, restart the program using the [Reset the chip and restart debug session] toolbar button or [Run]>[Restart] menu command. This resets the device, and starts the program if [Resume] is enabled in the debug configuration.

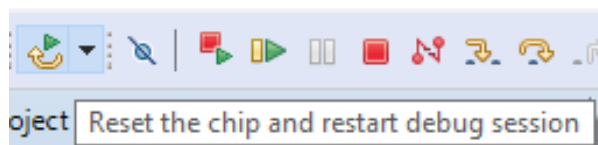
Note:

To make restart work, the interrupt vector must be configured and used with the hardware reset. This is usually the case for STM32 programs located in the Flash memory. However, if the program is located elsewhere such as in RAM, some manual handling may be needed to make the program start from the expected *Reset_Handler*.

3.5.7.2 Restart configurations

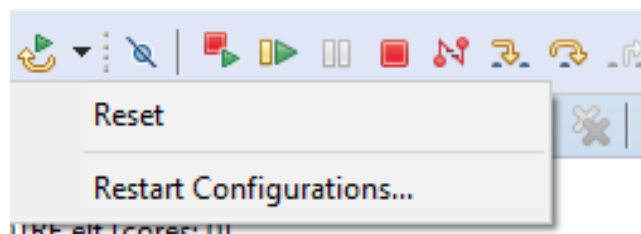
It is possible to create restart configurations defining how the reset and restart of a debug session must be performed. Click on the arrow to the right of the [Reset the chip and restart debug session] toolbar icon.

Figure 151. Reset the chip toolbar



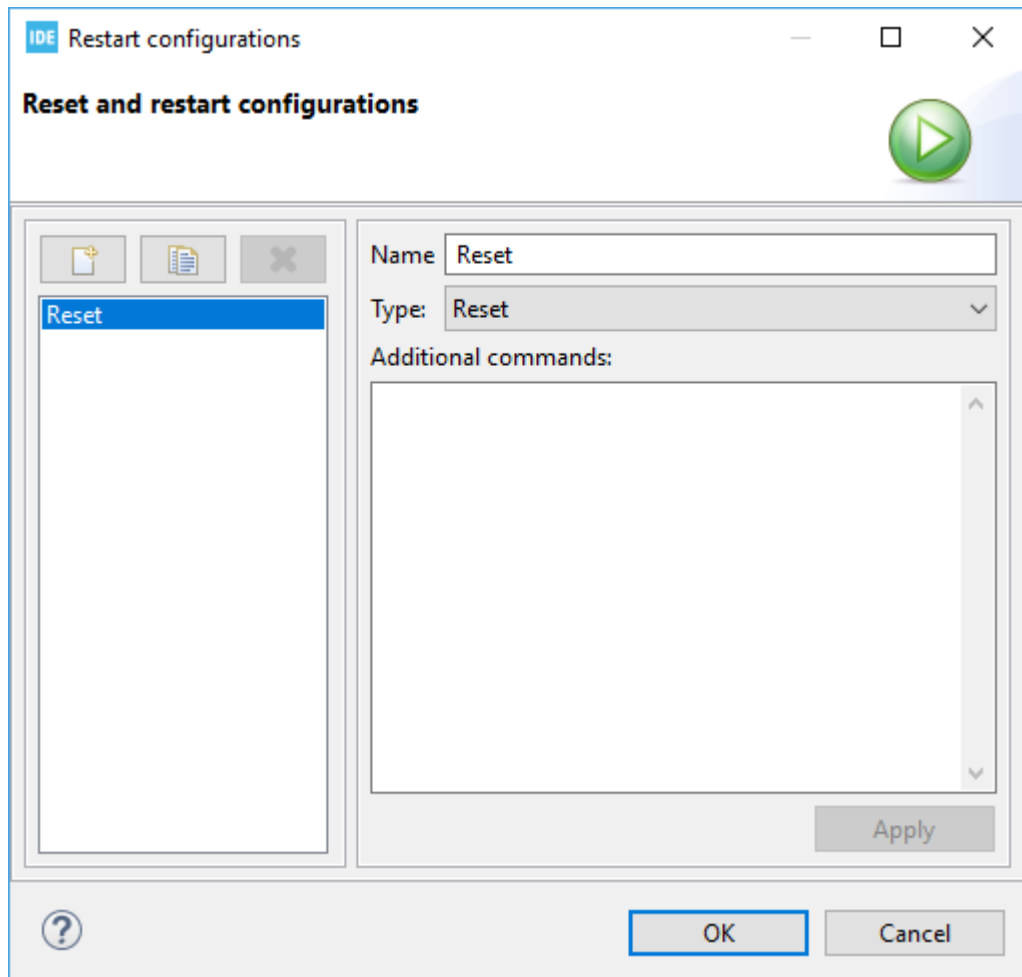
This expands the menu with the [Restart Configurations...] selection.

Figure 152. Restart configurations selection



When [Restart Configurations...] is selected, the restart configurations dialog opens.

Figure 153. Restart configurations dialog



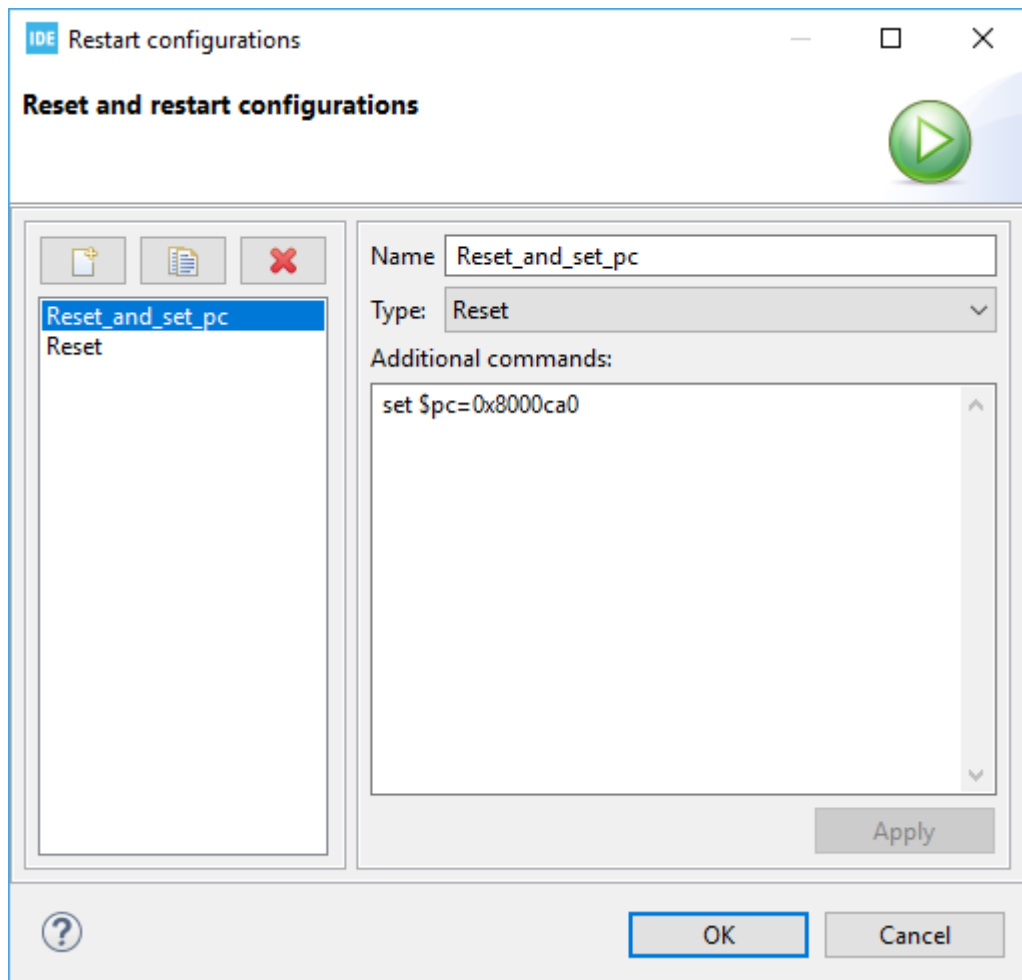
The dialog contains a left and right pane:

- The left pane is used to select and create new restart configuration, duplicate an existing restart configuration, and delete the selected restart configuration. The default restart configurations cannot be deleted.
- The right pane is used to set [Name] and select the [Type] of reset to be used for the selected configuration. It is also possible to add additional commands to be used with the reset.

Press [Apply] to save a setting.

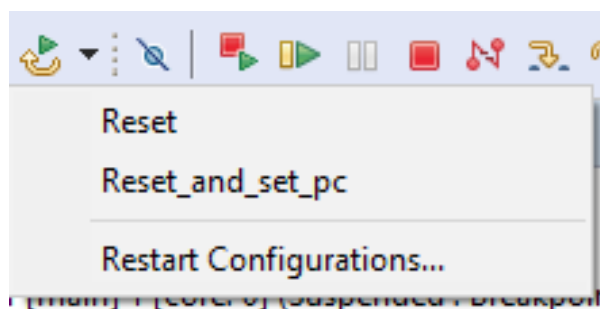
Figure 154 shows a setting where a new restart configuration is created, which contains an additional command to set pc to 0x8000ca0.

Figure 154. Restart configurations dialog with additional command



When several reset configurations are defined, they appear in the toolbar dropdown menu in order of use. Select the desired one to perform a reset.

Figure 155. Select restart configuration



3.5.7.3 Terminate

The most common way to stop a debug session is by clicking the **[Terminate]** toolbar button. It is also possible to stop the debug session with the **[Run]>[Terminate]** menu. When the debug session is stopped, STM32CubeIDE switches automatically to the C/C++ perspective.

3.5.7.4 Terminate and relaunch

Use the [Terminate And Relaunch] toolbar button if changes in the source code have been made during the debug session. Menu command [Run]>[Terminate And Relaunch] can also be used for this purpose. This stops the debug session, rebuild the program, and relaunches a debug session with the new program loaded.

3.6 Debug features

3.6.1 Live Expressions view

The *Live Expressions* view in STM32CubeIDE works very much like the *Expression* view with the exceptions that all the expressions are sampled live during debug execution. The sampling speed is determined by the number of expressions being sampled. An increased number of expressions being sampled results in a slower sample rate.

The view displays many different types of global variables. The view also allows users to create mathematical expressions that are evaluated automatically, such as $(i * 4 + offset)$.

Figure 156. Live Expressions

Expression	Type	Value
(x)= i	uint32_t	2
(x)= offset	uint32_t	8
(x)= i*4+offset	unsigned int	16
+ Add new expression		

The view can parse complicated data types and display complex data types like C-language structures. Only one format of numbers is used at the same time. To change this format, use the dropdown menu.

Figure 157. Live expressions number format

Expression	Type	Value
(x)= i	uint32_t	2
(x)= offset	uint32_t	8
(x)= i*4+offset	unsigned int	16
+ Add new expression		

The variable values can be changed on the fly in the *Live Expressions* view while the program is running. Select the variable and change its value. It requires that only single variable name is used in the expression, and that no calculation involved.

Note: To be able to use the *Live Expressions* view during debug, the live expression mechanism must be enabled during startup.

3.6.2 Shared ST-LINK

In the *Debugger* tab in *Debug Configurations* for ST-LINK GDB server and OpenOCD, a selection enables shared ST-LINK. When shared ST-LINK is enabled, the communication to ST-LINK goes via the ST-LINK server. The ST-LINK server makes it possible for several programs to access the same ST-LINK when shared ST-LINK is enabled.

STM32CubeProgrammer (STM32CubeProg) also contains a configuration for shared ST-LINK. This means that when shared ST-LINK is enabled in the debug configuration in STM32CubeIDE, it is possible to debug a program and let STM32CubeProgrammer access and read the device Flash memory and RAM at the same time.

3.6.3 Debug multiple boards

Debugging with multiple boards is possible using two ST-LINK or SEGGER J-Link probes at the same time. Connected to two different microcontrollers, both probes are connected to one PC on different USB ports. In this section, let us suppose that two different boards/microcontrollers are used: HW_A and HW_B.

It is possible to run one instance of STM32CubeIDE containing one project for HW_A and one project for HW_B.

The default port to be used is:

- 61234 for ST-LINK GDB server
- 3333 for OpenOCD
- 2331 SEGGER J-Link

This is presented in the *Debugger* tab in the *Debug Configurations* dialog. The port number must be changed for one of the projects to use another port, such as port 61244.

The debug configuration can use GDB connection selection [**Autostart local GDB server**]. Note that when debugging multiple boards, two or more debug probes are connected to the PC; the correct serial number must be selected for each debug configuration.

When the debug configurations has been configured for both projects so that each board is associated to a specific probe, it is time to test and debug each board individually first. When it is confirmed that this is working, the debug of both targets at the same time can be started as follow:

1. Start to debug HW_A.
2. The perspective switches automatically to the *Debug* perspective in STM32CubeIDE when a debug session for HW_A is started.
3. Switch to the C/C++ perspective.
4. Select the project for HW_B and start debugging it. The *Debug* perspective opens again.
5. There are two application stacks/nodes in the *Debug* view, one for each project. When changing the selected node in the *Debug* view, the related editor, variable view and others are updated to present information associated to the selected project.

It is also possible to start the GDB servers manually: select [**Connect to remote GDB server**] in the debug configuration. In such case, make sure that the GDB servers are started with parameters defining the individual ports and serial numbers to be used, and that the corresponding port numbers are used in the *Debug Configurations* dialog for each project.

Below is an example using SEGGER J-Link GDB server connecting to SEGGER J-Link, with port=2341 and S/N=123456789:

```
>JLinkGDBServerCL.exe -port 2341 -if SWD -select usb=123456789
```

Information on command-line parameters to be used when starting the GDB servers manually are provided in the GDB server manuals available from the *Information Center*.

3.6.4 STM32H7 multicore debugging

Information about how to use STM32H7 multicore devices in STM32CubeIDE is available in [ST-09].

3.6.5 STM32MP1 debugging

Information about how to use STM32MP1 devices in STM32CubeIDE is available in [ST-08].

Users are advised to keep updated with the documentation evolution of the STM32MP1 Series at www.st.com/en/microcontrollers-microprocessors/stm32mp1-series.

3.6.6 STM32L5 debugging

Information about how to use STM32L5 devices with TrustZone® in STM32CubeIDE is available in [ST-10].

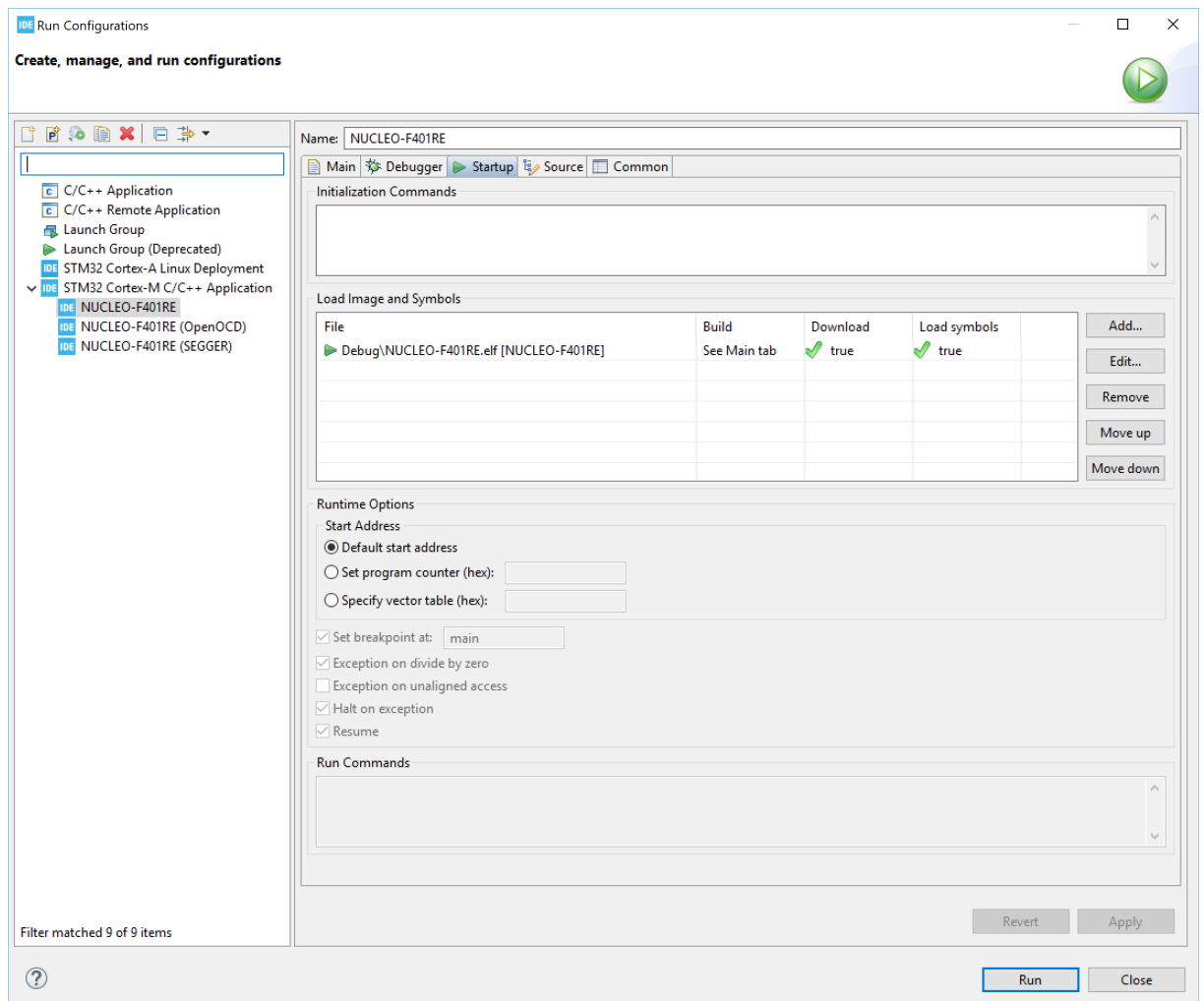
3.7 Run configurations

It is possible to create run configurations to download applications and reset the target without launching a full debug session. The *Run Configurations* dialog is similar to the *Debug Configurations* dialog, however disabled widgets in the lower part of the *Startup* tab are not performed. When running a run configuration, the specified program is flashed but, after program counter is set, the program execution is started in target and the "run" session in STM32CubeIDE is closed.

To create a run configuration for the project, right-click on the project name in the *Project Explorer* view and select **[Run As]>[STM32 Cortex-M C/C++ Application]**.

Another way to create a run configuration is to select the project name in the *Project Explorer* view and use the menu **[Run]>[Run As]>[STM32 Cortex-M C/C++ Application]**.

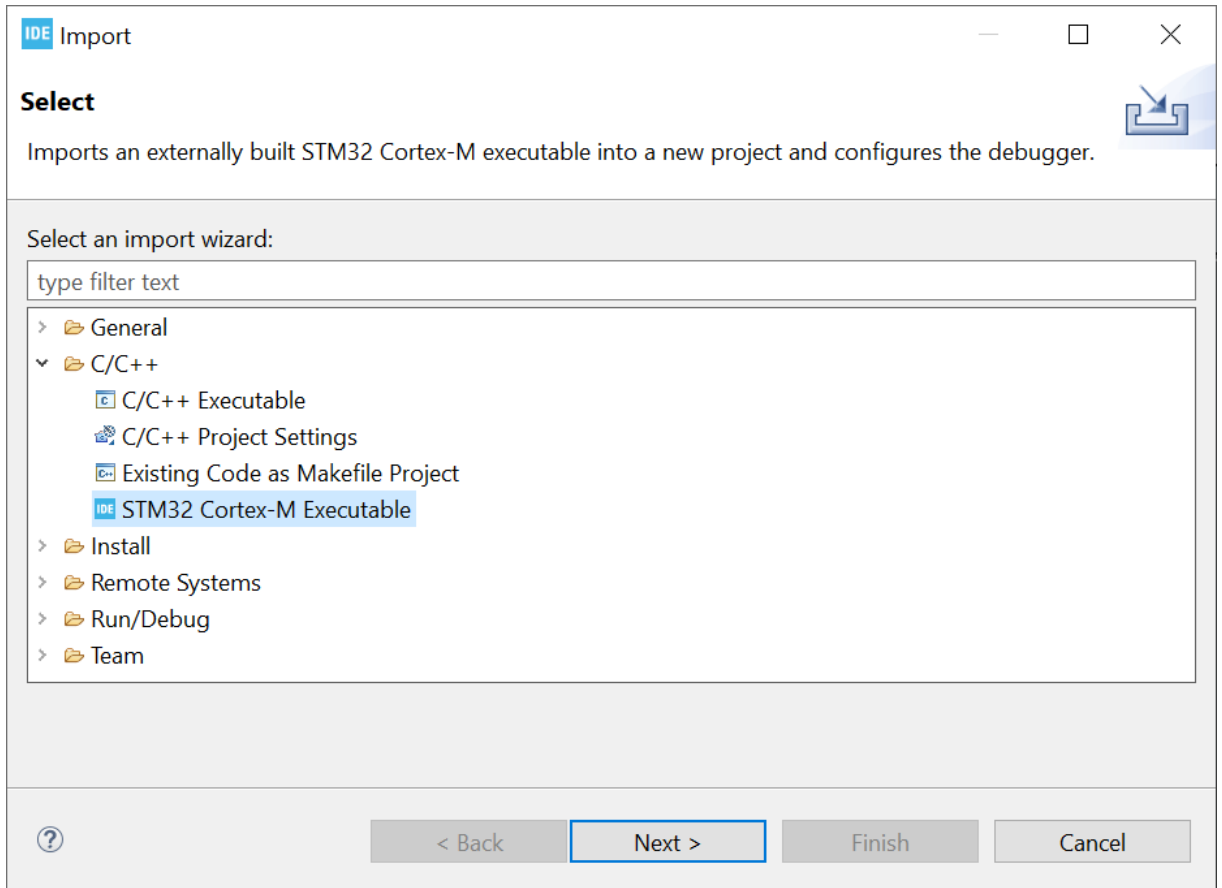
Figure 158. Run configurations startup tab



3.8 Import STM32 Cortex®-M executable

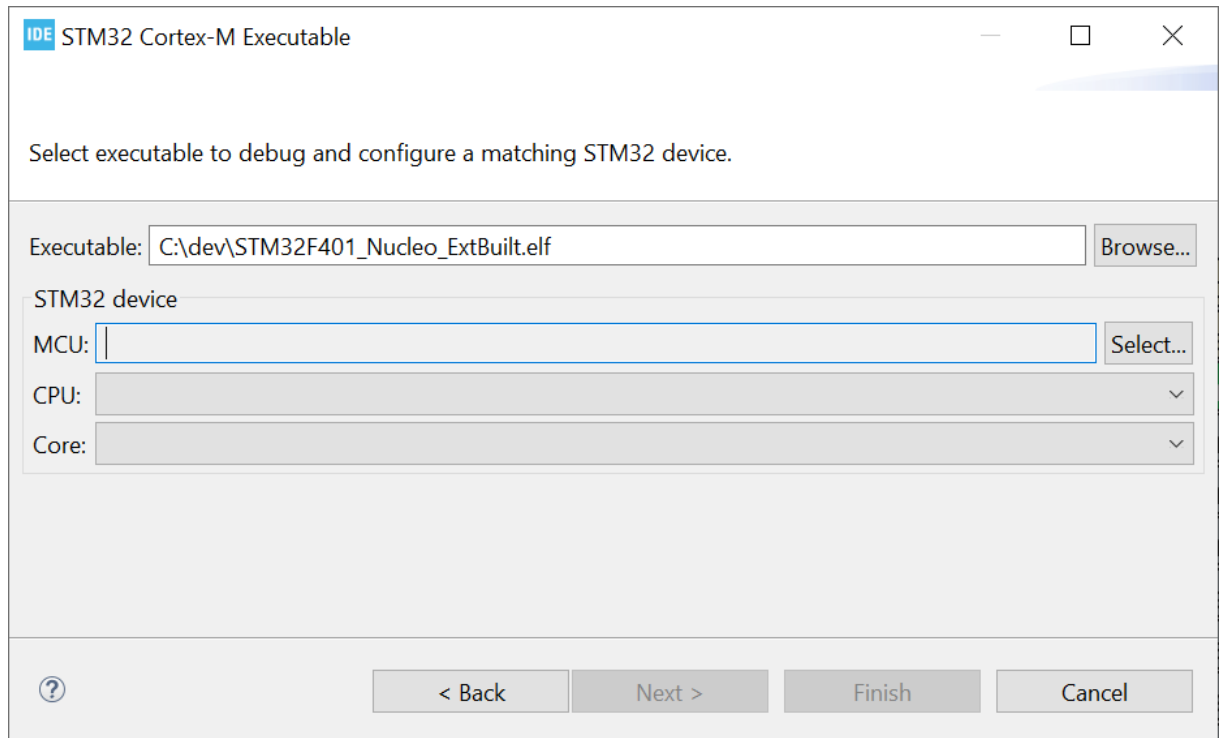
Use menu [File]>[Import...] to open the *Import* dialog.

Figure 159. Cortex®-M executable import dialog



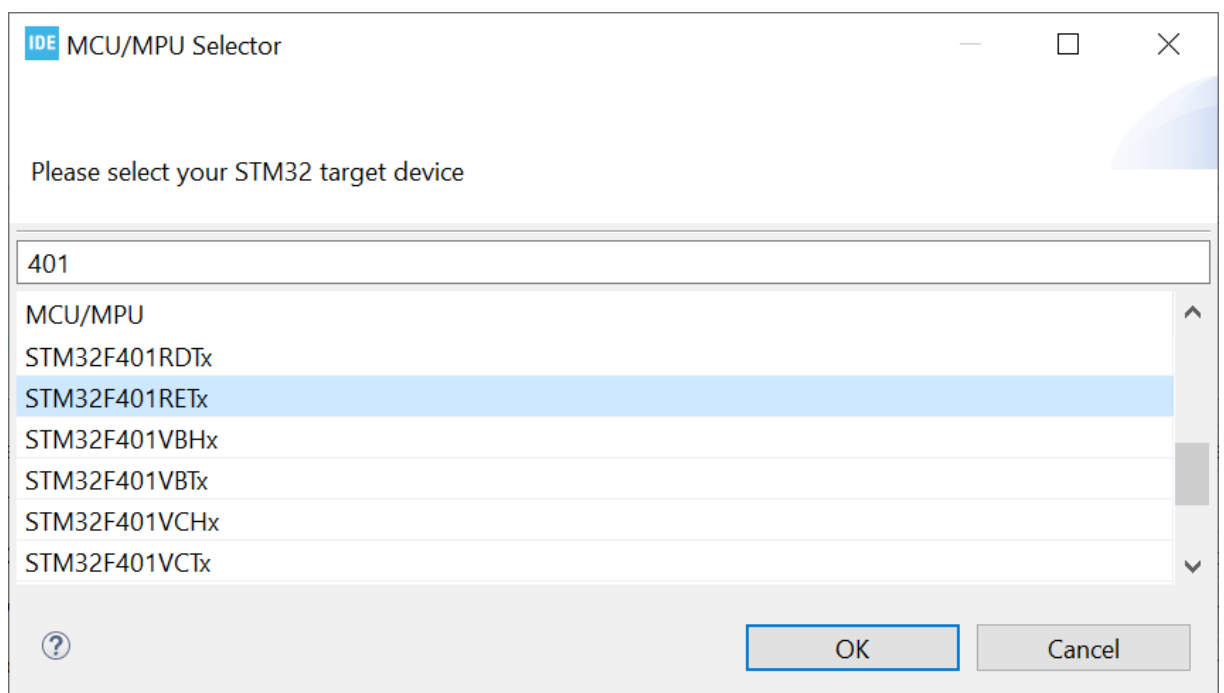
Select [**STM32 Cortex-M Executable**] and press [**Next >**].

Figure 160. STM32 Cortex®-M executable dialog



Use the [**Browse...**] button and select the `elf` file to import. When the `elf` file is selected, the STM32 device must be selected manually so that STM32CubeIDE can be used for debugging. Press [**Select...**] to open the *MCU/MPU Selector* dialog.

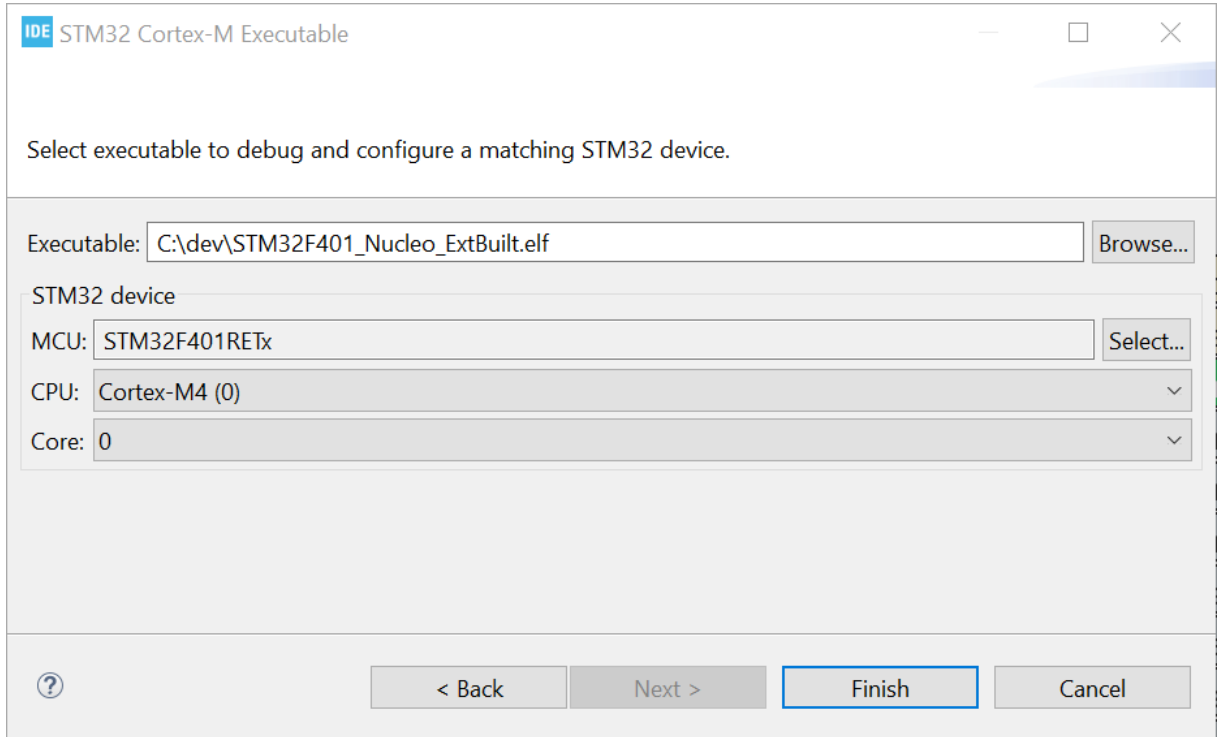
Figure 161. STM32 Cortex®-M executable MCU/MPU selection



Select the microcontroller or microprocessor to be used. The search field can be used to find the device. Press [OK] once the device is selected.

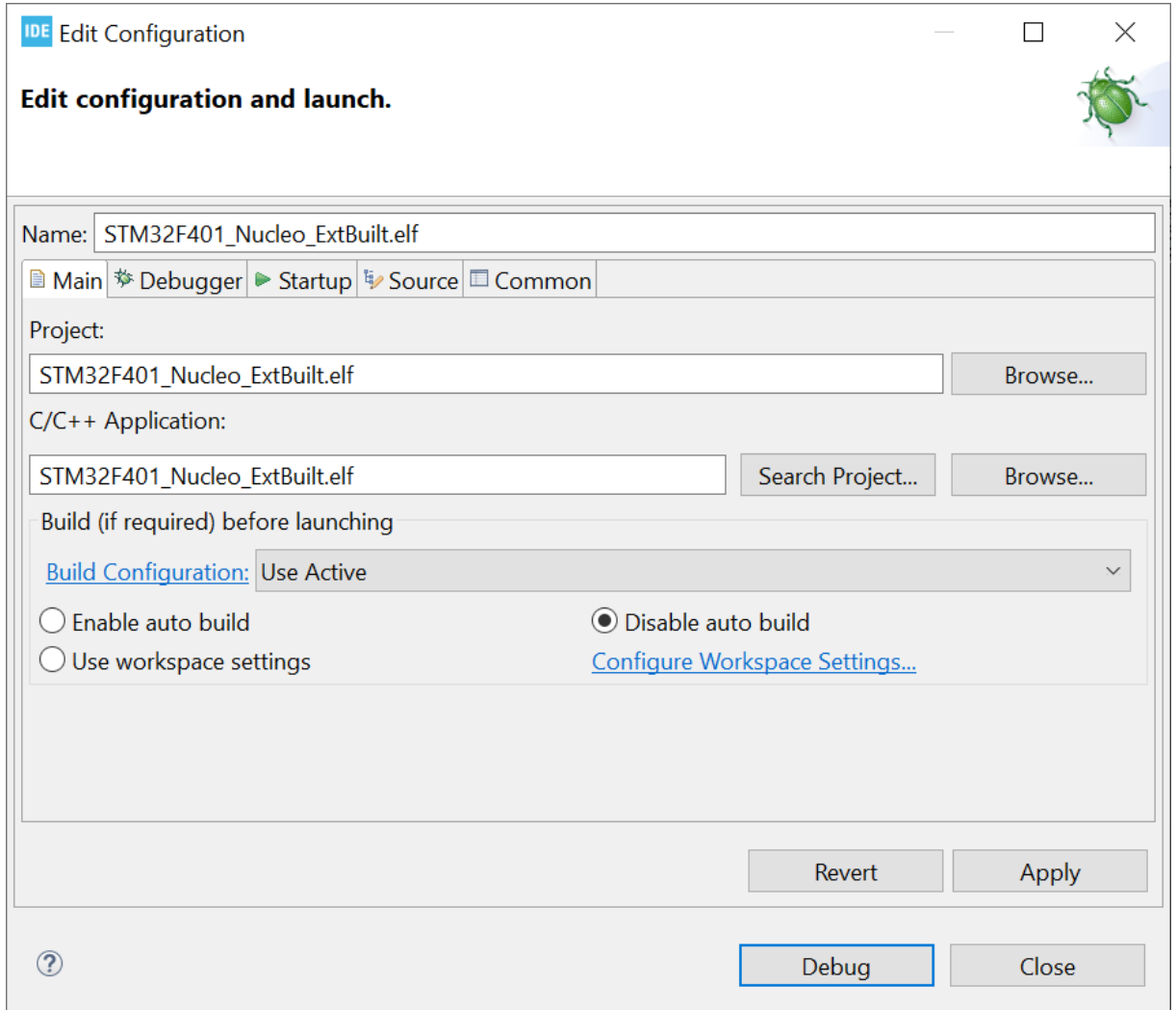
As a result, the CPU and core are presented in the dialog.

Figure 162. STM32 Cortex[®]-M CPU and core



Press **[Finish]** and the debug configuration dialog automatically opens.

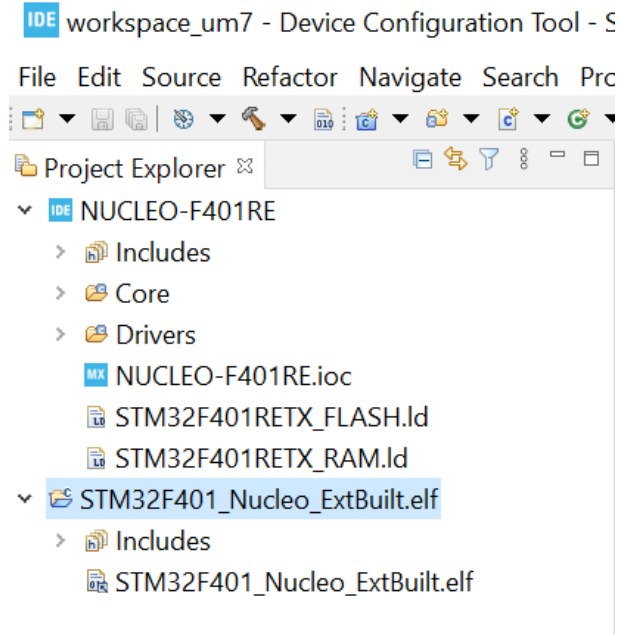
Figure 163. Cortex®-M debug configuration for imported project



The debug configuration can then be set up in similar way as with any other STM32CubeIDE project. Once the configuration is completed, press **[Debug]** to start a debug session.

The imported project is displayed in the *Project Explorer* view.

Figure 164. Project explorer view with imported project



4 Debug with Serial Wire Viewer tracing (SWV)

4.1 Introduction to SWV and ITM

This section provides information on how to use Serial Wire Viewer tracing (SWV) in [STM32CubeIDE](#).

System analysis and real-time tracing in STM32 requires a number of interaction technologies: Serial Wire Viewer (SWV), Serial Wire Debug (SWD), Instrumentation Trace Macrocell (ITM) and Serial Wire Output (SWO). These technologies are part of the Arm® CoreSight™ debugger technology. They are explained below.

Serial Wire Debug (SWD) is a debug port similar to JTAG. It provides the same debug capabilities (run, stop on breakpoints, single-step) but with fewer pins. It replaces the JTAG connector with a 2-pin interface (one clock pin and one bi-directional data pin). The SWD port alone does not allow real-time tracing.

The Serial Wire Output (SWO) pin can be used in combination with SWD. It is used by the processor to emit real-time trace data, thus extending the two SWD pins with a third pin. The combination of the two SWD pins and SWO pin enables Serial Wire Viewer (SWV) real-time tracing in compatible Arm® processors.

Beware that, SWO being just one pin, it is easy to set a configuration that produces more data than the SWO is able to send.

The Serial Wire Viewer (SWV) is a real-time trace technology that uses the Serial Wire Debug (SWD) port and the Serial Wire Output (SWO) pin. The Serial Wire Viewer provides advanced system analysis and real-time tracing without the need to halt the processor to extract the debug information.

Serial Wire Viewer (SWV) provides the following types of target information:

- Event notification on data reading and writing
- Event notification on exception entry and exit
- Event counters
- Timestamp and CPU cycle information, which can be used for program statistical profiling

The Instrumentation Trace Macrocell (ITM) enables applications to write arbitrary data to the SWO pin, which can be interpreted and visualized in the debugger. For example, ITM can be used to redirect `printf()` output to a SWV console view in the debugger. The standard is to use port 0 for this purpose.

The ITM port has 32 channels. Writing different types of data to different ITM channels allows the debugger to interpret or visualize the data on various channels differently.

Writing a byte to the ITM port takes only one write cycle, thus taking almost no execution time from the application logic.

Based on SWV, and ITM trace data, STM32CubeIDE can provide advanced debugger capabilities with special SWV views.

Note: Arm® does not include SWV/ITM in Cortex®-M0 or Cortex®-M0+ cores. Therefore, STM32 devices based on these cores, such as STM32L053 microcontrollers, do not support SWV/ITM.

4.2 SWV debugging

To debug and use the Serial Wire Viewer (SWV) in STM32CubeIDE, the JTAG probe and the GDB server must support SWV. The board must also support SWD, and the SWO pin needs to be available and connected to the JTAG probe.

The following sections describe the process to create a debug configuration, SWV settings configuration, and how to use SWV tracing in a debug session.

4.2.1 SWV debug configuration

Step 1: Open the *Debug Configurations* dialog

Use for instance menu **[Run]>[Debug Configurations...]** and select the STM32 Cortex®-M debug configuration to update.

Step 2: Select the SWD interface

Select the **[SWD]** interface in the *Debug Configurations* dialog.

Step 3: Enable SWV

Enable **[SWV]** in the *Debug Configurations* dialog.

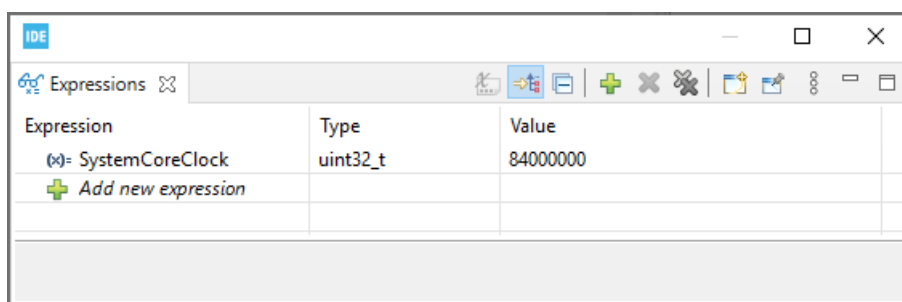
Step 4: Enter the core clock frequency

Enter the **[Core Clock]** frequency in the *Debug Configurations* dialog. This must correspond to the value set by the application program to be executed.

Usually, the core clock setting is stored in the `SystemCoreClock` variable when using projects imported from STM32 firmware examples or created with **STM32CubeMX**. One method to inspect the core clock value is to start a debug session and add the `SystemCoreClock` variable to the *Expressions* view. Make sure that the system core clock is configured by the application before reading the value.

If the `SystemCoreClock` is not updated, change the program and add a call to the function `SystemCoreClockUpdate()`. Rebuild the program, restart debugging and inspect the `SystemCoreClock` value again.

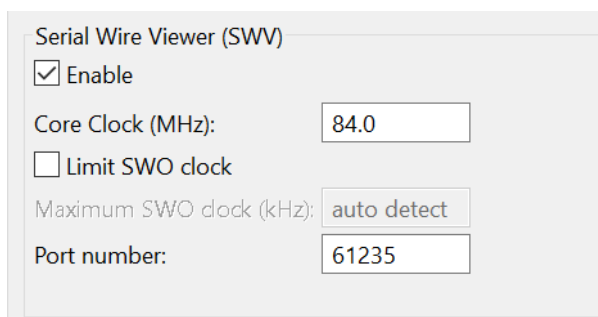
Figure 165. SWV core clock



Step 5: Enter the SWO clock frequency

The **[Serial Wire Viewer (SWV)]** selections in the *Debug Configurations* dialog can be used only when the **[SWD]** interface is selected. When **[SWV]** is enabled, it is required to configure the **[Clock Settings]**. The **[Core Clock]** must be set to the device speed. The SWO clock is automatically set to the highest possible speed depending on debug probe used and core clock. However, if the debugged hardware does not allow too-high SWO clock speed, it is possible to enable **[Limit SWO clock]** and enter the maximum SWO clock speed in kHz. The SWV **[Port number]** must be set to the port to be used for SWV data communication. The SWV port cannot be set equal to the GDB connection **[Port number]**.

Figure 166. SWV debug configuration



Step 6: Save the configuration

Press **[Apply]** in the *Debug Configurations* dialog to save the configuration.

Step 7: Start a debug session

Press **[Debug]** to start a debug session. Make sure that the probe and board are connected.

Step 8: Possibly suspend the target

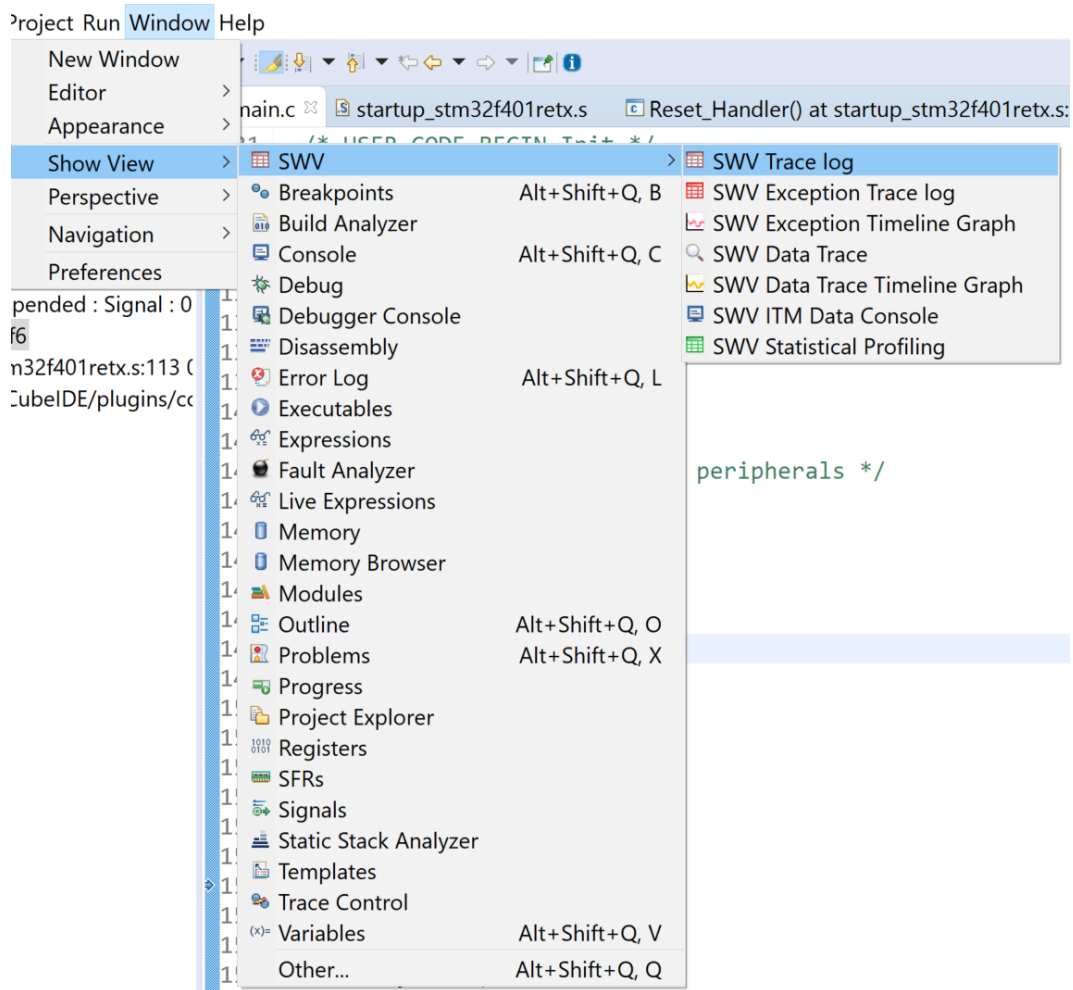
[Suspend] the target if it has not stopped at a breakpoint.

Step 9: Open a SWV view

Open one of the SWV views. For first-time users, it is recommended to open the *SWV Trace log* view because it gives a good overview of incoming SWV packets and how well the tracing is working.

Select the [Window]>[Show View]>[SWV]>[SWV Trace log] menu command to open the *SWV Trace log* view.

Figure 167. SWV show view



Step 10: View the trace log

The *SWV Trace log* view is now visible.

- **[Clock Settings]:** These fields are disabled and only present the values used and configured in the *Debug Configurations* for the debug session. If these values need to be changed, close the debug session and open the *Debug Configurations* to modify them.
- **[Trace Events]:** The following events can be traced.
 - **[CPI]:** Cycles per instruction. For each cycle beyond the first one that an instruction uses, an internal counter is increased with one. The counter (`DWT CPI count`) can count up to 256 and is then set to 0. Each time that happens, one of these packets are sent. This is one aspect of the processors performance and used to calculate instructions per seconds. The lower the value, the better the performance.
 - **[SLEEP]:** Sleep cycles. The number of cycles the CPU is in sleep mode. Counted in `DWT Sleep count` register. Each time the CPU has been in sleep mode for 256 cycles, one of these packets is sent. This is used when debugging for power consumption or waiting for external devices.
 - **[FOLD]:** Folded instructions. A counter for how many instructions are folded (removed). Every 256 instruction folded (taken zero cycles) will receive one of these events. Counted in `DWT Fold count` register.
 Branch folding is a technique where, on the prediction of most branches, the branch instruction is completely removed from the instruction stream presented to the execution pipeline. Branch folding can significantly improve the performance of branches, taking the CPI for branches below 1.
 - **[EXC]:** Exception overhead. The `DWT Exception count` register keeps track of the number of CPU cycles spent in exception overhead. This includes stack operations and returns but not the time spent processing the exception code. When the timer overflows, one of these events is sent. Used to calculate the actual exception handling cost to the program.
 - **[LSU]:** Load Store Unit Cycles. The `DWT LSU count` register counts the total number of cycles the processor is processing an LSU operation beyond the first cycle. When the timer overflows, one of these events is sent.
 With this measurement, it is possible to track the amount of time spent in memory operations.
 - **[EXETRC]:** Trace Exceptions. Whenever an exception occurs, exception entry, exception exit and exception return events are sent. These events can be monitored in the *SWV Exception Trace Log* view. From this view, it is possible to jump to the exception handler code for that exception.
- **[PC Sampling]:** Enabling this starts sampling the Program Counter at some cycle interval. Since the SWO pin has a limited bandwidth, it is not advised to sample too fast. Experiment with the **[Resolution]** (cycles/sample setting) to be able to sample often enough. The results from the sampling are used, among other things, for the *SWV Statistical Profiling* view.
- **[Timestamps]:** Must be enabled to know when an event occurred. The **[Prescaler]** should only be changed as a last effort to reduce overflow packets.
- **[Data Trace]:** It is possible to trace up to four different C variable symbols, or fixed numeric areas of the memory. To do that, enable one comparator and enter the name of the variable or the memory-address to trace. The value of the traced variables can be displayed both in the *Data Trace* and *Data Trace Timeline Graph* views.
- **[ITM Stimulus Ports]:** There are 32 ITM ports available, which can be used by the application. For instance, the CMSIS function `ITM_SendChar` can be used to send characters to port 0 refer to [Section 4.3.5 SWV ITM Data Console and printf redirection](#)). The packets from the ITM ports are displayed in the *SWV ITM Data Console* view.

Note: *It is recommended to limit the amount of data traced. Most STM32 microcontrollers read and write data faster than the maximum SWO pin throughput. Too many trace data result in data overflow, lost packets and possibly corrupt data. For optimum performance, trace only data necessary to the task at hand.*

Overflow while running SWV is an indication that SWV is configured to trace more data than the SWO pin is able to process. In such a case, decrease the amount of data traced.

Enable **[Timestamps]** to use any of the timeline views in STM32CubeIDE. The default **[Prescaler]** is 1. Keep this value, unless problems occur related to SWV packet overflow.

Three examples are provided below for illustrating SWV trace configuration:

- Example 1:** To trace the value of a global variable, enable [**Comparator**] and enter the name of the variable or the memory address to be traced.
 The value of the traced variable is displayed both in the *Data Trace* and *Data Trace Timeline Graph* views.
- Example 2:** To profile program execution, enable [**PC sampling**]. In the beginning, a high value for the [**Cycles/sample**] is recommended.
 The result from the PC sampling is then displayed in the *SWV Statistical Profiling* view.
- Example 3:** To trace the exceptions occurring during program execution, enable [**Trace Event EXETRC: Trace Exceptions**].
 Information about the exceptions is then displayed in the *SWV Exception Trace Log* view.

Step 3: Save the SWV configuration

Click on the [**OK**] button to save the SWV configuration. The configuration is saved together with other debug configurations and remains effective until changed.

4.2.3 SWV tracing

Step 1: Start SWV trace recoding

Press the [**Start/Stop Trace**] toolbar button in one of the SWV views to send the SWV settings to the target board and start the SWV trace recoding. This toolbar button is available in all SWV views. The board does not send any SWV packet until it is properly configured. The SWV configuration must be resent if the configuration registers on the target board are reset. Actual tracing does not start until the target starts to execute.

Figure 171. SWV [Start/Stop Trace] toolbar button



Note: The tracing cannot be configured while the target is running. Pause the debugging before attempting to send a new configuration to the board. Each new or updated configuration must be sent to the board to take effect. The configuration is sent to the board when the [**Start/Stop Trace**] button is pressed.

Step 2: Start the target

Press the [**Resume**] toolbar button on top of the *Debug* perspective to start the target.

Step 3: SWV Trace Log view

SWV packets are displayed in the **SWV Trace Log** view.

Figure 172. SWV Trace Log PC sampling

Index	Type	Data	Cycles	Time(s)	Extra info
10362	PC Sample	0x8000508	169777034	2.021155 s	
10363	PC Sample	0x8000516	169793417	2.021350 s	
10364	PC Sample	0x8000528	169809800	2.021545 s	
10365	PC Sample	0x8000500	169826183	2.021740 s	
10366	PC Sample	0x8000510	169842566	2.021935 s	
10367	PC Sample	0x80004f2	169858949	2.022130 s	
10368	PC Sample	0x8000504	169875332	2.022325 s	
10369	PC Sample	0x8000516	169891715	2.022520 s	
10370	PC Sample	0x8000528	169908098	2.022715 s	

Overflow packets: 0

Step 4: Clear collected SWV data

When the target is not running, the collected SWV data can be cleared by pressing the **[Remove all collected SWV data]** toolbar button. This toolbar button is available in all SWV views.

Figure 173. [Remove all collected SWV data] toolbar button

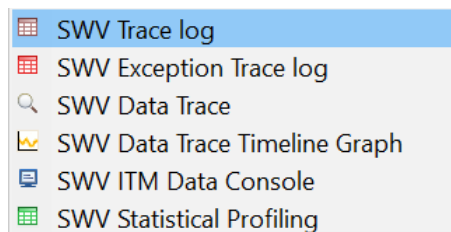


4.3 SWV views

The SWV views that display SWV traces data are:

- *SWV Trace Log*: Lists all incoming SWV packets in a spreadsheet. Useful as a first diagnostic for the trace quality.
- *SWV Exception Trace Log*: The view has two tabs, one is similar to the *SWV Trace Log* view and the other tab displays statistical information about exception events.
- *SWV Data Trace*: Tracks up to four different symbols or areas in the memory.
- *SWV Data Trace Timeline Graph*: A graphical display that shows the distribution of variable values over time.
- *SWV ITM Data Console*: Prints readable text output from the target application. Typically this is done via `printf()` with output redirected to ITM channel 0.
- *SWV Statistical Profiling*: Displays statistics based on the Program Counter (PC) sampling. Shows the amount of execution time spent within various functions.

Figure 174. SWV views selectable from the menu



Note: More than one SWV view may be open at the same time for the simultaneous tracking of various events. The SWV views toolbars contain these usual control icons.

Figure 175. SWV views common toolbar

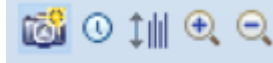


These icons are used for the following purpose, from left to right:

- Configure trace
- Start/Stop trace
- Remove all collected SWV data
- Scroll lock
- Minimize
- Maximize

The SWV graph views toolbars contain these extra control icons.

Figure 176. SWV graph views extra toolbar



These icons are used for the following purpose, from left to right:

- Save graph as image
- Switch between seconds and cycle scale
- Adjust the Y-axis to best fit
- Zoom in
- Zoom out

4.3.1 SWV Trace Log

The *SWV Trace Log* view lists all incoming SWV packets in a spreadsheet. The data in this view can be copied to other applications in CSV format by selecting the rows to copy and type **Ctrl+C**. The copied data can be pasted into another application with the **Ctrl+V** command.

Figure 177. SWV Trace Log PC sampling and exceptions

Index	Type	Data	Cycles	Time(s)	Extra info
25012	PC Sample	0x80004f6	258481871	3.077165 s	
25013	PC Sample	0x8000508	258498254	3.077360 s	
25014	PC Sample	0x8000518	258514637	3.077555 s	
25015	Exception entry	SYSTICK (EXC 15)	258522309	3.077647 s	
25016	Exception exit	SYSTICK (EXC 15)	258522367	3.077647 s	
25017	Exception return	N/A (EXC 0)	258522374	3.077647 s	
25018	PC Sample	0x80004fc	258531017	3.077750 s	
25019	PC Sample	0x800050e	258547400	3.077945 s	
25020	PC Sample	0x800051e	258563783	3.078140 s	
25021	PC Sample	0x80004fa	258580166	3.078335 s	

Overflow packets: 0

The column information in the *SWV Trace Log* view is described in Table 6.

Table 6. SWV Trace Log columns details

Name	Description
Index	The packet ID. Shared with the other SWV packets.
Type	The type of packet (example PC sample, data PC value (comp 1), exceptions, overflow).
Data	The packet data information.
Cycles	The timestamp of the packet in cycles.
Time(s)	The timestamp of the packet in seconds.
Extra info	Optional extra packet information.

4.3.2 SWV Exception Trace Log

The *SWV Exception Trace Log* view is composed of two tabs.

Data tab

The first tab is similar to the *SWV Trace Log* view, but is restricted to exception events. It also provides additional information about the type of event. The data can be copied and pasted into other applications. Each row is linked to the code for the corresponding exception handler. Double-click on the event to open the corresponding interrupt handler source code in the *Editor* view.

Note: Enable **[Trace Event EXETRC: Trace Exceptions]** in the *Serial Wire Viewer settings dialog* to trace exceptions during program execution. Enable **[Timestamps]** to log cycle and time for each interrupt packet.

Figure 178. SWV Exception Trace Log – Data tab

Index	Type	Name	Peripheral	Function	Cycles	Time(s)	Extra info
17629	Exception exit	SYSTICK (EXC 15)		SysTick_Handler()	58204401	692.909536 ms	
17630	Exception return	N/A (EXC 0)			58205926	692.927690 ms	Timestamp delayed. Packet delayed.
17636	Exception entry	SYSTICK (EXC 15)		SysTick_Handler()	58288335	693.908750 ms	
17637	Exception exit	SYSTICK (EXC 15)		SysTick_Handler()	58288393	693.909440 ms	
17638	Exception return	N/A (EXC 0)			58288400	693.909524 ms	
17644	Exception entry	SYSTICK (EXC 15)		SysTick_Handler()	58372327	694.908655 ms	
17645	Exception exit	SYSTICK (EXC 15)		SysTick_Handler()	58372385	694.909345 ms	
17646	Exception return	N/A (EXC 0)			58372392	694.909429 ms	

Overflow packets: 0

The column information in the *SWV Exception Trace Log – Data* tab is described in [Table 7](#).

Table 7. SWV Exception Trace Log – Data columns details

Name	Description
Index	The exception packet ID. Shared with the other SWV packets.
Type	Each exception generates three packets: Exception entry, Exception exit and then an Exception return packet.
Name	The name of the exception. Also the exception or interrupt number.
Peripheral	The peripheral for the exception.
Function	The name of the interrupt handler function for this interrupt. Updated when debug is paused. Is cached during the whole debug session. By double clicking the function, the editor will open that function in the source code.
Cycles	The timestamp for the exception in cycles.
Time(s)	The timestamp for the exception in seconds.
Extra info	Optional extra information about that packet.

Statistics tab

The second tab displays statistical information about exception events. This information may be of great value when optimizing the code. Hypertext links to exception handler source code in the editor is included.

Figure 179. SWV Exception Trace Log – Statistics tab

Exception	Handler	% of	Number of	% of excepti...	% of debug time	Total runtime	Avg runtime	Fastest	Slowest	First	First (s)	Latest	Latest (s)
SYSTICK (EXC 15)	SysTick_Handler()	100.0000%	2172	100.0000%	0.0690%	40309	57	57	58	71567	851.988095 µs	58372327	694.908655 ms
Total for all			2172		0.0690%	40309	18						

Overflow packets: 0

The column information in the *SWV Exception Trace Log – Statistics* tab is described in [Table 8](#).

Table 8. SWV Exception Trace Log – Statistics columns details

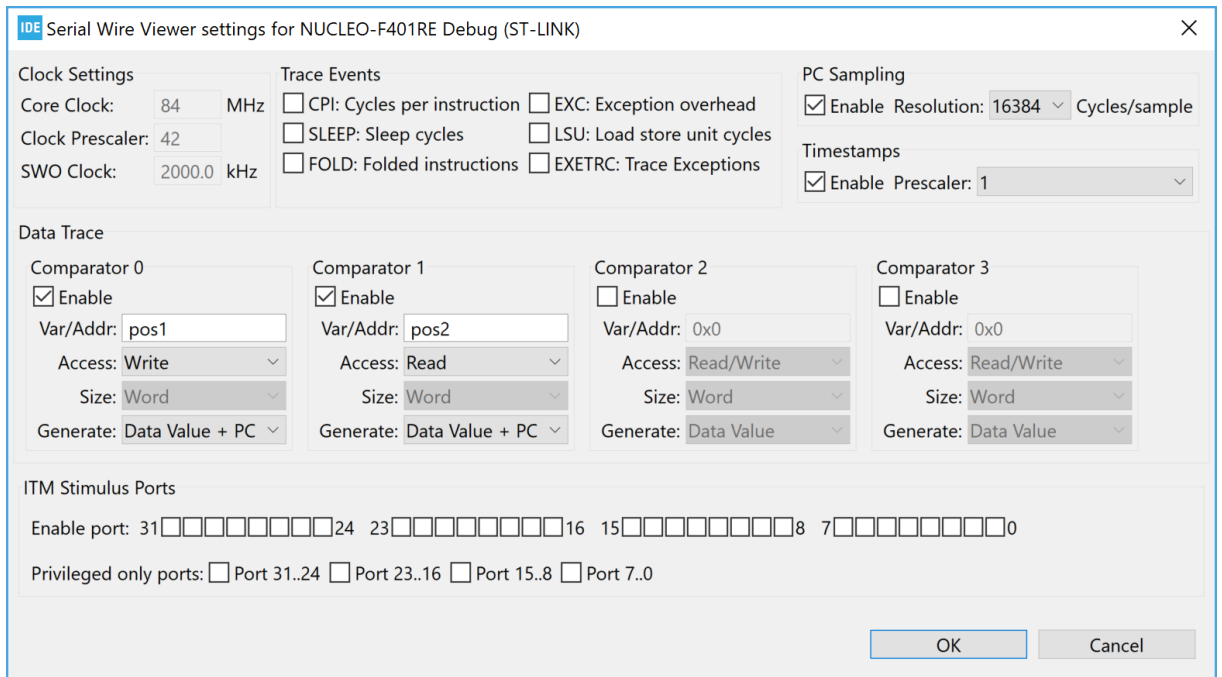
Name	Description
Exception	The name of the exception provided by the manufacturer. Also the exception or interrupt number.
Handler	The name of the interrupt handler for this interrupt. Updated when debug is paused. Is cached during the whole debug session. By double clicking the handler, the editor will open that function in the source code.
% of	This exception type's share, in percentage, of all exceptions.
Number of	The total number of entry packets received by SWV of this exception type.
% of exception time	How big part of the execution time for all exceptions that this exception type have.
% of debug time	How big part of the total execution time for this debug session that this exception type have. All the timers are restarted when the Empty SWV-Data button is pressed.
Total runtime	The total execution time in cycles for this exception type.
Avg runtime	The average execution time in cycles for this exception type.
Fastest	The execution time in cycles for the fastest exception of this exception type.
Slowest	The execution time in cycles for the slowest exception of this exception type.
First	The first encounter of an entry event for this exception type in cycles.
First(s)	The first encounter of an entry event for this exception type in seconds.
Latest	The latest encounter of an entry event for this exception type in cycles.
Latest(s)	The latest encounter of an entry event for this exception type in seconds.

4.3.3 SWV Data Trace

The *SWV Data Trace* view tracks up to four different symbols or areas in the memory. For example, global variables can be referenced by name. The data can be traced on Read, Write and Read/Write.

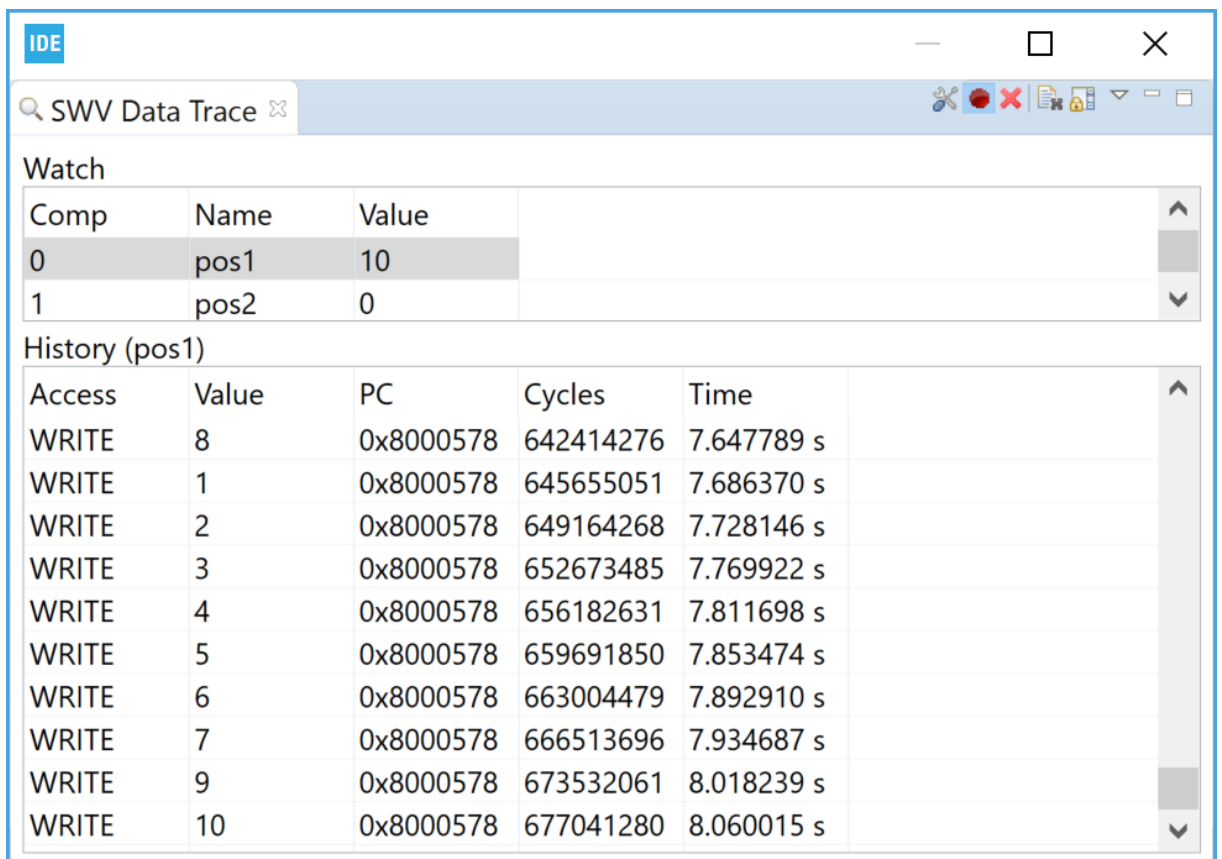
Enable **[Data Trace]** in *Serial Wire Viewer settings*. In [Figure 180](#), two global variables `pos1` and `pos2` in the program are traced on **[Write]** access.

Figure 180. SWV Data Trace configuration



When running the program in debugger with SWV trace enabled the *SWV Data Trace* view displays this information when [Comparator 0] with pos1 data is selected in the [Watch] list.

Figure 181. SWV Data Trace



The column information in the *SWV Data Trace* described in [Table 9](#).

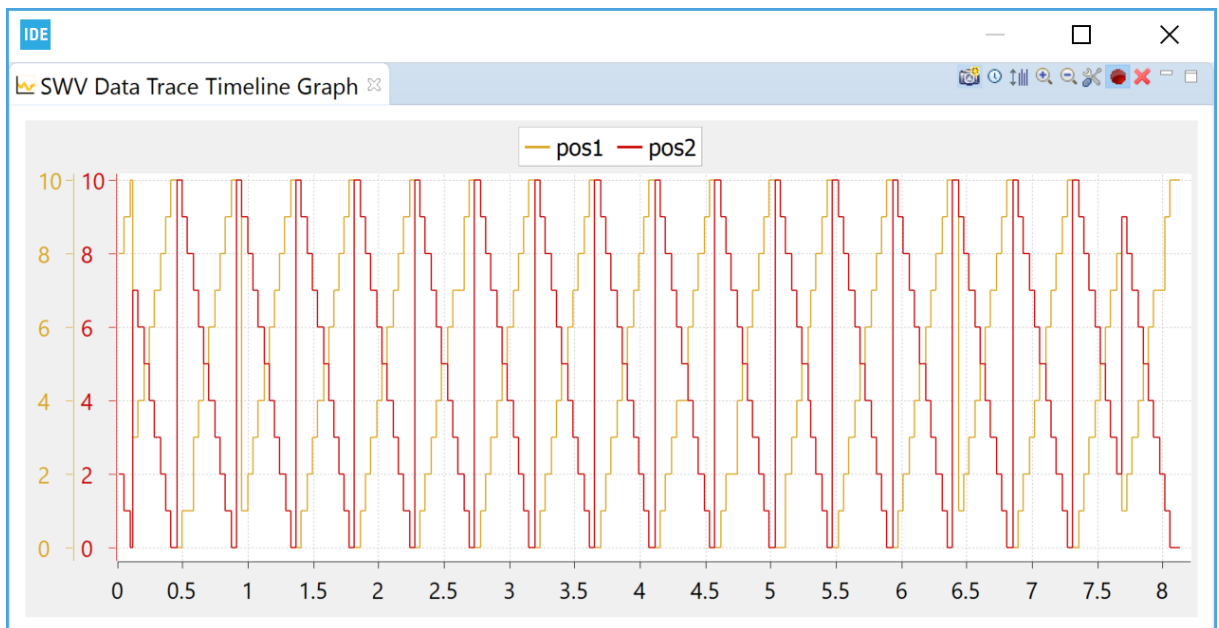
Table 9. SWV Data Trace columns details

Name	Description
Access	Read or Write access type.
Value	The value of data read or written.
PC	The PC location where read or write access occurs.
Cycles	The timestamp for the packet in cycles.
Time(s)	The timestamp for the packet in seconds.

4.3.4 SWV Data Trace Timeline Graph

The *SWV Data Trace Timeline Graph* view contains a graphical display that shows the distribution of variable values over time. It applies to the variables or memory areas in the *SWV Data Trace*. The following is displayed when using the timeline graph displaying global variables `pos1` and `pos2` counting up and down.

Figure 182. SWV Data Trace Timeline Graph



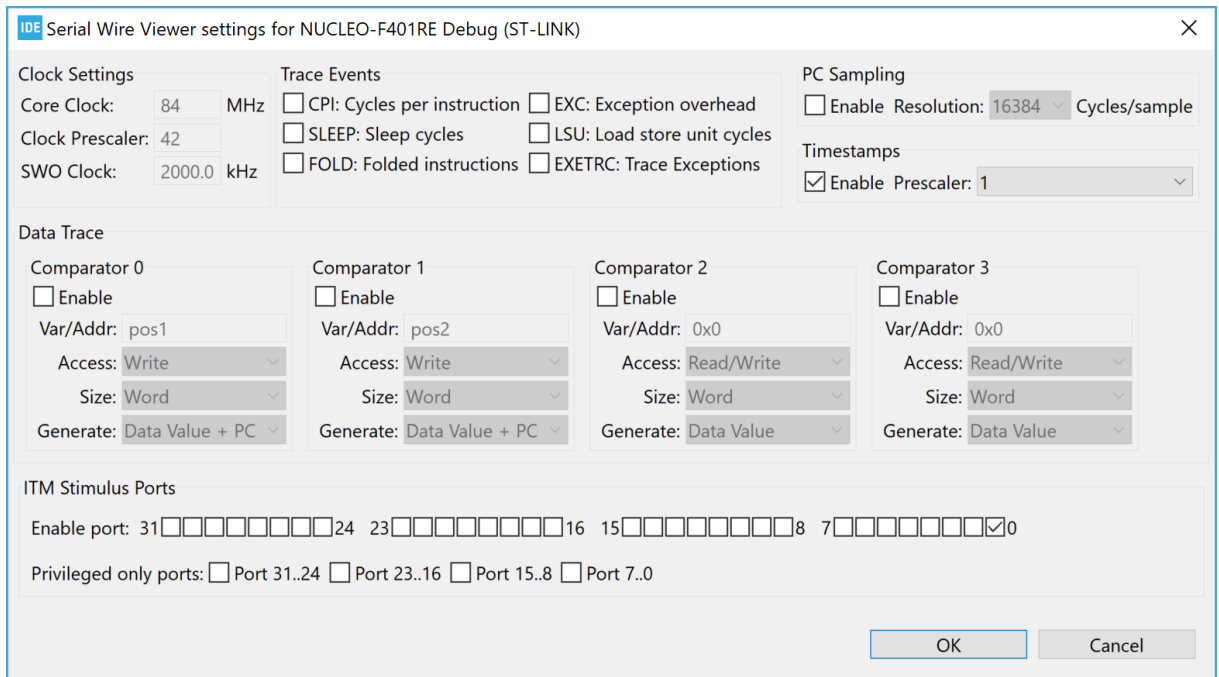
The *SWV Data Trace Timeline Graph* has the following features:

- The graph can be saved as a JPEG image file by clicking on the camera toolbar button.
- The graph shows the time in seconds by default but can be changed to cycles by clicking on the clock toolbar button.
- Y-axis can be adjusted to best fit by clicking on the y-axis toolbar button.
- Zoom in and out by clicking on the **[+]** and **[-]** toolbar buttons.
- The zoom range is limited while debug is running. Zoom details are available when debug is paused.

4.3.5 SWV ITM Data Console and printf redirection

The *SWV ITM Data Console* prints readable text output from the target application. Typically, this is done via `printf()` with output redirected to ITM channel 0. Other ITM channels can get their own console views.

To use the *SWV ITM Data Console* view, first enable one or more of the 32 ITM ports in the *Serial Wire Viewer settings* dialog.

Figure 183. SWV settings


The packets from the ITM ports are displayed in the *SWV ITM Data Console* view. The CMSIS function `ITM_SendChar()` can be used by the application to send characters to the port 0, and the `printf()` function can be redirected to use the `ITM_SendChar()` function.

The following describes how to setup `printf` redirection over ITM:

1. Configure first file `syscalls.c`. Usually, the `syscalls.c` file is located in the same source folder as `main.c`.

If no `syscalls.c` file is available in the project, it can be copied from another STM32CubeIDE project. One way to get the file is to create a new STM32 empty project for the device. In the `Src` folder, this project contains a `syscall.c` file. Copy this file to a source folder in the project where it is needed.

2. Inside the `syscalls.c` file, replace the `_write()` function with code calling `ITM_SendChar()` instead of `__io_putchar()`

```
int _write(int file, char *ptr, int len)
{
    int DataIdx;

    for (DataIdx = 0; DataIdx < len; DataIdx++)
    {
        //__io_putchar(*ptr++);
        ITM_SendChar(*ptr++);
    }
    return len;
}
```

3. Locate the `core_cmX.h` file, which contains the function `ITM_SendChar()`. The `core_cmX.h` file is included by the Device Peripheral Access Layer header file (for instance `stm32f4xx.h`, which in turn must be included in the `syscalls.c` file).

```
#include "stm32f4xx.h"
```

Use the *Include Browser* view to find the Device Peripheral Access Layer header file. Drop the core file in the *Include Browser* view, and check which files are including the `core_cmX.h` file.

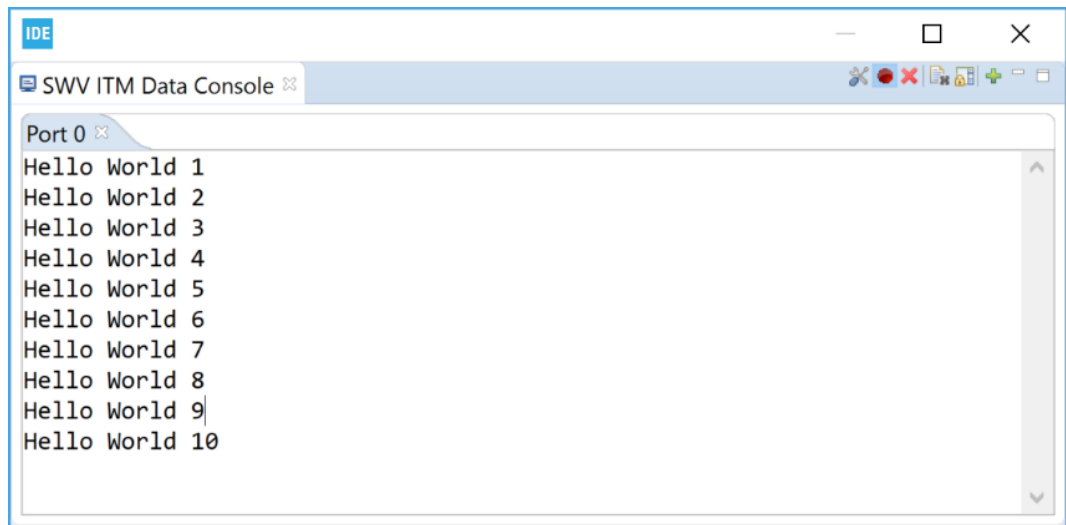
4. Test by adding include `stdio.h` and call to `printf()` into the application. Make sure that `printf()` is not called too often.

```
#include <stdio.h>

printf("Hello World %d\n", pos1);
```

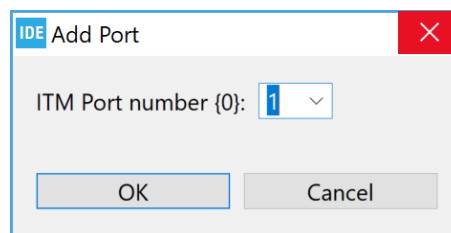
5. Start a debug session and enable **[ITM port 0]** in the *SWV ITM Data Console* view.
6. Open the *SWV ITM Data Console* view and start tracing using the red **[Start/Stop Trace]** button on the toolbar in this view.
7. Start the program. Print commands are logged to the *Port 0* tab in the view.

Figure 184. SWV ITM Data Console



8. It is possible to open new port x tabs (x from 1 to 31) by pressing the green **[+]** button on the toolbar. This opens the *Add Port* dialog. In the dialog select the **[ITM Port number]** to be opened to display it as a tab in the *SWV ITM Data Console* view.

Figure 185. SWV ITM port configuration



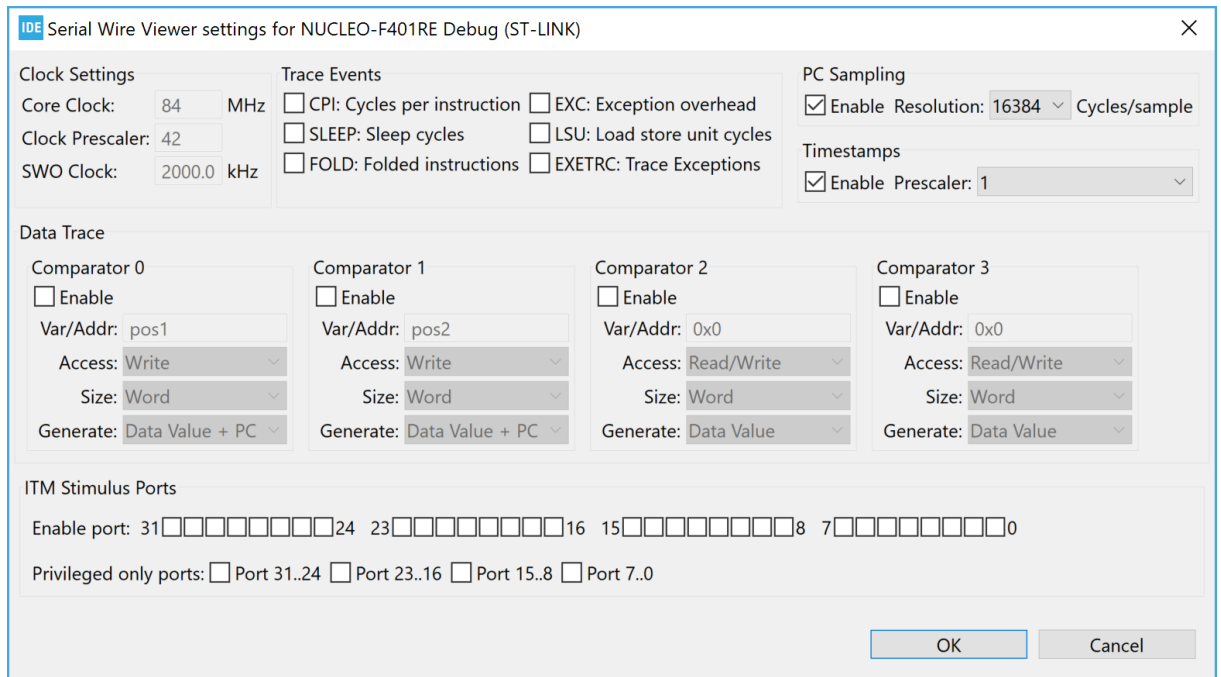
Note: Study the `ITM_SendChar()` function to learn how to write a function that transmits characters to another ITM port channel.

4.3.6 SWV Statistical Profiling

The *SWV Statistical Profiling* view displays statistics based on Program Counter (PC) sampling. It shows the amount of execution time spent within various functions. This is useful when optimizing code. The data can be copied and pasted into other applications. The view is updated when debugging is suspended.

1. Configure SWV to send Program Counter samples, as shown in Figure 186. Enable **[PC Sampling]** and **[Timestamps]**.
With the given **[Core Clock]** cycle intervals, SWV reports the Program Counter values to STM32CubeIDE. Set the **[PC Sampling]** to a high **[Cycle/sample]** value to avoid interface overflow.

Figure 186. SWV PC sampling enable



2. Open the *SWV Statistical Profiling* view by selecting **[Window]>[Show View]>[SWV Statistical Profiling]**. The view is empty since no data is collected yet.
3. Press the red **[Start/Stop Trace]** button to send the configuration to the board.
4. Resume program debugging. STM32CubeIDE starts collecting statistics about function usage via SWV when the code is executing in the target system.
5. Suspend (Pause) the debugging. The view displays the collected data. The longer the debugging session, the more statistics are collected.

Figure 187. SWV Statistical Profiling

Function	% in use	Samples	Start addr...	Size
main()	59.06%	48575	0x80005bd	0x100
readSpeed()	24.82%	20413	0x80004d5	0x46
readTemp()	16.04%	13191	0x800051b	0x2e
HAL_IncTick()	0.07%	56	0x8000b1d	0x34
SysTick_Handler()	0.01%	8	0x80009e5	0xc
writeSpeed()	0.00%	1	0x8000549	0x46

Overflow packets: 0 PC Samples: 82244

Note: A double-click on a function line in the SWV Statistical Profiling view opens the file containing the function in the editor.

The column information in the SWV Statistical Profiling is described in [Table 10](#).

Table 10. SWV Statistical Profiling columns details

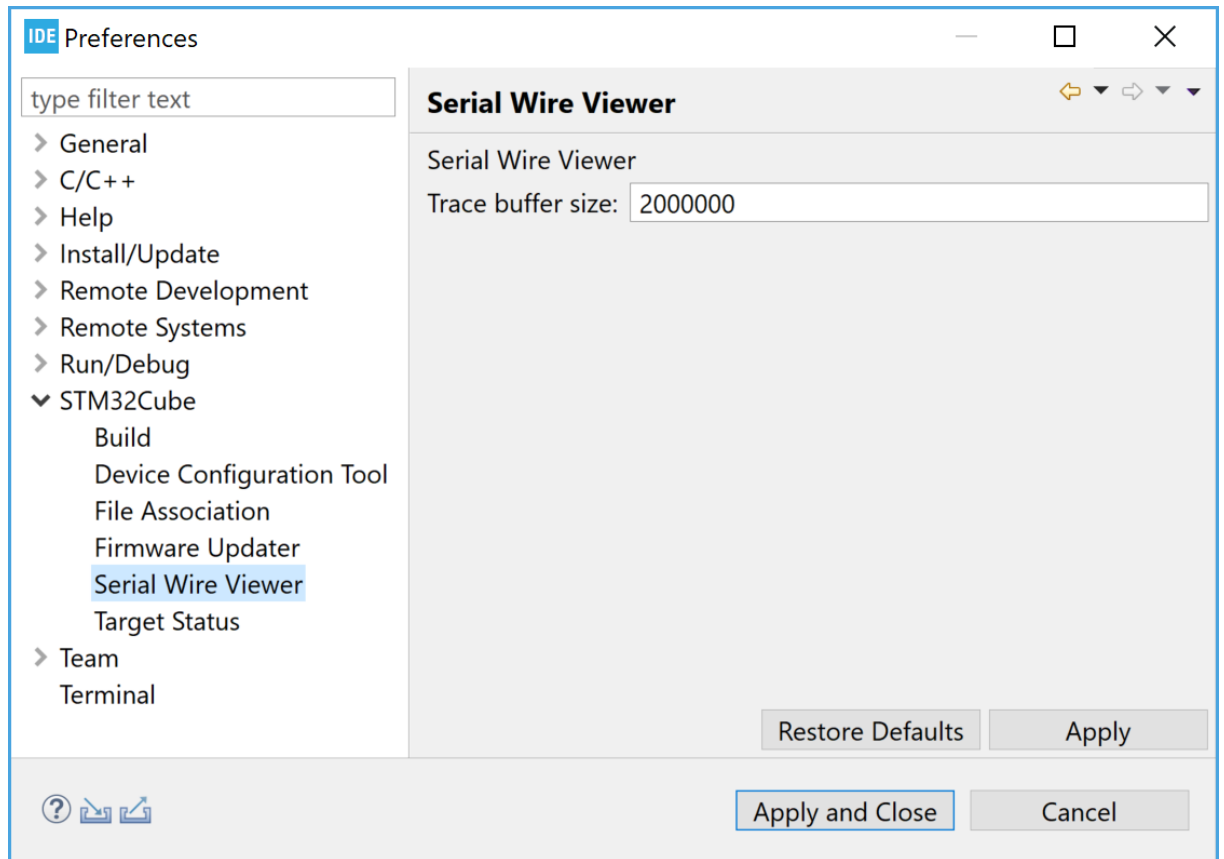
Name	Description
Function	The name of the function which is calculated by comparing address information in SWV packets with the program <code>elf</code> file symbol information.
% in use	The calculated percentage of time the function is used.
Samples	The number of samples received from the function.
Start address	The start address for the function.
Size	The size of the function.

4.4 Change the SWV trace buffer size

The incoming SWV packets are saved in the Serial Wire Viewer trace buffer, which has a default maximum size of 2 000 000 packets. To trace more packets, this figure must be increased.

Select the **[Windows]>[Preferences]** menu. In the *Preferences* dialog, select **[STM32Cube]>[Serial Wire Viewer]**. Update **[Trace buffer size]** if needed.

Figure 188. SWV Preferences



The buffer is stored in the heap. The allocated heap is displayed by first selecting the **[Windows]>[Preferences]** menu. In the *Preferences* dialog, select **[General]**. Enable **[Show heap status]** to display the current heap and allocated memory in the bottom right corner of STM32CubeIDE. There is an upper limit to the amount of memory STM32CubeIDE can allocate. This limit can be increased to store more information during a debug session.

To update the memory limit, proceed as follows:

1. Navigate to the STM32CubeIDE installation directory. Open the folder in which the IDE is stored.
2. Edit the `stm32cubeide.ini` file and change the `-Xmx1024m` parameter to the desired size in megabytes.
3. Save the file and restart STM32CubeIDE.

4.5 Common SWV problems

The following issues can occur when attempting to debug with SWV tracing:

- SWV is not enabled in the debug configuration currently used.
- The SWV Trace is not started, the red **Start/Stop Trace** button on the toolbar in some SWV view needs to be pressed to enable SWV and send SWV configuration to the target board. Then start the program to receive SWV data. For some SWV views the program then needs to be stopped again to visualize received SWV information.
- The SWO receives an excess of data. Reduce the amount of data enabled for tracing.
- The JTAG probe, the GDB server, the target board, or possibly some other part, does not support SWV.

- The target [**Core Clock**] is incorrectly set. It is very important to select the right [**Core Clock**].
If the frequency of the target [**Core Clock**] is unknown, it can sometimes be found by setting a breakpoint in a program loop and open the *Expressions* view, when the breakpoint is hit.
Click on [**Add new expression**], type `SystemCoreClock` and press [**Enter**]. This is a global variable that, according to the CMSIS standard, must be set by the software to the correct speed of the [**Core Clock**].
In CMSIS standard libraries, a function called `SystemCoreClockUpdate()` can be included in `main()` to set the `SystemCoreClock` variable. Use the *Variable* view to track it.

Note: *If the software dynamically changes the CPU clock speed during runtime, this might cause SWV to stop as the clocking suddenly becomes wrong during execution.*

To make sure that all data is received, apply the following steps:

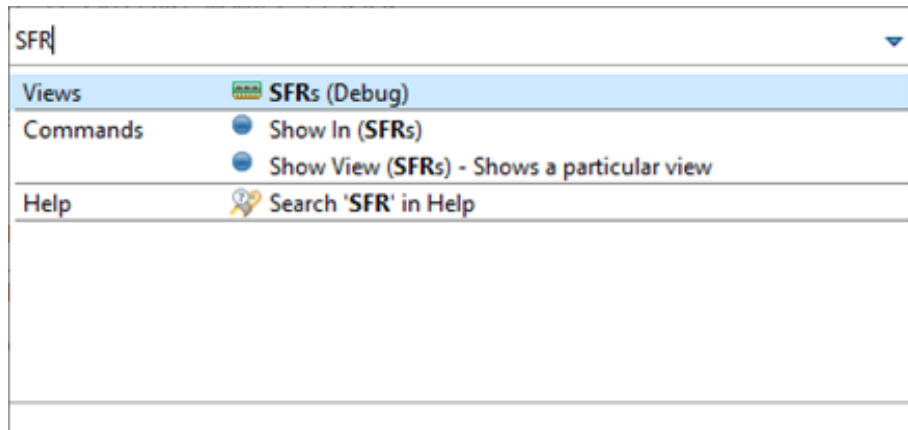
1. Open the SWV configuration. Disable all tracing except [**PC Sampling**] and [**Timestamps**]. Set the [**Resolution**] to the highest possible value.
2. Save, and open the *SWV Trace Log* view.
3. Start tracing.
4. Make sure that incoming packets can all be seen in the *SWV Trace Log* view.

5 Special Function Registers (SFRs)

5.1 Introduction to SFRs

Special Function Registers (SFRs) can be viewed, accessed and edited via the *SFRs* view. The view displays the information for the current project. Its content changes if another project is selected. To open the view from the menu, select the **[Window]>[Show View]>[SFRs]** menu command or use the **[Quick Access]** field, search for “SFR”, and select it from the views.

Figure 189. Open the *SFRs* view using the **[Quick Access]** field



5.2 Using the *SFRs* view

The *SFRs* view contains information about peripherals, registers and bit fields for the STM32 device used in the project. When debugging the project, the registers and bit fields are populated with the values read from the target. The view contains two main nodes, the Cortex[®]-M node and the STM32 node. The Cortex[®]-M node includes common Cortex[®]-M core information and the STM32 node includes the STM32 device specific peripherals.

The toolbar buttons are located at the top-right corner of the *SFRs* view.

Figure 191. *SFRs* view toolbar buttons



The **[RD]** button in the toolbar is used to force a read of the selected register. It causes a read of the register even if the register, or some of the bit fields in the register, contains a `ReadAction` attribute set in the *SVD* file.

When the register is read by pressing the **[RD]** button, all the other registers visible in the view are read again also to reflect all register updates.

The program must be stopped to read registers.

The base format buttons (**[X16]**, **[X10]**, **[X2]**) are used to change the registers display base.

The **[Configure SVD settings]** button opens the *CMSIS-SVD Settings Properties Panel* for the current project.

The **[Pin]** button ("don't follow" selection) can be used to keep focus on the current displayed *SVD* file even if the *Project Explorer* view is switched to another project.

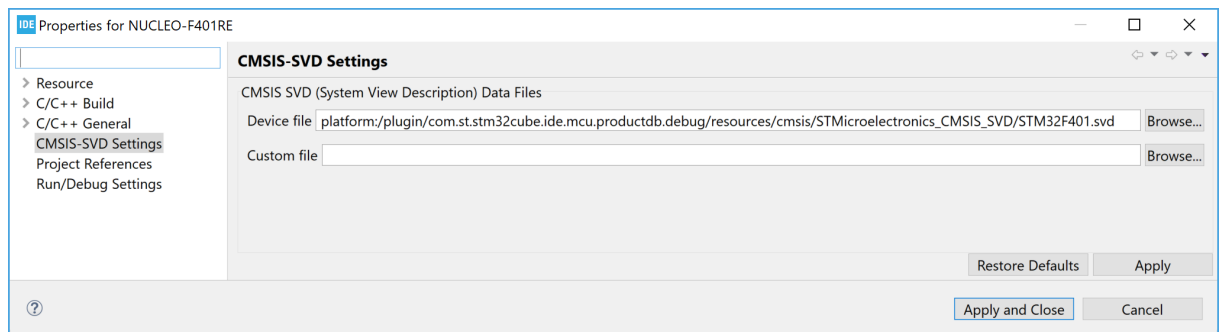
5.3 Updating CMSIS-SVD settings

The *SFRs* view for a project can display two CMSIS-SVD (System View Description) files for this project:

- The default file selected by STM32CubeIDE is the *SVD* file for the selected device in the project
- The other file can be a custom *SVD* file made to visualize specific user hardware configuration

To update the settings, use the **[Configure SVD settings]** toolbar button in the *SFRs* view to open the *CMSIS-SVD Settings* properties.

Figure 192. *SFRs* CMSIS-SVD Settings



All *SVD* files must comply with the syntax outlined in the CMSIS-SVD specification available on Arm® website. If these requirements are not met, the *SFRs* view is likely not to show any register information.

The **[Device file]** field is used for the System View Description (*SVD*) file. This file must describe the whole device. Other views may fetch information from the *SVD* file pointed out by this field, therefore it is recommended to use this field only for *SVD* files containing full STM32 device description. Updated *SVD* files can be obtained from STMicroelectronics (refer to the *HW Model, CAD Libraries and SVD* columns in the device description section on the STMicroelectronics website at www.st.com).

The **[Custom file]** field can be used to define special function registers related to custom hardware, in order to simplify the viewing of different register states. Another possible use case is to create an *SFR* favourites' file, containing a subset of the content in the **[Device file]**. This subset may be for instance composed of frequently checked registers. If a **[Custom file]** is pointed out, a new top-node in the *SFRs* view is created, which contains the **[Custom file]** related register information.

Both fields may be changed by the user and both fields may be used at the same time.

Note:

- *It is possible to write new values in the value columns of registers and bit fields when these have write access permission.*
- *It is possible to use the SFRs view while the target is running when using the ST-LINK GDB server. However the **[Live expression]** option in the debug configuration must be enabled in this case.*
- *It is not possible to use SFRs view while the target is running when using OpenOCD or SEGGER J-Link.*
- *The SFRs view can also be useful in the C/C++ Editing perspective, however then only the names and addresses of the registers are displayed.*

6 RTOS-aware debugging

Real-time operating systems (RTOS) add different kinds of objects to the design such as threads, semaphores, and timers. STM32CubeIDE includes dedicated set of views to handle Microsoft® Azure® RTOS ThreadX and FreeRTOS™ kernel objects.

These views visualize the status of the RTOS objects when stepping through the code or when the program hits a breakpoint during a debug session.

Note: FreeRTOS is a trademark of Amazon in the United States and/or other countries. All other trademarks are the property of their respective owners.

6.1 Azure® RTOS ThreadX

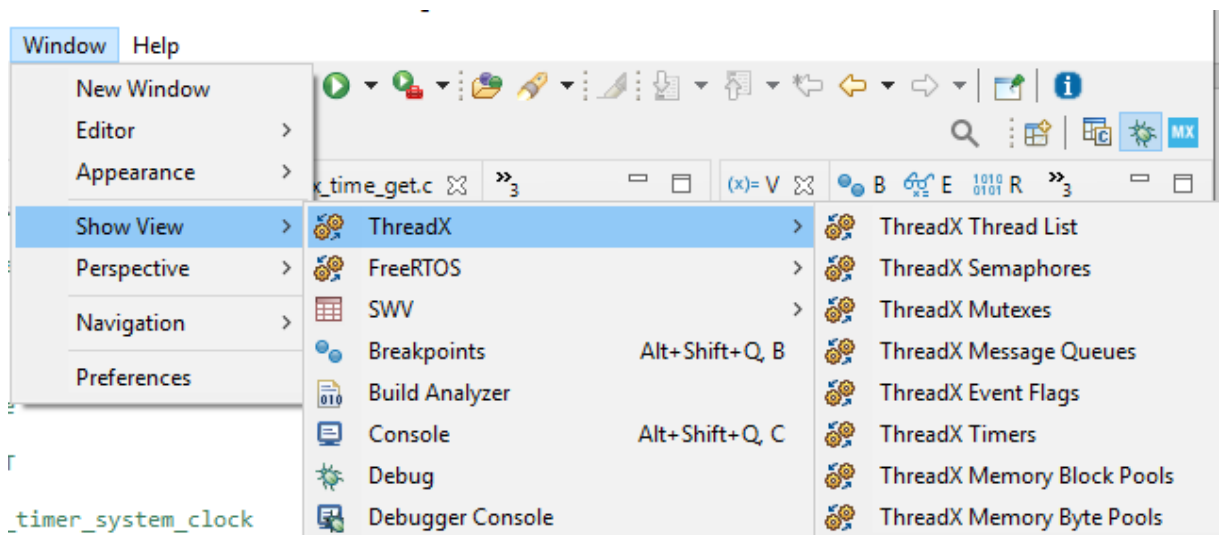
The following views are available for ThreadX:

- *ThreadX Thread List*
- *ThreadX Semaphores*
- *ThreadX Mutexes*
- *ThreadX Message Queues*
- *ThreadX Event Flags*
- *ThreadX Timers*
- *ThreadX Memory Block Pools*
- *ThreadX Memory Byte Pools*

6.1.1 Finding the views

In the *Debugger* perspective, the ThreadX-related views are opened from the menu. Select the menu command **[Window]>[Show View]>[ThreadX]>[...]** or use **[Quick Access]** and search for “ThreadX” and select it from the views.

Figure 193. ThreadX views selectable from the menu



6.1.2 ThreadX Thread List view

The *ThreadX Thread List* view displays detailed information regarding all available threads in the target system. The thread list is updated automatically each time the target execution is suspended.

There is one column for each type of thread parameter, and one row for each thread. If the value of any parameter for a thread has changed since the last time the debugger was suspended, the corresponding row is highlighted in yellow.

Figure 194. ThreadX Thread List view (default)

Name	Priority	State	Run Count	Stack Start	Stack End	Stack Size	Stack Ptr	Stack Usage
Main Thread	5	SUSPENDED (MySemaphore_1)	1	0x24001634	0x24001833	512	0x240016cc	Disabled
System Timer Thread	0	SUSPENDED	0	0x24000a18	0x24000e17	1024	0x24000dcc	Disabled
Thread One	10	SUSPENDED (MyMutex_1)	1	0x2400183c	0x24001a3b	512	0x240018d4	Disabled
Thread Two	10	RUNNING	1	0x24001a44	0x24001c43	512	0x24001bfc	Disabled
Idle								

Due to performance reasons, the *Stack Usage* column is disabled by default. To enable the stack analysis, use the [Toggle Stack Checking] toolbar button (circled in pink in Figure 195) in the *ThreadX Thread List* view toolbar.

Figure 195. ThreadX Thread List view (Stack Usage enabled)

Name	Priority	State	Run Count	Stack Start	Stack End	Stack Size	Stack Ptr	Stack Usage
Main Thread	5	SUSPENDED (Event Flag)	2	0x240015a4	0x240017a3	512	0x24001614	512
System Timer Thread	0	SUSPENDED	505	0x24000984	0x24000d83	1024	0x24000c4c	1024
Thread One	10	READY	381	0x240017ac	0x240019ab	512	0x24001824	512
Thread Two	8	RUNNING	126	0x240019b4	0x24001bb3	512	0x24001b0c	512
Idle								

The column information in the *ThreadX Thread List* view is described in Table 11.

Table 11. ThreadX Thread List details

Name	Description
N/A	A green arrow symbol indicates the currently running thread.
Name	The name assigned to the thread.
Priority	The thread priority.
State	The current state of the thread.
Run Count	The threads run counter.
Stack Start	The start address of the stack area.
Stack End	The end address of the stack area.
Stack Size	The size of the stack area (bytes).
Stack Ptr	The address of the stack pointer.
Stack Usage	The maximum thread stack (bytes). By default, ThreadX fills every byte of thread stacks with a 0xEF data pattern during thread creation. See the note below for more information.

Note: *If the Stack Usage column contains the same values as the Stack Size column for all threads, the reason could be that the thread stack has not been filled with the 0xEF data pattern during task creation. This happens if the ThreadX kernel is built with stack data pattern filling disabled. Normally, a <tx_user.h> file is used, which contains a TX_DISABLE_STACK_FILLING define. Comment this define as shown in the example below and rebuild the project to solve the problem. It is good to know that the <tx_user.h> file also contains a TX_ENABLE_STACK_CHECKING define, which can be enabled to get run-time stack checking if stack corruption is detected. Additional information can be found in the ThreadX user guide.*

Example of tx_user.h file from ThreadX header file with commented TX_ENABLE_STACK_CHECKING define:

```

/* Determine if stack filling is enabled. By default, ThreadX stack filling is enabled,
   which places an 0xEF pattern in each byte of each thread's stack. This is used by
   debuggers with ThreadX-aw:areness and by the ThreadX run-time stack checking feature. */

/* #define TX_DISABLE_STACK_FILLING */

/* Determine whether or not stack checking is enabled. By default, ThreadX stack checking is
   disabled. When the following is defined, ThreadX thread stack checking is enabled. If
   stack
   checking is enabled (TX_ENABLE_STACK_CHECKING is defined), the TX_DISABLE_STACK_FILLING
   define is negated, thereby forcing the stack fill which is necessary for the stack
   checking
   logic. */

/*#define TX_ENABLE_STACK_CHECKING*/

```

6.1.3 ThreadX Semaphores view

The *ThreadX Semaphores* view displays detailed information regarding all available resource semaphores in the target system. The view is updated automatically each time the target execution is suspended.

There is one column for each type of semaphore parameter, and one row for each semaphore. If the value of any parameter for a particular semaphore has changed since the last time the debugger was suspended, the corresponding row is highlighted in yellow.

Figure 196. ThreadX Semaphores view

Name	Count	Suspended
MySemaphore_1	0	Main Thread
MySemaphore_2	5	
MySemaphore_3	8	

Table 12. ThreadX Semaphores details

Name	Description
Name	The name assigned to the semaphore.
Count	The current semaphore count.
Suspended	The threads currently suspended because of the semaphore state.

6.1.4 ThreadX Mutexes view

The *ThreadX Mutexes* view displays detailed information regarding all available mutexes in the target system. The view is updated automatically each time the target execution is suspended.

There is one column for each type of mutex parameter, and one row for each mutex. If the value of any parameter for a particular mutex has changed since the last time the debugger was suspended, the corresponding row is highlighted in yellow.

Figure 197. ThreadX Mutexes view

Name	Owner	Owner Count	Suspended
MyMutex_1	Main Thread	1	Thread One, Thread Two
MyMutex_2		0	
MyMutex_3		0	

Table 13. ThreadX Mutexes details

Name	Description
Name	The name assigned to the mutex.
Owner	The thread that currently owns the mutex.
Owner Count	The mutex owner count (number of get operations performed by the owner thread).
Suspended	The threads currently suspended because of the mutex state.

6.1.5 ThreadX Message Queues view

The *ThreadX Message Queues* view displays detailed information regarding all available message queues in the target system. The view is updated automatically each time the target execution is suspended.

There is one column for each type of message queue parameter, and one row for each message queue. If the value of any parameter for a particular message queue has changed since the last time the debugger was suspended, the corresponding row is highlighted in yellow.

Figure 198. ThreadX Message Queues view

Name	Address	Capacity	Used	Free	Message size	Suspended
Message Queue One	0x240002b8	10	0	10	1	
Message Queue Two	0x2400003c	10	0	10	1	

Table 14. ThreadX Message Queues details

Name	Description
Name	The name assigned to the message queue.
Address	The address of the message queue.
Capacity	The maximum number of entries allowed in the queue.
Used	The current number of used entries in the queue.
Free	The current number of free entries in the queue.
Message size	The size (in 32-bit words) of each message entry.
Suspended	The threads currently suspended because of the message queue state.

6.1.6 ThreadX Event Flags view

The *ThreadX Event Flags* view displays detailed information regarding all available event flag groups in the target system. The view is updated automatically each time the target execution is suspended.

There is one column for each type of parameter, and one row for each event flag group. If the value of any parameter for a particular event flag group has changed since the last time the debugger was suspended, the corresponding row is highlighted in yellow.

Figure 199. ThreadX Event Flags view

Name	Flags	Suspended
Event Flag1	0	Main Thread
Event Flag2	0	

Table 15. ThreadX Event Flags details

Name	Description
Name	The name assigned to the event flag group.
Flags	The current value of the event flag group.
Suspended	The threads currently suspended because of the event flag group.

6.1.7 ThreadX Timers view

The *ThreadX Timers* view displays detailed information regarding all available software timers in the target system. The timers view is updated automatically each time the target execution is suspended.

There is one column for each type of timer parameter, and one row for each timer. If the value of any parameter for a particular timer has changed since the last time the debugger was suspended, the corresponding row is highlighted in yellow.

Figure 200. ThreadX Timers view

Name	Remaining	Re-init	Function
MyTimer_1	68	100	0x80005d1 <MyTimerFunction1>
MyTimer_2	72	200	0x80005f3 <MyTimerFunction2>
MyTimer_3	276	500	0x8000615 <MyTimerFunction3>

Table 16. ThreadX Timers details

Name	Description
Name	The name assigned to the timer.
Remaining	The remaining number of ticks before the timer expires.
Re-init	The timer re-initialization value (ticks) after expiration. It contains value 0 for one-shot timers.
Function	The address and name of the function that is called when the timer expires.

6.1.8 ThreadX Memory Block Pools view

The *ThreadX Memory Block Pools* view displays detailed information regarding all available memory block pools in the target system. The view is updated automatically each time the target execution is suspended.

There is one column for each type of parameter, and one row for each memory block pool. If the value of any parameter for a particular memory block pool has changed since the last time the debugger was suspended, the corresponding row is highlighted in yellow.

Figure 201. ThreadX Memory Block Pools view

Name	Address	Used	Free	Total	Block size	Pool size	Suspended
MyBlockPool_1	0x240005ec <P...	0	3	3	28	100	
MyBlockPool_2	0x240004ac <P...	0	4	4	40	200	
MyBlockPool_3	0x2400034c <P...	0	5	5	52	300	

Table 17. ThreadX Memory Block Pools details

Name	Description
Name	The name assigned to the memory block pool.
Address	The starting address of the memory block pool.
Used	The current number of allocated blocks.
Free	The current number of free blocks.
Total	The total number of memory block pools available.
Block size	The size (bytes) of each block.
Pool size	The total pool size (bytes).
Suspended	The threads currently suspended because of the memory block pool state.

6.1.9 ThreadX Memory Byte Pools view

The *ThreadX Memory Byte Pools* view displays detailed information regarding all available memory byte pools in the target system. The view is updated automatically each time the target execution is suspended.

There is one column for each type of parameter, and one row for each memory byte pool. If the value of any parameter for a particular memory byte pool has changed since the last time the debugger was suspended, the corresponding row is highlighted in yellow.

Figure 202. ThreadX Memory Byte Pools view

Name	Address	Used	Free	Size	Fragments	Suspended
Byte Pool	0x24000f84 "\214\021"	1664	6528	8192	7	

Table 18. ThreadX Memory Byte Pools details

Name	Description
Name	The name assigned to the memory byte pool.
Address	The starting address of the memory byte pool.
Used	The current number of allocated bytes.
Free	The current number of free bytes.
Size	The number of fragments.
Fragments	The size (bytes) of each block.
Suspended	The threads currently suspended because of the memory byte pool state.

6.2 FreeRTOS™

The following views are available for FreeRTOS™:

- *FreeRTOS Task List*
- *FreeRTOS Timers*
- *FreeRTOS Semaphores*
- *FreeRTOS Queues*

6.2.1 Requirements

To be able to populate the FreeRTOS™-related views with detailed information about the RTOS status, some files in the FreeRTOS™ kernel must be configured. The following sections describes some required configurations. Consult the *FreeRTOS reference manual* for detailed information.

6.2.1.1 Enable trace information

The define `configUSE_TRACE_FACILITY` in `freeRTOSConfig.h` must be enabled (set to 1). It results in additional structure members and functions to be included in the build and enables for instance stack checking in the *FreeRTOS Task List* view and lists the semaphore types in the *FreeRTOS Semaphores* view.

Example:

```
freeRTOSConfig.h
#define configUSE_TRACE_FACILITY 1
```

6.2.1.2 Add to registry

The application software must call the `vQueueAddToRegistry()` function to make the *FreeRTOS Queues* and *FreeRTOS Semaphores* views able to display objects. The function adds an object to the FreeRTOS™ *Queue registry* and takes two parameters, the first is the handle of the queue, and the second is a description of the queue, which is presented in FreeRTOS™-related views.

Example:

```
vQueueAddToRegistry(mailId, "osMailQueue");
vQueueAddToRegistry(osQueueHandle, "osQueue");
vQueueAddToRegistry(osSemaphoreHandle, "osSemaphore");
```

6.2.1.3 RTOS profiling information

To get valid RTOS run-time statistics, the application must set up a run-time statistics time base. The time-base clock is recommended to run at least 10 times faster than the frequency of the clock used to handle the RTOS tick interrupt. To enable the FreeRTOS™ collection of run-time statistics, file `freeRTOSConfig.h` must include:

1. Define `configGENERATE_RUN_TIME_STATS` 1
2. Define `portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()` to call the function that configures a timer to be used for profiling
3. Define `portGET_RUN_TIME_COUNTER_VALUE()` to call the function that reads the current value from the profiling timer

Example:

```
freeRTOSConfig.h
#define configGENERATE_RUN_TIME_STATS 1
#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() configureRunTime()
#define portGET_RUN_TIME_COUNTER_VALUE() getRunTimeCounter()
```

Or, if a run-time variable is available in the system:

```
freeRTOSConfig.h
#define configGENERATE_RUN_TIME_STATS 1
#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() (RunTime=0UL)
#define portGET_RUN_TIME_COUNTER_VALUE() RunTime
```

If the *Run Time* column in the *FreeRTOS Task List* view displays N/A after making these three settings, the problem can arise if project is not built with optimization level `-O0`. The reason is quite likely found in the declaration in `tasks.c` of `ulTutoralRunTime`.

Example:

```
#if ( configGENERATE_RUN_TIME_STATS == 1 )
    PRIVILEGED_DATA static uint32_t ulTaskSwitchedInTime = 0UL;
    /*< Holds the value of a timer/counter the last time a task was switched in. */
    PRIVILEGED_DATA static uint32_t ulTotalRunTime = 0UL;
    /*< Holds the total amount of execution time as defined by the run time counter clock. */
#endif
```

Solutions:

- Either declare the variable as volatile:

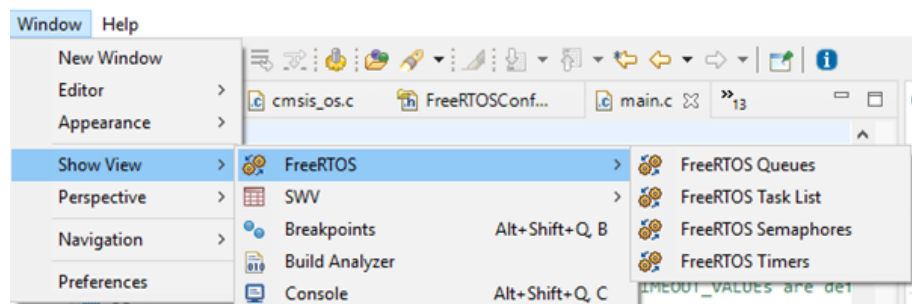
```
PRIVILEGED_DATA volatile static uint32_t ulTotalRunTime = 0UL;
/*< Holds the total amount of execution time as defined by the run time counter clock.
*/
```

- Or simply change the optimization level only for `tasks.c` by
 1. Right-clicking it in *Project Explorer* view and open *Properties*
 2. Select **[Properties]>[C/C++ Build]>[Settings]>[Tool Settings]>[Optimization]**
 3. Set **[Optimization Level]** to `None (-O0)`

6.2.2 Finding the views

In the *Debugger* perspective, the FreeRTOS™-related views are opened from the menu. Select the menu command **[Window]>[Show View]>[FreeRTOS]>[...]** or use **[Quick Access]**, search for “FreeRTOS” and select from the views.

Figure 203. FreeRTOS™-related views selectable from the menu



6.2.3 FreeRTOS Task List view

The *FreeRTOS Task List* view displays detailed information regarding all available tasks in the target system. The task list is updated automatically each time the target execution is suspended.

There is one column for each type of task parameter, and one row for each task. If the value of any parameter for a task has changed since the last time the debugger was suspended, the corresponding row is highlighted in yellow, as shown in the example in [Figure 204](#).

Figure 204. FreeRTOS Task List (default)

Name	Priority (Base/...	Start of Stack	Top of Stack	State	Event Object	Min Free Stack	Run Time (%)
IDLE	0/0	0x200001c0	0x20000558 <ucHeap=1308>	RUNNING		Disabled	99%
LEDThread	3/3	0x20000150	0x20000308 <ucHeap=716>	SUSPENDED		Disabled	0%
Timer Svc	2/2	0x20000630	0x20000960 <ucHeap=2420>	BLOCKED	TimerQ	Disabled	1%

Due to performance reasons, stack analysis (the *Min Free Stack* column) is disabled by default. To enable stack analysis (refer to Figure 206), use the *Toggle Stack Checking* toolbar button in the *FreeRTOS Task List* view toolbar as shown in Figure 205.

Figure 205. FreeRTOS™ *Toggle Stack Checking*

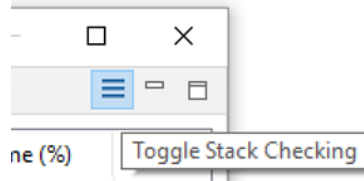


Figure 206. FreeRTOS Task List (*Min Free Stack* enabled)

Name	Priority (Base/...)	Start of Stack	Top of Stack	State	Event Object	Min Free Stack	Run Time (%)
IDLE	0/0	0x200003c0	0x20000560 <ucHeap+1316>	RUNNING		>256	99%
LEDThread	3/3	0x20000150	0x200002b8 <ucHeap+636>	DELAYED		>256	0%
Tmr Svc	2/2	0x20000630	0x20000960 <ucHeap+2420>	BLOCKED	TmrQ	>256	1%

The *FreeRTOS Task List* view in Figure 206 contains a *Min Free Stack* column. The column information is changed to *Stack Usage* if the project is built with the following define set:

```
#define configRECORD_STACK_HIGH_ADDRESS 1
```

In this case, the full stack usage is presented according to the format *Used/Total(%Used)* as shown in Figure 207.

Figure 207. FreeRTOS Task List with *ConfigRECORD_STACK_HIGH_ADDRESS* enabled

Name	Priority (B...	Start of S...	Top of St...	State	Event Object	Stack Usage	Run Time...
IDLE	0/0	0x20000...	0x20000...	RUNNING		96B / 2052B (4.7%)	N/A
THREAD1	24/24	0x20001...	0x20001...	DELAYED		144B / 512B (28.1%)	N/A
THREAD2	24/24	0x20001...	0x20001...	DELAYED		144B / 512B (28.1%)	N/A
Tmr Svc	2/2	0x20000...	0x20000...	BLOCKED	TmrQ	168B / 1028B (16.3%)	N/A

The column information in the *FreeRTOS Task List* view is described in [Table 19](#).

Table 19. FreeRTOS Task List details

Name	Description
N/A	A green arrow symbol indicates the task currently running.
Name	The name assigned to the task.
Priority (Base/Actual)	The task base priority and actual priority. The base priority is the priority assigned to the task. The actual priority is a temporary priority assigned to the task due to the priority inheritance mechanism.
Start of Stack	The address of the stack region assigned to the task.
Top of Stack	The address of the saved task stack pointer.
State	The current state of the task.
Event Object	The name of the resource that has caused the task to be blocked.
Min Free Stack ⁽¹⁾	The stack “high watermark”. Displays the minimum number of bytes left on the stack for a task. A value of 0 (most likely) indicates that a stack overflow has occurred. <i>Note: This feature must be enabled in the “View” toolbar.</i>
Run Time (%)	The run-time statistics provide information on the percentage of time the task has been used. This can be used for profiling the system during development.

1. When the application is built with `configRECORD_STACK_HIGH_ADDRESS = 1`, the column name is changed to “Stack Usage”. It displays the stack usage in detailed format as “Used/Total(%Used)”.

6.2.4 FreeRTOS Timers view

The *FreeRTOS Timers* view displays detailed information regarding all available software timers in the target system. The view is updated automatically each time the target execution is suspended. There is one column for each type of timer parameter, and one row for each timer. If the value of any parameter for a timer has changed since the last time the debugger was suspended, the corresponding row is highlighted in yellow.

Figure 208. FreeRTOS Timers

Name	Active	Period	Type	Id	Callback
myTimerTEST	True	200	Auto-Reload	0x0	0x8000429 <osTimerCallback>

The column information in the *FreeRTOS Timers* view is described in [Table 20](#).

Table 20. FreeRTOS Timers details

Name	Description
Name	The name assigned to the timer.
Active	The active status information.
Period	The time (in ticks) between timer start and the execution of the callback function.
Type	The type of timer. Auto-reload timers are automatically reactivated after expiration. One-shot timers expire only once.
Id	The timer identifier.
Callback	The address and name of the callback function executed when the timer expires.

- Note:**
1. If no name appears in the Name field, check that the timer is created with a name. The first parameter when calling `xTimerCreate()` must contain the timer name string.
 2. When using software timers, a `Tmr Svc` task and a `TmrQ` queue are created automatically. These objects are displayed in the *FreeRTOS Task List* view and *FreeRTOS Queues* view.

6.2.5 FreeRTOS Semaphores view

The *FreeRTOS Semaphores* view displays detailed information regarding all available synchronization objects in the target system, including:

- Mutexes
- Counting semaphores
- Binary semaphores
- Recursive semaphores

The view is updated automatically each time the target execution is suspended. There is one column for each type of semaphore parameter, and one row for each semaphore. If the value of any parameter for a semaphore has changed since the last time the debugger was suspended, the corresponding row is highlighted in yellow.

Figure 209. FreeRTOS Semaphores

Name	Address	Type	Size	Free	# Blocked tasks
osSemaphore	0x20000058	BINARY_SEMAPHORE	1	0	0

- Note:** If the Type information displays *N/A*, make sure that the define `configUSE_TRACE_FACILITY` is enabled in file `FreeRTOSconfig.h`.

The column information in the *FreeRTOS Semaphores* view is described in Table 21.

Table 21. FreeRTOS Semaphores details

Name	Description
Name	The name assigned to the semaphore.
Address	The address of the object.
Type	The type of the object.
Size	The maximum number of owning tasks.
Free	The number of free slots currently available.
#Blocked tasks	The number of tasks currently blocked waiting for the object.

6.2.6 FreeRTOS Queues view

The *FreeRTOS Queues* view displays detailed information regarding all available queues in the target system. The view is updated automatically each time the target execution is suspended. There is one column for each type of queue parameter, and one row for each queue. If the value of any parameter for a queue has changed since the last time the debugger was suspended, the corresponding row is highlighted in yellow.

Figure 210. FreeRTOS Queues

Name	Address	Max Length	Item Size	Current Length	# Waiting Tx	# Waiting Rx
osQueue	0x20000068	1	2	0	0	1

The column information in the *FreeRTOS Queues* view is described in Table 22.

Table 22. FreeRTOS Queues details

Name	Description
Name	The name assigned to the queue in the queue registry.
Address	The address of the queue.
Max Length	The maximum number of items that the queue can hold.
Item Size	The size in bytes of each queue item.
Current Length	The number of items currently in the queue.
#Waiting Tx	The number of tasks currently blocked waiting to be sent to the queue.
#Waiting Rx	The number of tasks currently blocked waiting to be received from the queue.

6.3 RTOS-kernel-aware debug

The RTOS-kernel-aware debug in *STM32CubeIDE* supports the Microsoft® Azure® RTOS ThreadX and FreeRTOS™ operating systems using an RTOS proxy. The RTOS proxy is included in *STM32CubeIDE* and can be used with ST-LINK GDB server, OpenOCD, and SEGGER J-Link GDB server.

When RTOS-kernel-aware debugging is enabled and a debug session is started, all threads are listed in the *Debug* view. By selecting a thread in the *Debug* view, the thread current context is visualized in views. For instance, the *Variables*, *Registers*, *Editor* views reflect the active stack frame.

Figure 211 shows a debug session. The *ThreadX Thread List* view displays that the Message Queue Receiver Thread is RUNNING. This can also be seen in the *Debug* view. In the *Debug* view the `MsgSenderThreadTwo_Entry` function is selected, and the editor area displays that the thread is waiting in a sleep for 500 ms state.

Figure 211. RTOS-kernel-aware debug

The screenshot displays the ST-Link debugger interface. The **ThreadX Thread List** window at the bottom shows the following threads:

Name	Pri...	State	Run Count	Stack Start	Stack End	Stack Size	Stack Ptr	Stack Usage
Message Queue Receiver Thread	10	RUNNING	875	0x2400139c	0x2400159b	512	0x2400141c	Disabled
Message Queue Sender Thread One	5	SLEEP (200)	74	0x24000f8c	0x2400118b	512	0x2400105c	Disabled
Message Queue Sender Thread Two	5	SLEEP (404)	30	0x24001194	0x24001393	512	0x24001264	Disabled
System Timer Thread	0	SUSPENDED	874	0x24000570	0x2400096f	1024	0x2400083c	Disabled
Idle								

The main editor window shows the source code for the `MsgSenderThreadTwo_Entry` function, which is currently in a sleep state (`tx_thread_sleep(500);`).

To enable RTOS-kernel-aware debugging the *Debugger* tab in the *Debug Configurations* dialog contains settings to enable RTOS proxy, driver (RTOS ThreadX or FreeRTOS™), port (Cortex® core) and configuration of port number to use with the proxy.

The *RTOS* tab also contains a *Driver settings* selection to select the *Driver* (“ThreadX” or “FreeRTOS”) and the port used. The “Auto-detect” driver setting is still experimental.

Figure 212. RTOS-kernel-awareness debug configuration

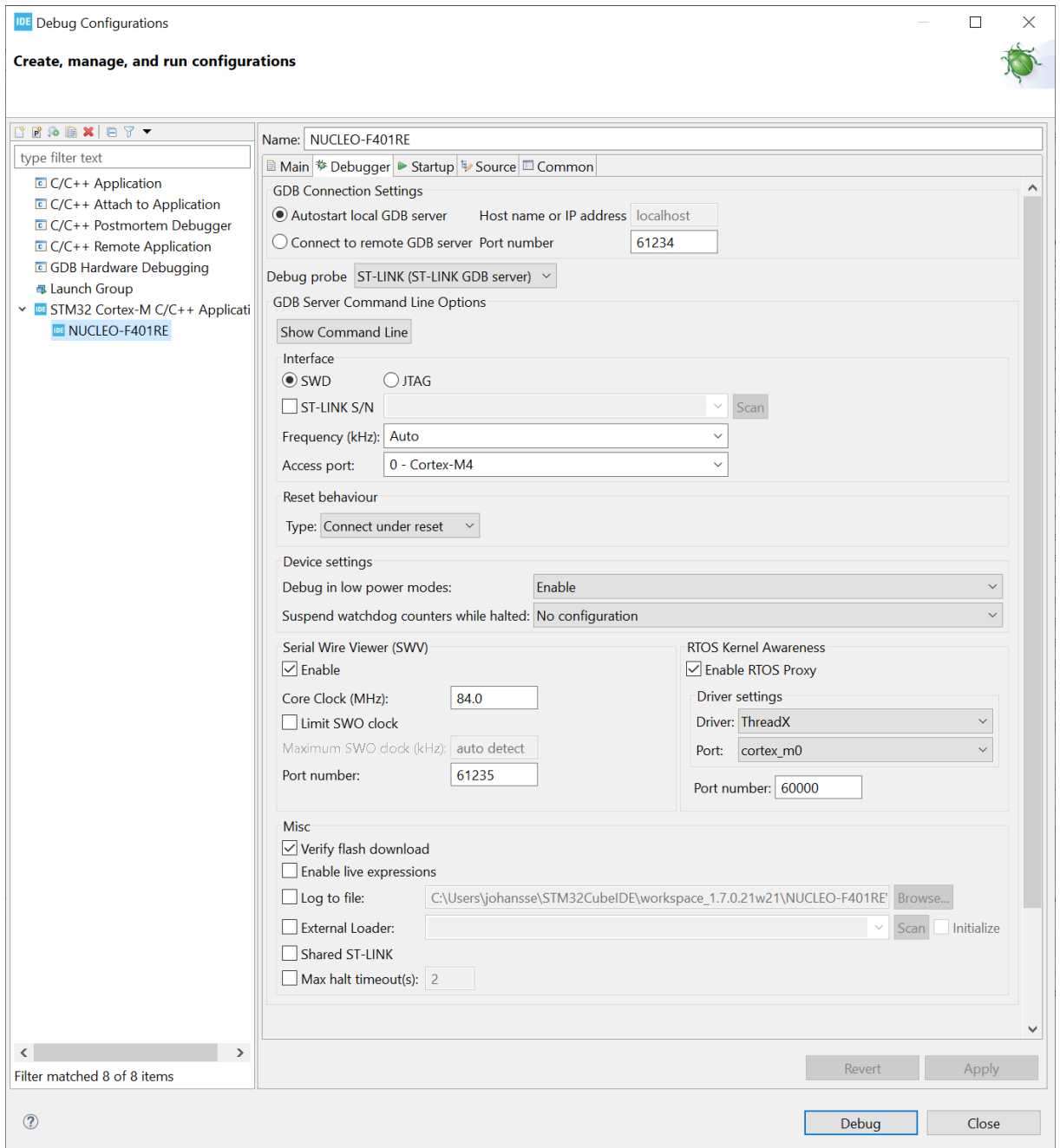
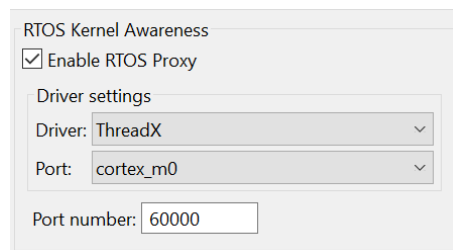


Figure 213. ThreadX-kernel-awareness debug configuration



The port selection lists the supported cores. The items listed depend on the selected RTOS driver as displayed in Figure 214 and Figure 215.

Figure 214. ThreadX port configuration

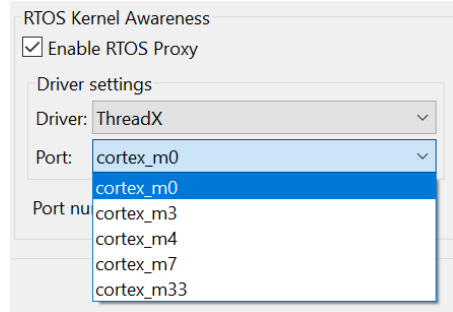
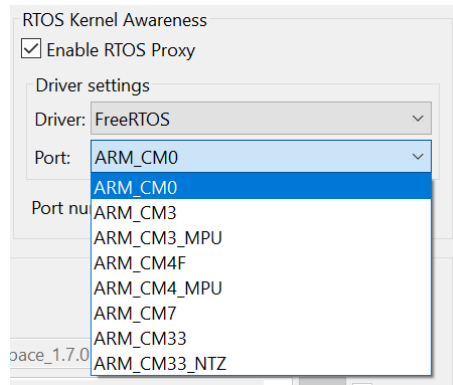


Figure 215. FreeRTOS™ port configuration



Known limitations

- Live expressions must be disabled when used with the ST-LINK GDB server
- The *Registers* view content for swapped out threads is intermixed with active CPU context for some registers (all registers are not saved by the context switcher)
- The *Registers* view floating point registers are not updated correctly

7 Fault Analyzer

7.1 Introduction to the Fault Analyzer

The *STM32CubeIDE* Fault Analyzer feature interprets information extracted from the Cortex[®]-M nested vector interrupt controller (NVIC) in order to identify the reasons that caused a fault. This information is visualized in the *Fault Analyzer* view. It helps to identify and resolve hard-to-find system faults that occur when the CPU is driven into a fault condition by the application software.

Among such conditions are:

- Accessing invalid memory locations
- Accessing memory locations on misaligned boundaries
- Executing undefined instruction
- Division by zero

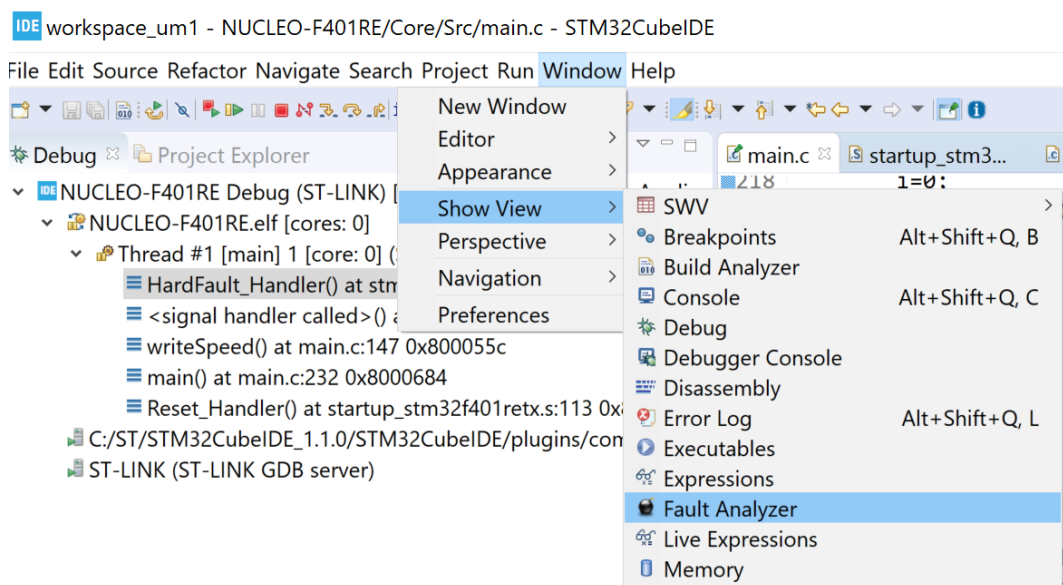
Upon fault occurrence, the code line where the fault occurred is displayed in the debugger. The view displays the reasons for the error condition. Faults are coarsely categorized into hard, bus, usage and memory faults.

- Hard and bus faults occur when an invalid access attempt is made across the bus, either of a peripheral register or a memory location
- Usage faults are the result of illegal instructions or other program errors
- Memory faults include attempts of access to an illegal location or violations of rules maintained by the memory protection unit (MPU)

To further assist fault analysis, an exception stack frame visualization option provides a snapshot of the MCU register values at the time of the crash. Isolating the fault to an individual instruction allows to reconstruct the MCU condition at the time the faulty instruction was executed.

In the *Debugger* perspective, the *Fault Analyzer* view is opened from the menu. Select the menu command **[Window]>[Show View]>[Fault Analyzer]** or use the **[Quick Access]** field, search for “*Fault Analyzer*” and select it from the views.

Figure 216. Open the *Fault Analyzer* view



7.2 Using the *Fault Analyzer* view

The *Fault Analyzer* view has five main sections, which can be expanded and collapsed. The sections contain different kinds of information for better understanding the reason that caused a particular fault to occur. The sections are:

- Hard Fault Details
- Bus Fault Details
- Usage Fault Details
- Memory Management Fault Details
- Register Content During Fault Exception

When a fault has occurred, it is possible to [**Open editor on fault location**] and [**Open disassembly on fault location**] by pressing the buttons in the view.

Figure 217 shows an example of the *Fault Analyzer* view when an error is detected. In this example, the error is caused by a project making a divide by zero with the debugger stopped in the `HardFault_Handler()`.

Opening the *Fault Analyzer* view when this happens displays the reason of the error. In the example, it displays [**Usage Fault Detected**] and [**Attempt to perform a division by zero (DIVBYZERO)**]. The *Register Content During Fault Exception* presents register values when the problem occurred.

Figure 217. Fault Analyzer view

Hard Fault Detected

Hard Fault Details

- Bus, memory management or usage fault (FORCED)
- Failed vector fetch (VECTBL)
- Debug event (DEBUGEVT)

Bus Fault Details

- Instruction access violation (IBUSERR)
- Precise data access violation (PRECISERR)
- Imprecise data access violation (IMPRECISERR)
- Unstacking error (UNSTKERR)
- Stacking error (STKERR)
- Floating point lazy state preservation error (LSPERR)

Bus fault address register (BFAR): 0xe0

Usage Fault Details

- Attempt to execute an undefined instruction (UNDEFINSTR)
- Attempt to switch to invalid state (INVSTATE)
- Attempt to do exception with bad value in EXEC_RETURN number (INVPC)
- Attempt to execute a coprocessor instruction (NOCP)
- Attempt to perform an unaligned access (UNALIGNED)
- Attempt to perform a division by zero (DIVBYZERO)

Memory Management Fault Details

- Instruction access violation (IACCVIOL)
- Data access violation (DACCVIOL)
- Unstacking error (MUNSTKERR)
- Stacking error (MSTKERR)
- Floating point lazy state preservation error (MLSPERR)

Mem manage address register (MMFAR): 0xe0

Register Content During Fault Exception

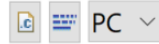
Name	Value
sp (M...	0x20017fc0
r0	0x0
r1	0x0
r2	0x0
r3	0x8000000
r12	0x0
lr	0x8000685
pc	0x800055c
xpsr	0x2100000

The value of the stack pointer when the fault occurred. Please verify that this value points to a valid stack memory region.

MSP = Main Stack Pointer
PSP = Process Stack Pointer

The *Fault Analyzer* view contains these toolbar buttons:

Figure 218. Fault Analyzer toolbar



- The first toolbar button (left) opens the *Editor* on the fault location return address by using the information in the PC and LR registers in the stack and the symbol information in the debugged `elf` file.
- The second toolbar button (middle) opens the *Disassembly* view on the fault location return address by using the information in PC and LR registers in the stack and the symbol information in the debugged `elf` file.
- The third toolbar button (right) selects if the PC or LR register is used when opening the *Editor* or *Disassembly* view on error location.

Figure 219 and Figure 220 show the *Editor* and *Disassembly* views opened using the toolbar buttons to find the fault location in the example.

Figure 219. Fault analyzer open editor on fault

```

142
143 int writeSpeed(int pos)
144 {
145
146     // update speed
147     speed= pos/tsec;
148     return speed;
149 }
150 }
    
```

Figure 220. Fault Analyzer open disassembly on fault

```

0800055c:  sdiv    r2, r1, r2
08000560:  ldr     r1, [pc, #28] ; (0x8000580 <writeSpeed+56>)
08000562:  ldr     r1, [r3, r1]
08000564:  str     r2, [r1, #0]
148      return speed;
    
```

Note: The *Fault Analyzer* can be used on all *STM32* projects. It requires no special code and no special build configuration. All data are collected for the Cortex[®]-M registers. The symbol information is read from the debugged `elf` file.

8 Build Analyzer

8.1 Introduction to the Build Analyzer

The STM32CubeIDE Build Analyzer feature interprets program information from the `elf` file in detail and presents the information in a view. If a `map` file, with similar name, is found in the same folder as the `elf` file the information from the `map` file is also used and even more information can be presented.

The *Build Analyzer* view is useful to optimize or simplify a program. The view contains two tabs, the *Memory Regions* and *Memory Details* tabs:

- The *Memory Regions* tab is populated with data if the `elf` file contains a corresponding `map` file. When the `map` file is available, this tab can be seen as a brief summary of the memory regions with information about the region name, start address and size. The size information also comprises the total size, free and used part of the region, and usage percentage.
- The *Memory Details* tab contains detailed program information based on the `elf` file. The different section names are presented with address and size information. Each section can be expanded and collapsed. When a section is expanded, functions/data in this section is listed. Each presented function/data contains address and size information.

8.2 Using the Build Analyzer

The *Build Analyzer* view is by default open in the C/C++ perspective. If the view is closed it can be opened from the menu. Select the menu command **[Window]>[Show View]>[Build Analyzer]** or use the **[Quick Access]** field, search for “*Build Analyzer*” and select it from the views.

When the *Build Analyzer* view is open, select an `elf` file in the *Project Explorer* view. The *Build Analyzer* view is then updated with the information from this file. When an `elf` file is selected and a `map` file, with similar name, is found in the same folder, additional information from the `map` file is also used by the view.

The *Build Analyzer* view is also updated if a project node in the *Project Explorer* view is selected. In this case the *Build Analyzer* uses the `elf` file that corresponds to the current active build configuration of the project.

Figure 221. Build analyzer

Region	Start address	End address	Size	Free	Used	Usage (%)
FLASH	0x08000000	0x08080000	524288	518840	5448	1.04%
RAM	0x20000000	0x20018000	98304	96604	1700	1.73%

8.2.1 Memory Regions tab

The *Memory Regions* tab in the *Build Analyzer* view displays information based on the corresponding `map` file. If no information is displayed, it means that there is no corresponding `map` file found. When a `map` file is found, the region names, start address, end address, total size of region, free size, used size and usage information are presented.

These regions are usually defined in the linker script file (`.ld`) used when building the program. Update the linker script file if a memory region location or size must be changed.

Note: *The Memory Regions tab is empty if the `elf` file has no corresponding `map` file.*

Figure 222. Memory Regions tab

Region	Start address	End address	Size	Free	Used	Usage (%)
RAM	0x20000000	0x20018000	96 KB	16.23 KB	79.77 KB	83.09%
FLASH	0x08000000	0x08040000	256 KB	236.17 KB	19.83 KB	7.75%
FLASH_ICONS	0x08040000	0x08050000	64 KB	44.47 KB	19.53 KB	30.52%
FLASH_IMAGES	0x08050000	0x08070000	128 KB	10.81 KB	117.19 KB	91.55%
FLASH_SOUND	0x08070000	0x0807f000	60 KB	7.75 KB	52.25 KB	87.08%
FLASH_D	0x0807f000	0x0807f800	2 KB	1.99 KB	8 B	0.39%
FLASH_V	0x0807f800	0x08080000	2 KB	1.99 KB	12 B	0.59%

The column information is described in the Table 23.

Table 23. Memory Regions tab information

Name	Description
Region	Name of memory region (if a corresponding map file is found).
Start address	The start address of the region, defined in the linker script.
End address	End address of the region.
Size	The total size of memory region.
Free	The free size in the memory region.
Used	The used size in the memory region.
Usage %	The percentage of used size relative to the total memory region size. See Table 24 for the bar icon color information.

The Usage (%) column contains a bar icon corresponding to the percentage value. The bar has different colors depending on the percentage of used memory.

Table 24. Memory Regions usage color

Usage color	Description
Green	Less than 75% of memory used.
Yellow	75% to 90% of memory used.
Red	More than 90% of memory used.

8.2.2 Memory Details tab

The Memory Details tab of the Build Analyzer view contains information for the elf file. Each section in the Memory Details tab can be expanded so that individual functions and data can be seen. The tab presents columns with name, run address, load address, and size information.

Figure 223. Memory Details tab

Name	Run address (VMA)	Load address (LMA)	Size
RAM	0x20000000		96 KB
> .data	0x20000000	0x08004f49	12 B
> .bss	0x2000000c		78.25 KB
> _user_heap_stack	0x2001390c		1.5 KB
FLASH	0x08000000		256 KB
> FLASH_ICONS	0x08040000		64 KB
> FLASH_IMAGES	0x08050000		128 KB
> .flash_images	0x08050000	0x08050000	117.19 KB
> image3	0x08064c08	0x08064c08	34.18 KB
> image2	0x080561a8	0x080561a8	58.59 KB
> image1	0x08050000	0x08050000	24.41 KB
> FLASH_SOUND	0x08070000		60 KB
> .flash_sound	0x08070000	0x08070000	52.25 KB
> FLASH_D	0x0807f000		2 KB
> .flash_d	0x0807f000		8 B
> FLASH_V	0x0807f800		2 KB
> .flash_v	0x0807f800	0x0807f800	12 B

The column information is described in Table 25.

Table 25. Memory Details tab information

Name	Description
Name	Name of memory region, section, function, and data. A green icon is used to mark functions while the blue icon is used for data variables.
Run Address (VMA)	The Virtual Memory Address contains the address used when the program is running.
Load Address (LMA)	The Load Memory Address is the address used for load, for instance for the initialization values of global variables.
Size	Used size (total size for <i>Memory Regions</i>).

Note: The memory region name is only displayed if a corresponding *map* file is found.

8.2.2.1 Size information

The size information in the *Memory Details* tab is calculated from the symbol size in the *elf* file. If a corresponding *map* file is investigated, it may contain a different size value. The size is usually correct for C files but the value presented for assembler files depends on how the size information is written in the assembler files. The constants used by the function must be defined within the section definition. At the end of the section, the size directive is used by the linker to calculate the size of the function.

Example: Reset_Handler in startup.s file

This example shows how to write the `Reset_Handler` in an assembler startup file to include the constants `_sidata`, `_sdata`, `_edata`, `_sbss`, and `_ebss` in the `Reset_Handler` size information in the `elf` file. If these constants are defined out of the `Reset_Handler` section definition, their sizes are not included in the calculated size of the `Reset_Handler`. To include them in the size of the `Reset_Handler`, these definitions must be placed inside the `Reset_Handler` section as presented in the code example below.

```
.section .text.Reset_Handler
.weak Reset_Handler
.type Reset_Handler, %function

Reset_Handler:
ldr sp, =_estack /* set stack pointer */

/* Copy the data segment initializers from flash to SRAM */
movs r1, #0
b LoopCopyDataInit

CopyDataInit:
ldr r3, =_sidata

/* initialization code data, bss, ... */
...

/* Call the application's entry point */
bl main
bx lr

/* start address for the initialization values defined in linker script */
.word _sidata
.word _sdata
.word _edata
.word _sbss
.word _ebss

.size Reset_Handler, .-Reset_Handler
```

8.2.2.2 Sorting

The sort order of a *Memory Details* tab column can be changed by clicking on the column name.

Figure 224. *Memory Details* sorted by size

Name	Run address (VMA)	Load address (LMA)	Size
FLASH	0x08000000		256 KB
FLASH_IMAGES	0x08050000		128 KB
flash_images	0x08050000	0x08050000	117.19 KB
image2	0x080561a8	0x080561a8	58.59 KB
image3	0x08064c08	0x08064c08	34.18 KB
image1	0x08050000	0x08050000	24.41 KB
RAM	0x20000000		96 KB
.bss	0x2000000c		78.25 KB
_user_heap_stack	0x2001390c		1.5 KB
.data	0x20000000	0x08004f49	12 B
FLASH_ICONS	0x08040000		64 KB
FLASH_SOUND	0x08070000		60 KB
FLASH_D	0x0807f000		2 KB
.flash_d	0x0807f000		8 B
FLASH_V	0x0807f800		2 KB
.flash_v	0x0807f800	0x0807f800	12 B

8.2.2.3 Search and filter

The information in the *Memory Detailstab* can be filtered by entering a string in the search field.

Figure 225 shows a search example for names including the string “sound”.

Figure 225. *Memory Details* search and filter

Name	Run address (VMA)	Load address (LMA)	Size
FLASH_SOUND	0x08070000		60 KB
.flash_sound	0x08070000	0x08070000	52.25 KB
sound1	0x08070000	0x08070000	19.53 KB
sound2	0x08074e20	0x08074e20	19.53 KB
sound4	0x0807afc8	0x0807afc8	8.3 KB
sound3	0x08079c40	0x08079c40	4.88 KB

8.2.2.4 Calculate the sum of sizes

The sum of the sizes of several lines in the *Memory Details* tab can be calculated by selecting these lines in the view. The sum of the selection is presented above the *Name* column in the view.

Figure 226. Sum of sizes

Build Analyzer
NUCLEO-F401RE.elf -/NUCLEO-F401RE/Debug - Oct 25, 2019 11:15:52 AM

Memory Regions Memory Details

Selection: 92.77 KB

Search

Name	Run address (VMA)	Load address (LMA)	Size
> FLASH	0x08000000		256 KB
▼ FLASH_IMAGES	0x08050000		128 KB
▼ .flash_images	0x08050000	0x08050000	117.19 KB
▪ image2	0x080561a8	0x080561a8	58.59 KB
▪ image3	0x08064c08	0x08064c08	34.18 KB
▪ image1	0x08050000	0x08050000	24.41 KB
> RAM	0x20000000		96 KB
> FLASH_ICONS	0x08040000		64 KB
▼ FLASH_SOUND	0x08070000		60 KB
> .flash_sound	0x08070000	0x08070000	52.25 KB
> FLASH_D	0x0807f000		2 KB
> FLASH_V	0x0807f800		2 KB

8.2.2.5 Display the size information in byte format

The *Build Analyzer* view can display size information in different format according to the [Show Byte], [Show Hex] or [Show Human] selection. The icon in the *Build Analyzer* toolbar is used to switch between these formats. Prefer [Show Byte] or [Show Hex] when copying and pasting of data into an Excel® document for later calculations.

Figure 227. Show byte count

Build Analyzer
NUCLEO-F401RE.elf -/NUCLEO-F401RE/Debug - Oct 25, 2019 11:15:52 AM

Memory Regions Memory Details

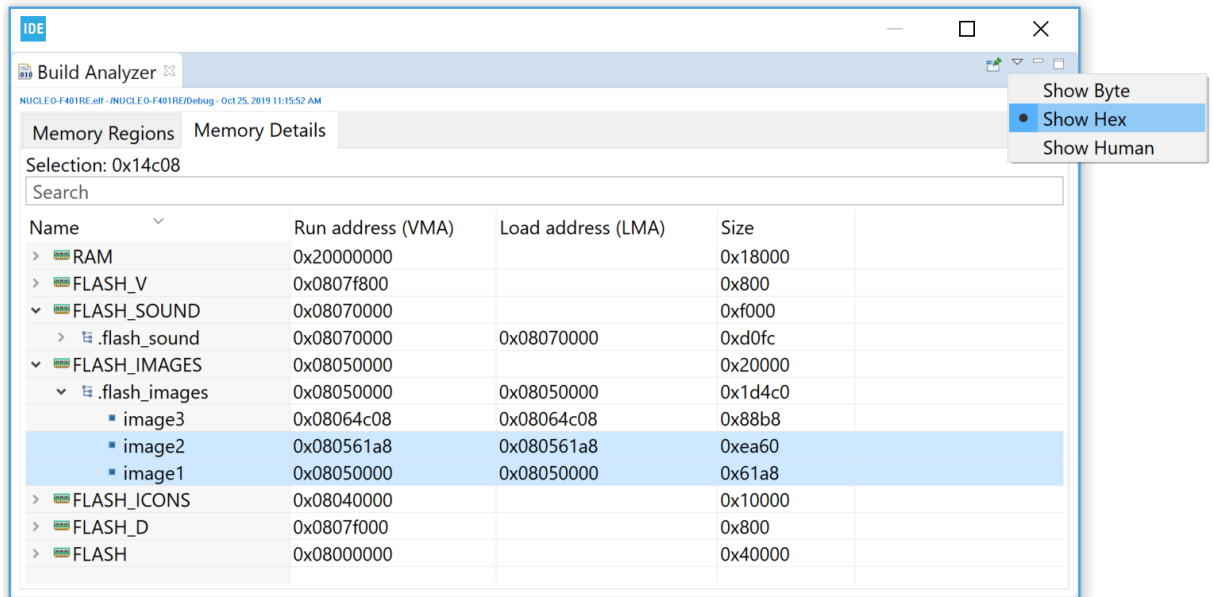
Selection: 85000

Search

Name	Run address (VMA)	Load address (LMA)	Size
> RAM	0x20000000		98304
> FLASH_V	0x0807f800		2048
▼ FLASH_SOUND	0x08070000		61440
> .flash_sound	0x08070000	0x08070000	53500
▼ FLASH_IMAGES	0x08050000		131072
▼ .flash_images	0x08050000	0x08050000	120000
▪ image3	0x08064c08	0x08064c08	35000
▪ image2	0x080561a8	0x080561a8	60000
▪ image1	0x08050000	0x08050000	25000
> FLASH_ICONS	0x08040000		65536
> FLASH_D	0x0807f000		2048
> FLASH	0x08000000		262144

- Show Byte
- Show Hex
- Show Human

Figure 228. Show hex count

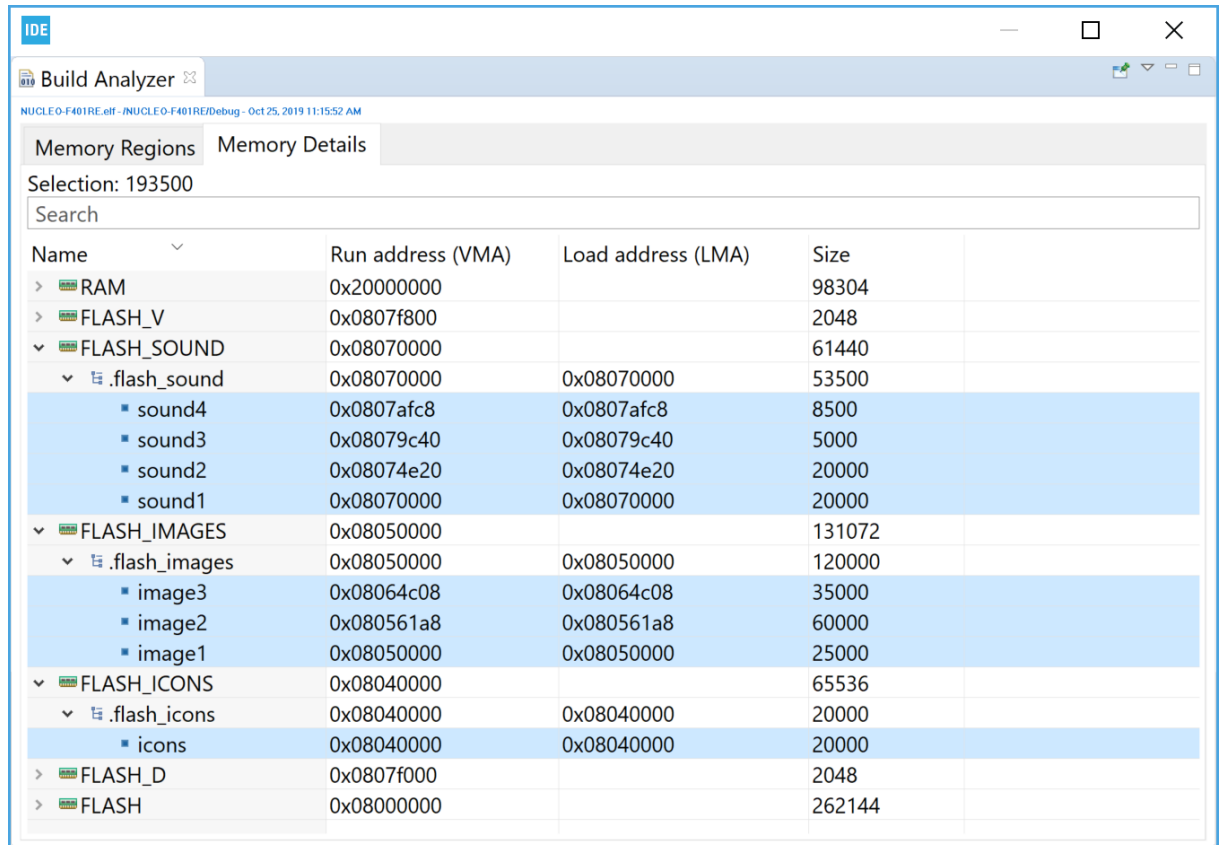


8.2.2.6

Copy and paste

The data in the *Memory Details* tab can be copied to other applications in CSV format by selecting the rows to copy and typing **Ctrl+C**. The copied data can be pasted into another application with the **Ctrl+V** command.

Figure 229. Copy and paste



The **Ctrl+C** copy of the lines selected in [Figure 229](#) provides the **Ctrl+V** results below:

```
"sound4";"0x0807afc8";"0x0807afc8";"8500"  
"sound3";"0x08079c40";"0x08079c40";"5000"  
"sound2";"0x08074e20";"0x08074e20";"20000"  
"sound1";"0x08070000";"0x08070000";"20000"  
"image3";"0x08064c08";"0x08064c08";"35000"  
"image2";"0x080561a8";"0x080561a8";"60000"  
"image1";"0x08050000";"0x08050000";"25000"  
"icons";"0x08040000";"0x08040000";"20000"
```

9 Static Stack Analyzer

9.1 Introduction to the Static Stack Analyzer

The *STM32CubeIDE Static Stack Analyzer* calculates the stack usage based on the built program. It analyzes the `.su` files, generated by `gcc`, and the `elf` file in detail, and presents the resulting information in the view.

The view contains two tabs, the *List* and *Call Graph* tabs.

The *List* tab is populated with the stack usage for each function included in the program. The tab lists one line per function, each line consisting of the *Function*, *Local cost*, *Type*, *Location* and *Info* columns.

Figure 230. Static Stack Analyzer List tab

Function	Local cost	Type	Location	Info
main	88	STATIC	main.c:79	
TIM_TI1_SetConfig	16	STATIC	stm32f4xx_hal_tim.c:4540	
TIM_SlaveTimer_SetConfig	12	STATIC	stm32f4xx_hal_tim.c:4983	
TIM_CCxChannelCmd	8	STATIC	stm32f4xx_hal_tim.c:4739	
TIM_Base_SetConfig	0	STATIC	stm32f4xx_hal_tim.c:4481	
SystemInit	0	STATIC	system_stm32f4xx.c:148	
HAL_TIM_TriggerCallback	0	STATIC	stm32f4xx_hal_tim.c:4364	
HAL_TIM_SlaveConfigSync...	16	STATIC	stm32f4xx_hal_tim.c:4143	
HAL_TIM_ReadCapturedVal...	0	STATIC	stm32f4xx_hal_tim.c:4217	
HAL_TIM_PeriodElapsedCal...	0	STATIC	stm32f4xx_hal_tim.c:4304	
HAL_TIM_PWM_PulseFinish...	0	STATIC	stm32f4xx_hal_tim.c:4349	
HAL_TIM_OC_DelayElapsed...	0	STATIC	stm32f4xx_hal_tim.c:4319	
HAL_TIM_IRQHandler	8	STATIC	stm32f4xx_hal_tim.c:2809	
HAL_TIM_IC_Start_IT	8	STATIC	stm32f4xx_hal_tim.c:1672	

The *Call Graph* tab contains an expandable list with functions included in the program. Lines representing functions calling other functions can be expanded to see the call hierarchy.

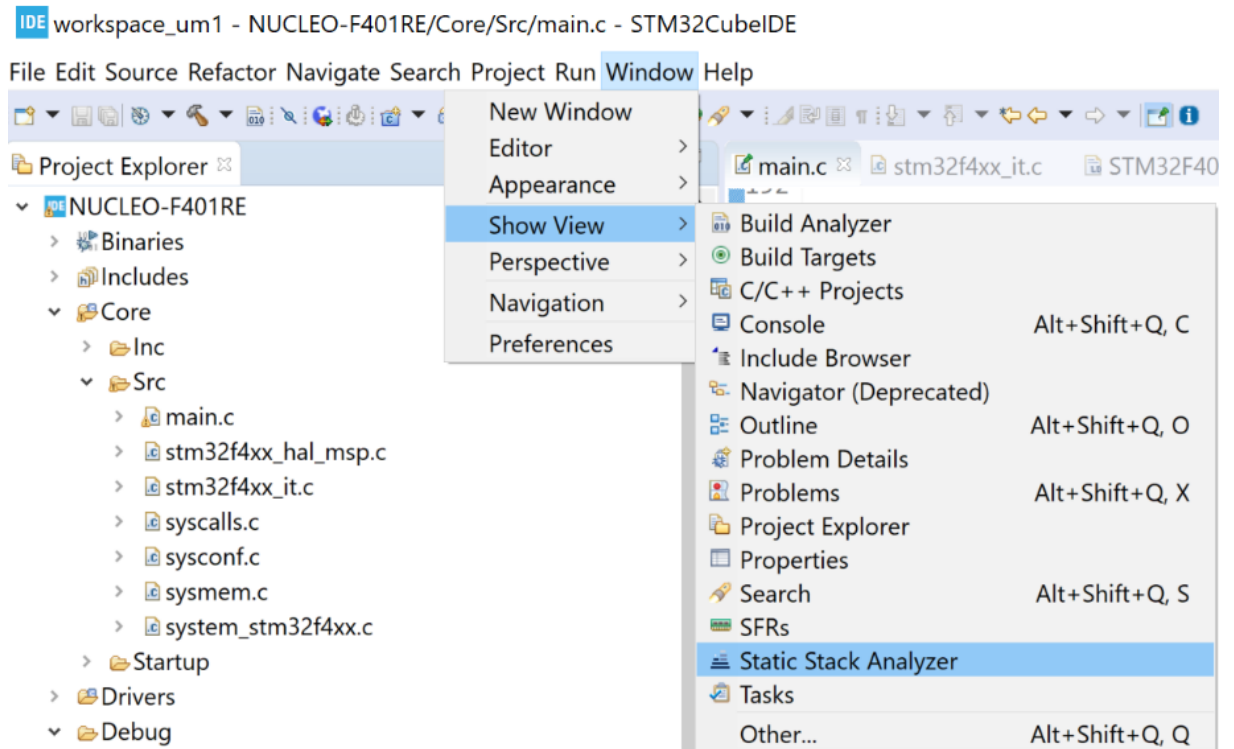
Figure 231. Static Stack Analyzer Call Graph tab

Function	Depth	Max cost	Local cost	Type	Location	Info
main						
UsageFault_Handler	?	?	0	STATIC	stm32f4xx_it.c:116	Max cost uncertain. Recursive
ADC_IRQHandler	?	?	0	STATIC	stm32f4xx_it.c:103	Max cost uncertain. Recursive. No stack usage information available for this ...
BusFault_Handler	?	?	0	STATIC	stm32f4xx_it.c:77	Max cost uncertain. Recursive
HardFault_Handler	?	?	0	STATIC	stm32f4xx_it.c:90	Max cost uncertain. Recursive
MemManage_Handler	?	?	0	STATIC	stm32f4xx_it.c:90	Max cost uncertain. Recursive
Reset_Handler	7	184	0			Max cost uncertain. No stack usage information available for this function
TIM4_IRQHandler	3	8	0	STATIC	stm32f4xx_it.c:173	Max cost uncertain
NMI_Handler	0	0	0	STATIC	stm32f4xx_it.c:68	
PendSV_Handler	0	0	0	STATIC	stm32f4xx_it.c:147	
frame_dummy	0	0	0			Max cost uncertain. No stack usage information available for this function
SysTick_Handler	1	0	0	STATIC	stm32f4xx_it.c:156	
SVC_Handler	0	0	0	STATIC	stm32f4xx_it.c:129	
DebugMon_Handler	0	0	0	STATIC	stm32f4xx_it.c:138	
_do_global_dtors_aux	0	0	0			Max cost uncertain. No stack usage information available for this function
_fini	0	0	0			Max cost uncertain. No stack usage information available for this function

9.2 Using the Static Stack Analyzer

The *Static Stack Analyzer* view is by default open in the *C/C++* perspective. If the view is closed, it can be opened from the menu. Select the menu command **[Window]>[Show View]>[Static Stack Analyzer]**. Another way to open the *Static Stack Analyzer* view is to type “*Static Stack Analyzer*” in the **[Quick Access search bar]** and select it from the views.

Figure 232. Open the Static Stack Analyzer view



The *Static Stack Analyzer* view is populated when a built project is selected in the *Project Explorer*. The project must be built with option **[Generate per function stack usage information]** enabled, otherwise the view cannot present any stack information.

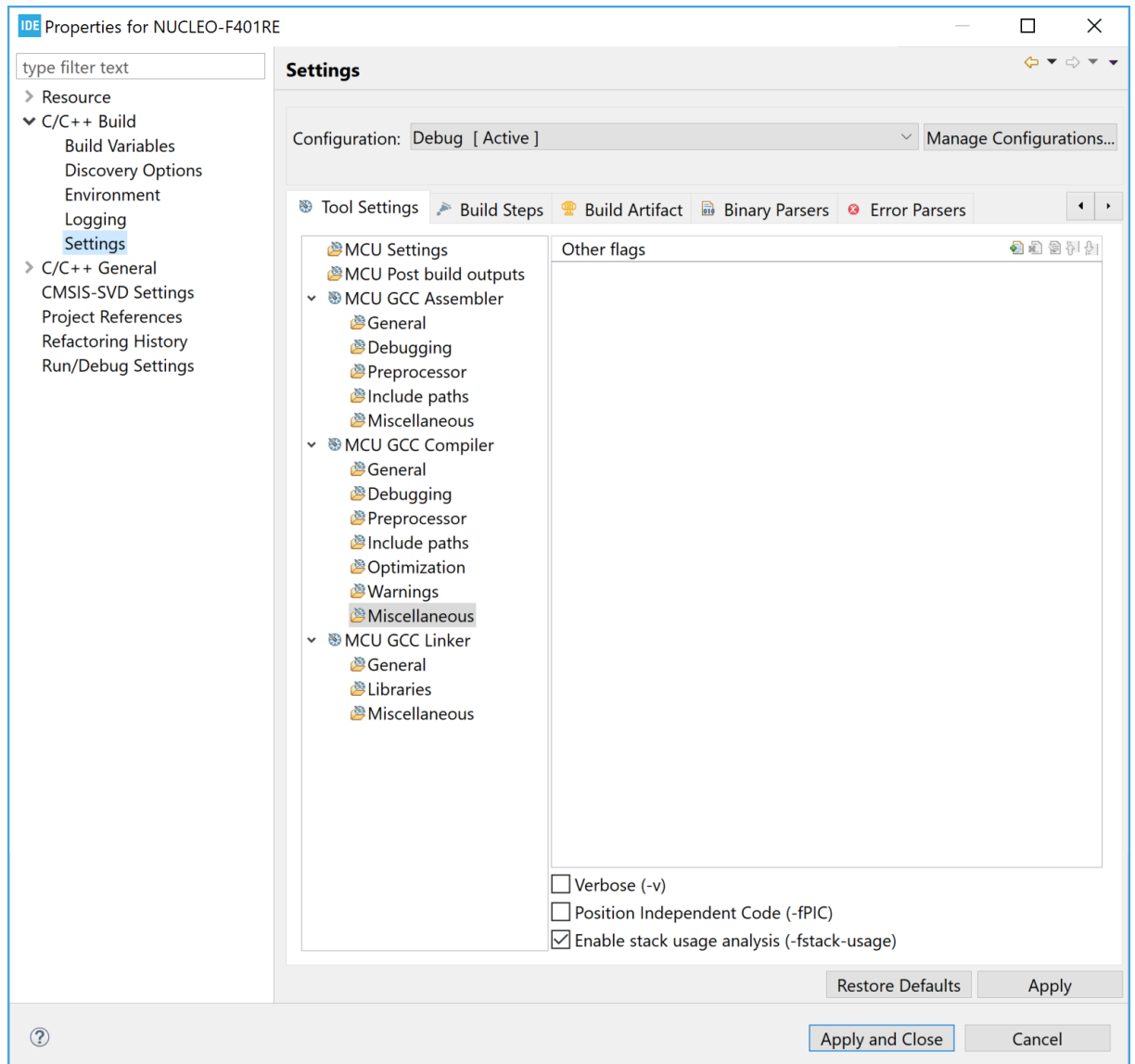
How to setup the compiler to generate stack usage information is explained in the next section.

9.2.1 Enable stack usage information

If the top of the view displays the message `No stack usage information found, please enable in the compiler settings`, the build configuration must be updated for the compiler to generate stack information:

1. Open the project properties, for instance with a right-click on the project in the *Project Explorer* view
2. Select *Properties* and, in the dialog, select **[C/C++ Build]>[Settings]**
3. Select the *Tool Settings* tab
4. Select **[MCU GCC Compiler]>[Miscellaneous]**
5. Select **[Enable stack usage information (-fstack-usage)]** as shown in [Figure 233](#)
6. Save the setting and rebuild the program

Figure 233. Enable generate per function stack usage information



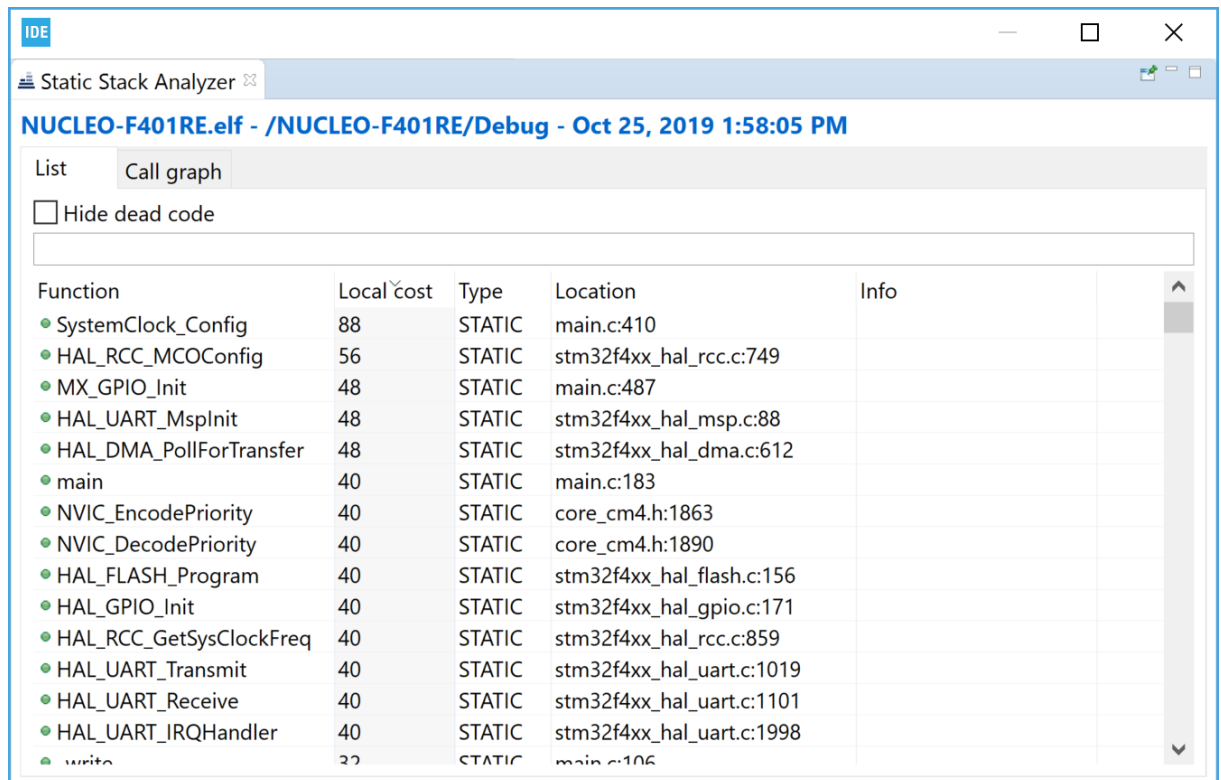
9.2.2 List tab

The *List* tab contains a list of all functions included in the selected program with options to [Hide dead code] functions and [Filter] visible functions.

Use the [Hide dead code] selection to enable or disable the listing of dead code functions.

If used, the [Filter] field restricts the display to functions matching the characters it contains.

Figure 234. Static Stack Analyzer List tab



The column information in the *List* tab is described in Table 26.

Table 26. Static Stack Analyzer List tab details

Name	Description
Function	Function name.
Local cost	The number displays how many bytes of stack the function uses.
Type	Tells if the function uses a STATIC or DYNAMIC stack allocation. When DYNAMIC allocation is used the actual stack size is run-time dependent and the the <i>Local cost</i> value is uncertain due to the dynamic size of stack.
Location	Indicates where the function is declared. It is possible to double-click on a line and open the file with the defined function in the editor.
Info	Additional information about the calculation.

The *List* tab sort order can be changed by clicking on a column name.

Note: By double-clicking on a line that displays the file location and line number in the *List* tab, the function is opened in the Editor view.

9.2.3 Call Graph tab

The *Call Graph* tab contains detailed program information in a tree view. Each function included in the program but not called by any other function is presented at the top level. It is possible to expand the tree to see called functions. Only functions available in the `elf` file can be visible in the tab.

When used, the **[Search...]** button triggers the display of the functions matching the characters in the search field. The search can be made case sensitive or not depending on the selection in checkbox **[Case sensitive]**.

Figure 235. Static Stack Analyzer Call Graph tab

Function	Depth	Max cost	Local cost	Type	Location	Info
ADC_IRQHandler	?	?	0			Max cost uncertain. Recursive. No stack usage information available for this...
Reset_Handler	16	248	0			Max cost uncertain. No stack usage information available for this function
_swrite	3	48	0			Max cost uncertain. No stack usage information available for this function
_sread	2	32	0			Max cost uncertain. No stack usage information available for this function
_sseek	2	24	0			Max cost uncertain. No stack usage information available for this function
_sclose	2	16	0			Max cost uncertain. No stack usage information available for this function
SysTick_Handler	1	12	8	STATIC	stm32f4xx_it.c:182	
NMI_Handler	0	4	4	STATIC	stm32f4xx_it.c:70	
DebugMon_Handler	0	4	4	STATIC	stm32f4xx_it.c:156	
MemManage_Handler	0	4	4	STATIC	stm32f4xx_it.c:98	
UsageFault_Handler	0	4	4	STATIC	stm32f4xx_it.c:128	
PendSV_Handler	0	4	4	STATIC	stm32f4xx_it.c:169	
BusFault_Handler	0	4	4	STATIC	stm32f4xx_it.c:113	
HardFault_Handler	0	4	4	STATIC	stm32f4xx_it.c:83	
SVC_Handler	0	4	4	STATIC	stm32f4xx_it.c:143	
frame_dummy	0	0	0			Max cost uncertain. No stack usage information available for this function

The column information in the *Call Graph* tab is described in Table 27.

Table 27. Static Stack Analyzer Call Graph tab details

Name	Description
Function	Function name.
Depth	Specifies the call stack depth this function uses: <ul style="list-style-type: none"> 0: the function does not call any other functions Number ≥ 1: the function calls other functions ?: the function makes recursive calls or the depth cannot be calculated
Max cost	Specifies how many bytes of stack the function uses including stack needed for called functions.
Local cost	Specifies how many bytes of stack the function uses. This column does not take into account any stack that may be needed by the functions it may call.
Type	Specifies if the function uses a STATIC or DYNAMIC stack allocation. <ul style="list-style-type: none"> STATIC: the function uses a fixed stack DYNAMIC: the function uses a run-time dependent stack Empty field: no stack usage information available for the function
Location	Indicates where the function is declared. It is possible to double-click on a line and open the file with the defined function in the editor.
Info	Contains specific information about the stack usage calculation. For instance, it can hold a combination of the following messages: <ul style="list-style-type: none"> Max cost uncertain: the reason can be that the function makes a call to some sub-function where the stack information is not known, the function makes recursive calls, or others Recursive: the function makes recursive calls No stack usage information available for this function: no stack usage information available for this function Local cost uncertain due to dynamic size, verify at run-time: the function allocates stack dynamically, for instance depending on a parameter

The *Call Graph* tab sort order can be changed by clicking on a column name.

By double-clicking on a line that displays the file location and line number in the tab, the function is opened in the *Editor* view.

Note: *The main function is usually called by the Reset_Handler and can in those cases be seen when expanding the Reset_Handler node.*

If unused functions are listed in the tab, check if linker option [dead code removal] is enabled to remove unused code from the program. Read more on this in Section 2.5.2 Discard unused sections.

The small icon left of the function name in column *Function* column indicates the following:

- Green dot: the function uses STATIC stack allocation (fixed stack).
- Blue square: the function uses DYNAMIC stack allocation (run-time dependent).
- 010 icon: used if the stack information is not known. This can be the case for library functions or assembler functions.
- Three arrows in a circle: used in the *Call Graph* tab when the function makes recursive calls.

Figure 236. Function symbols in Static Stack Analyzer

The screenshot shows the IDE interface for the Static Stack Analyzer. The window title is 'Static Stack Analyzer'. The main content area shows the analysis for 'NUCLEO-F401RE.elf - /NUCLEO-F401RE/Debug'. There are two tabs: 'List' and 'Call graph'. The 'List' tab is active, displaying a table of functions and their stack depths.

Function	Depth
ADC_IRQHandler	?
Reset_Handler	16
LoopCopyDataInit	15
LoopFillZerobss	14
main	13
SystemClock_Config	5
MX_USART2_UART_Init	4
MX_GPIO_Init	1
SystemCoreClockUpdate	0
iprintf	12
readTemp	0
readSpeed	0
writeSpeed	0
writeTemp	0
SystemInit	0
FillZerobss	0

9.2.4 Using the filter and search field

The *List* and *Call Graph* tabs contain a filter/search field, which can be used to search a specific function or functions matching the characters entered in the field.

Figure 237 displays the *List* tab where the [Filter] field is used to seek functions containing the “read” string in their name.

Figure 237. Static Stack Analyzer List tab using search

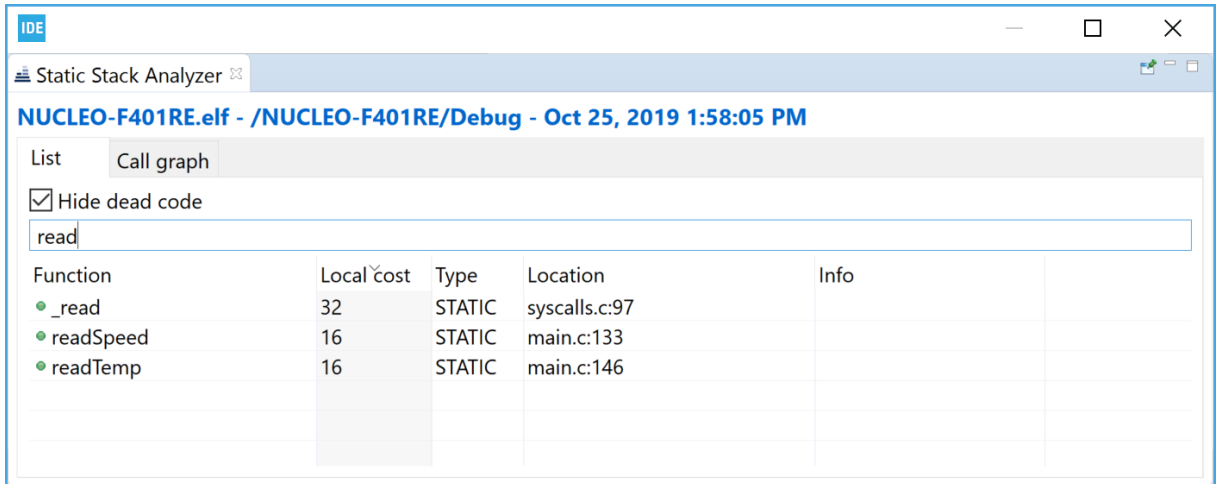
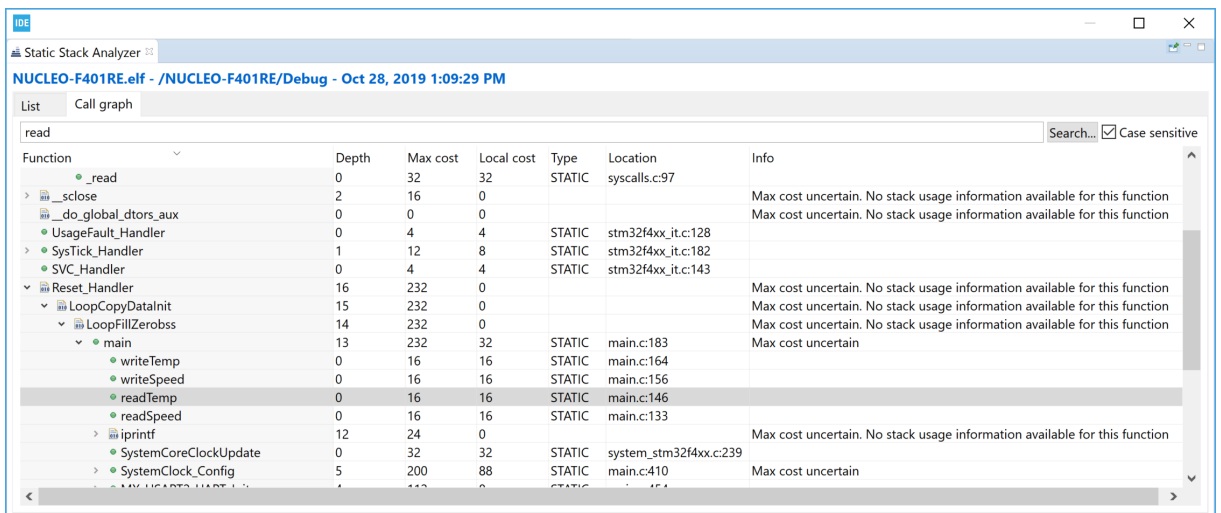


Figure 238 shows a use example of the [Search...] field in the *Call Graph* tab for filtering functions with name matching the “read” string.

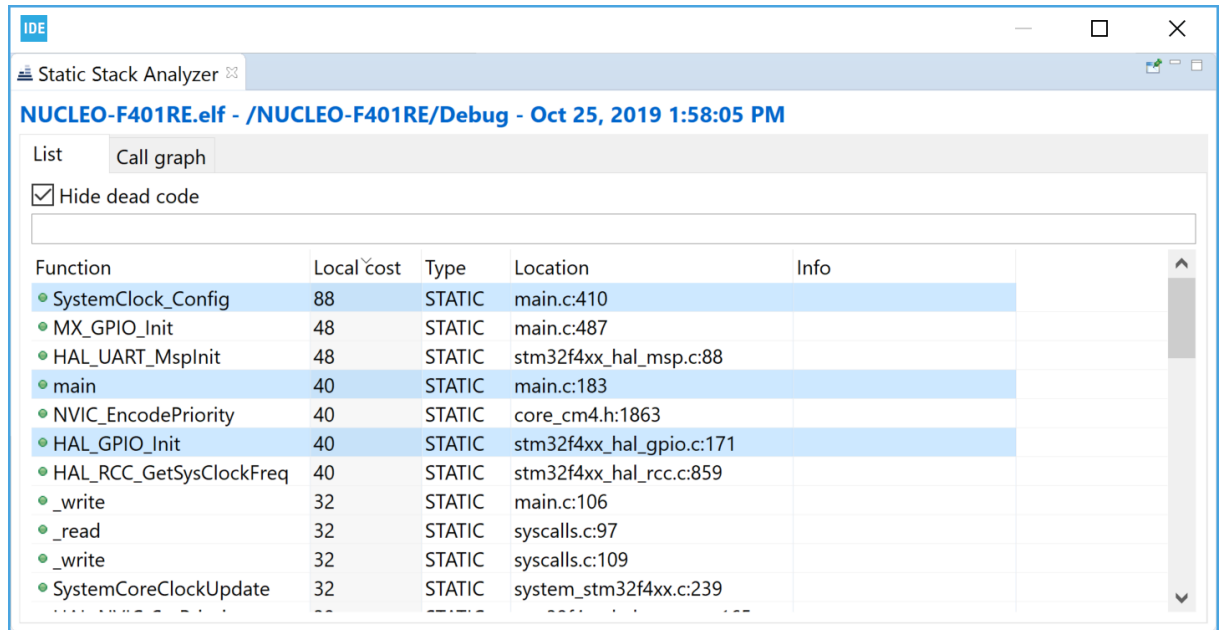
Figure 238. Static Stack Analyzer Call Graph using search



9.2.5 Copy and paste

The data in the *List* tab can be copied to other applications in CSV format by selecting the rows to copy and typing **Ctrl+C**. The copied data can be pasted into another application with the **Ctrl+V** command.

Figure 239. Copy and paste



The **Ctrl+C** copy of the lines selected in Figure 239 provides the **Ctrl+V** results below:

```
"SystemClock_Config";"88";"STATIC";"main.c:410";""
"main";"40";"STATIC";"main.c:183";""
"HAL_GPIO_Init";"40";"STATIC";"stm32f4xx_hal_gpio.c:171";""
```

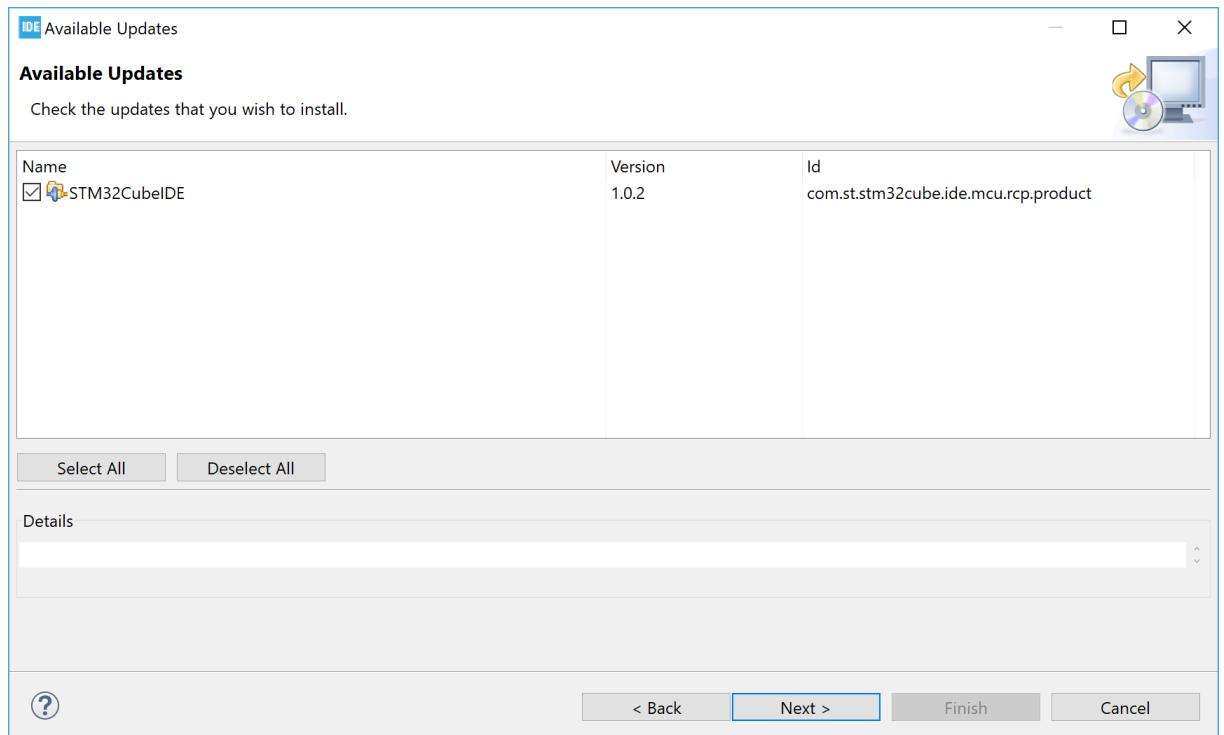
10 Installing updates and additional Eclipse® plugins

10.1 Check for updates

STM32CubeIDE checks for available updates regularly and opens the *Available Updates* dialog when a new update is detected. It is also possible to check for updates manually. Use menu **[Help]>[Check for Updates]** to check if new software is available.

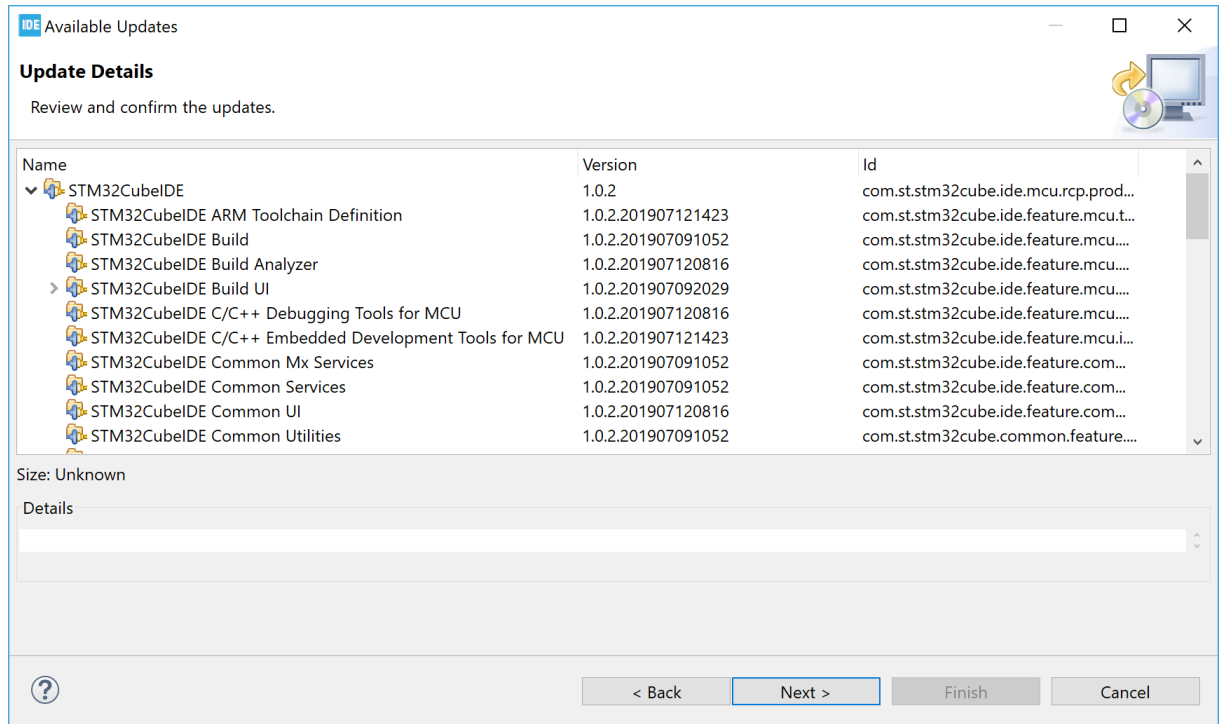
When updates are found, select the update to install and press **[Next]**.

Figure 240. STM32CubeIDE available updates



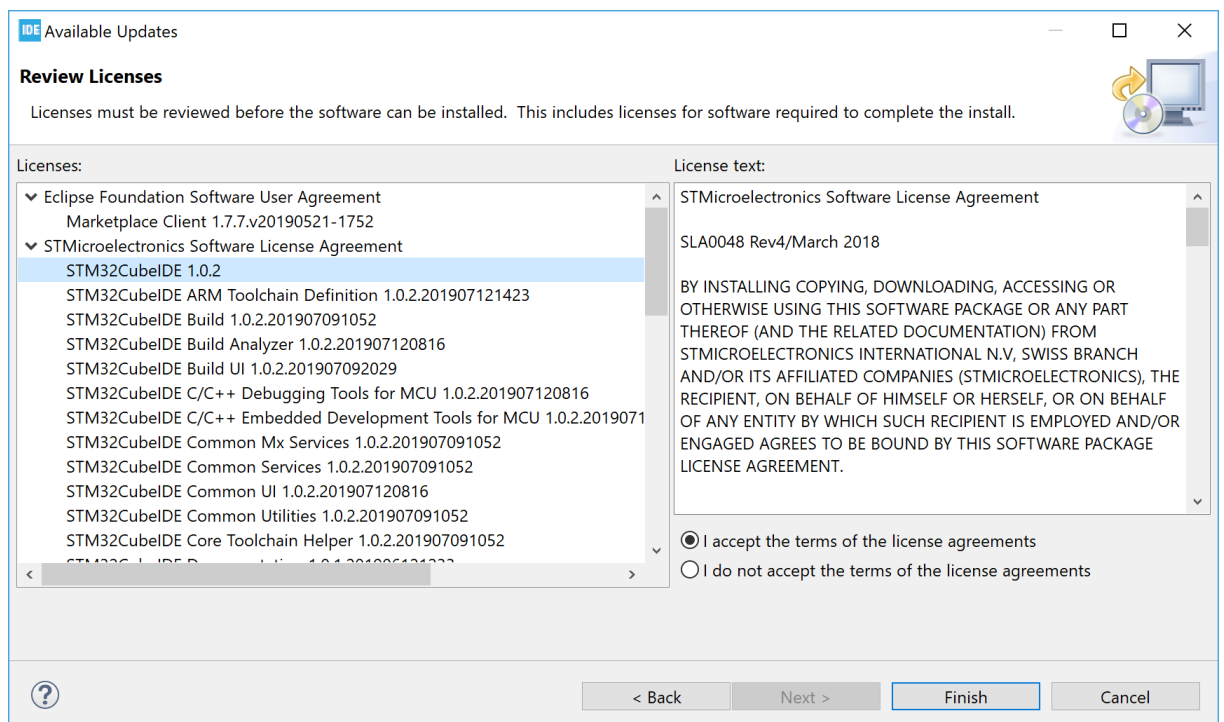
Update details is displayed. Review and confirm the update. Press **Next**.

Figure 241. STM32CubeIDE update details



Review Licenses details are displayed. Review the licenses, select **[I accept the terms of the license agreements]** and press **[Finish]** to install the update.

Figure 242. STM32CubeIDE update review licenses

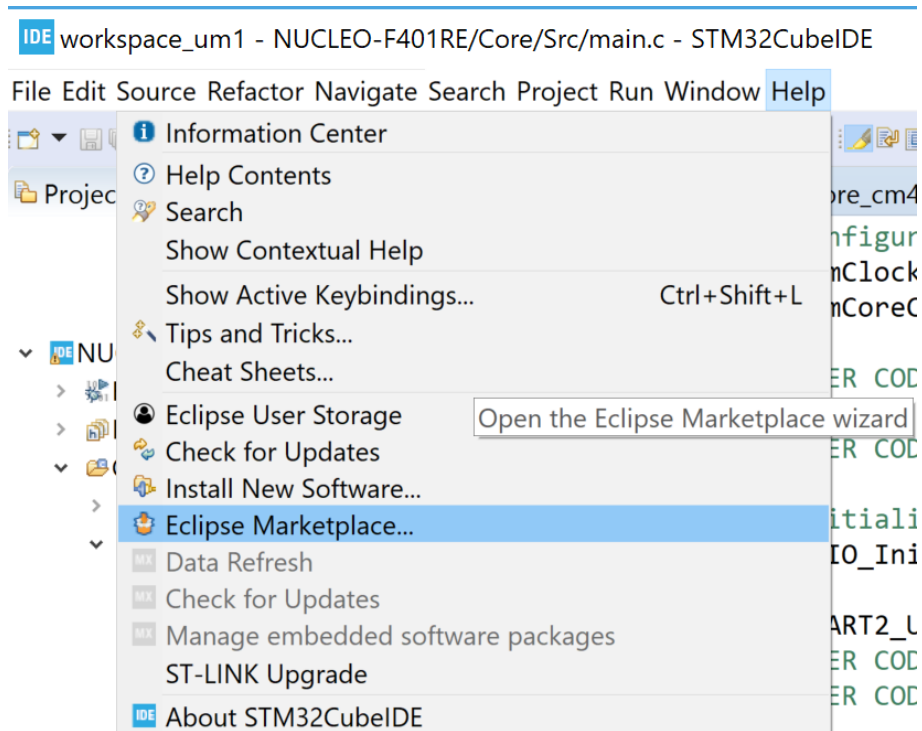


The progress bar displayed at the bottom of the STM32CubeIDE window shows the installation completion rate. Restart STM32CubeIDE when the update is finished.

10.2 Install from the Eclipse® market place

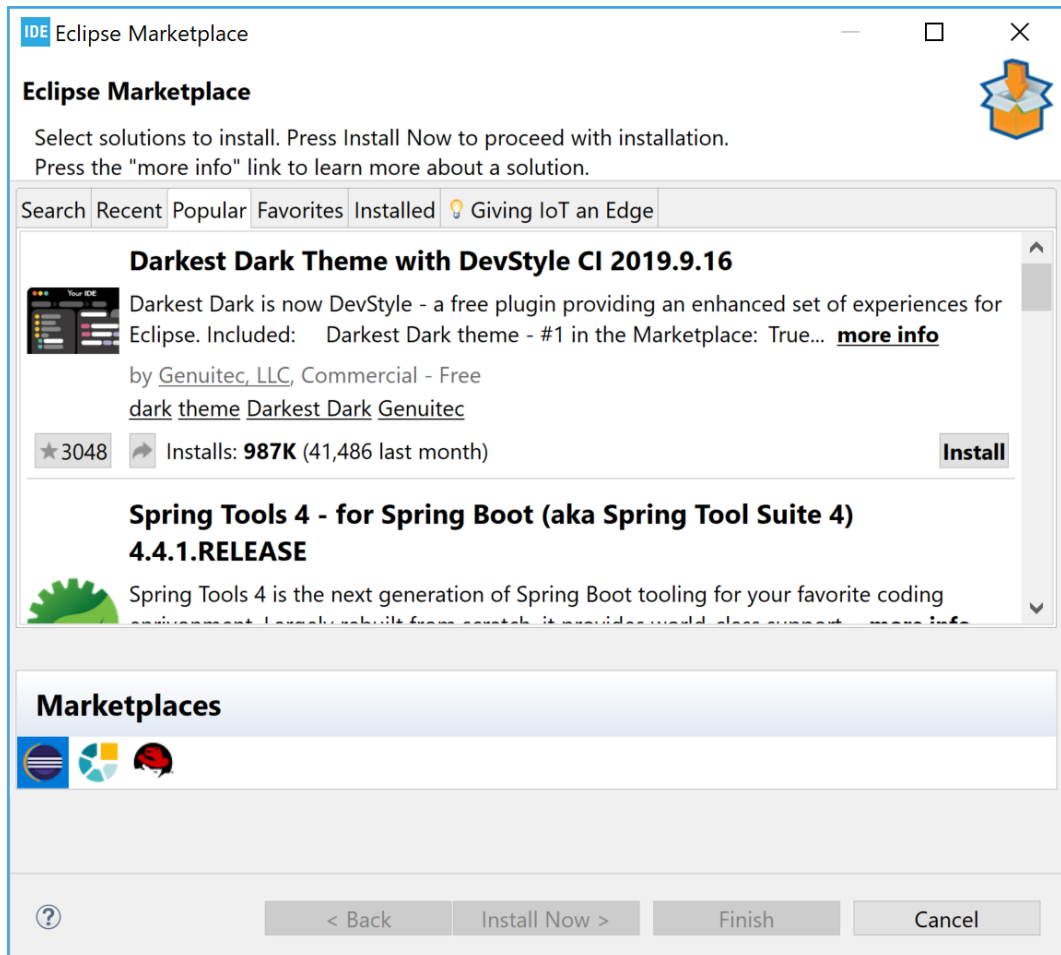
It is possible to install additional third-party Eclipse® plugins in STM32CubeIDE using the Eclipse Marketplace. To install from Eclipse Marketplace, select menu **[Help]>[Eclipse Marketplace...]**.

Figure 243. Eclipse Marketplace menu



The *Eclipse Marketplace* dialog opens. Search for the plugin or use the tabs (*Recent*, *Popular*, *Favorites*) to find the software wanted and install it.

Figure 244. Eclipse marketplace

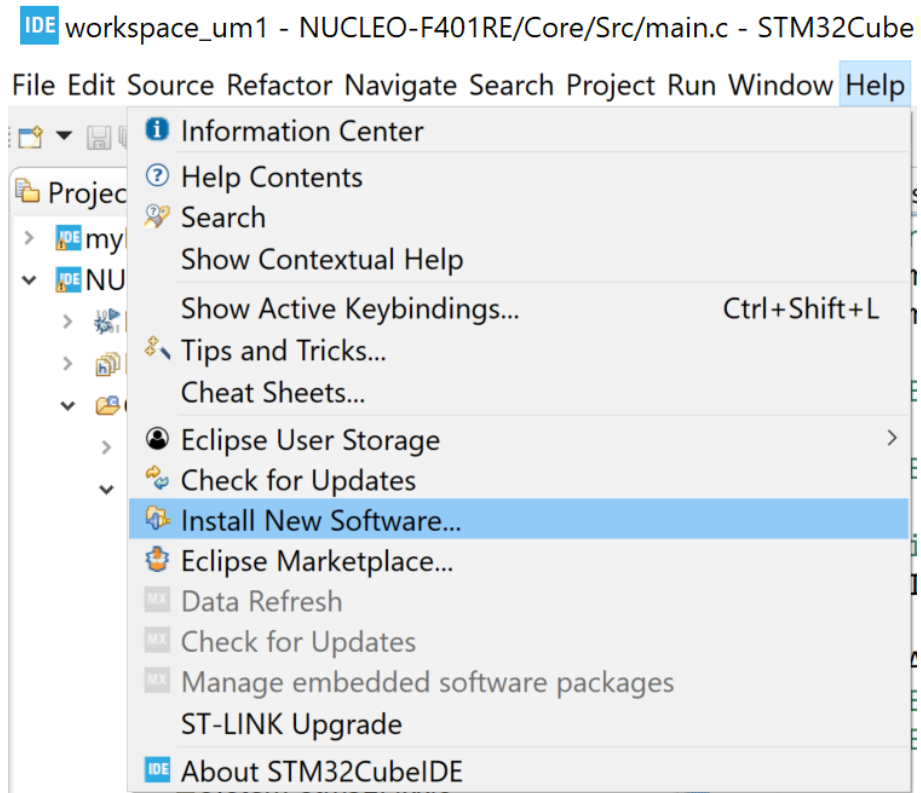


Wait until the installation is finished and restart STM32CubeIDE.

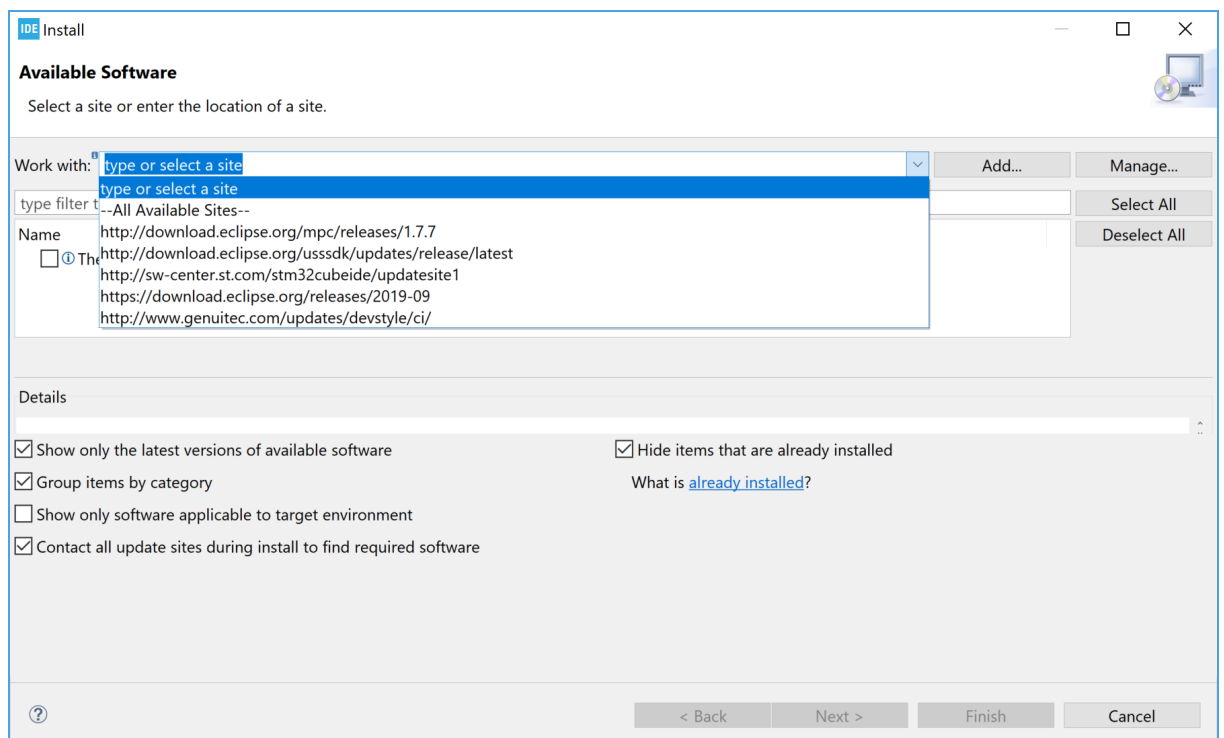
10.3 Install using [Install new software...]

Another way to install new software is to use menu [Help]>[Install New Software...].

Note: When installing a new toolchain, it is recommended to use the Toolchain Manager described in Section 2.11 Toolchain Manager.

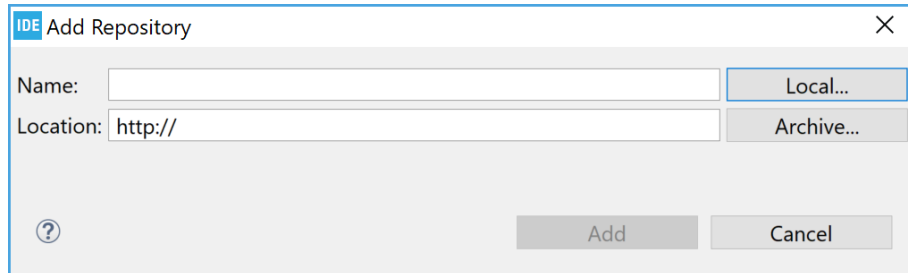
Figure 245. Install new software menu


The *Install* dialog opens. Enter the plugin update site URL. If the URL is not known, use [--All Available Sites--].

Figure 246. Install new software


If no direct Internet connection is available, the plugin can be downloaded into an archive on a computer with an Internet connection, and then manually transferred to the computer with an STM32CubeIDE installation. Add the archived file by clicking on the **[Add...]** button and then select **[Archive and select the downloaded file]**.

Figure 247. Install new software from computer



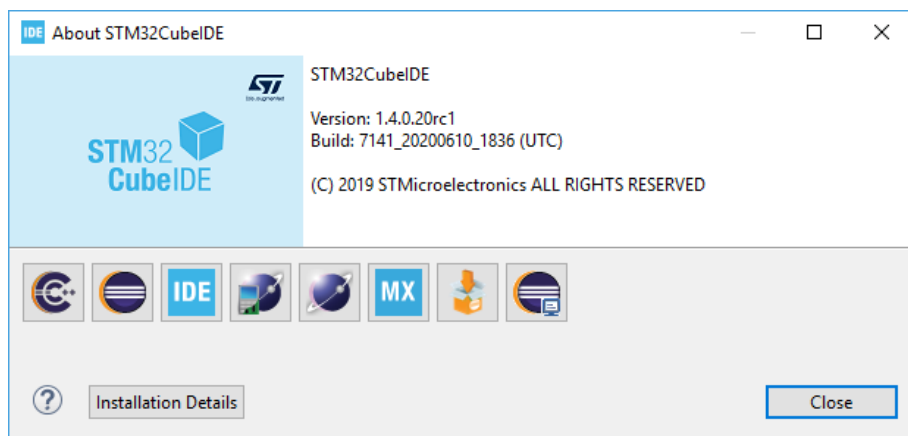
Select the appropriate plugins and install the software. Restart STM32CubeIDE when installation is finished.

Remember: Not all Eclipse® plugins are compatible with STM32CubeIDE.

10.4 Uninstalling installed additional Eclipse® plugins

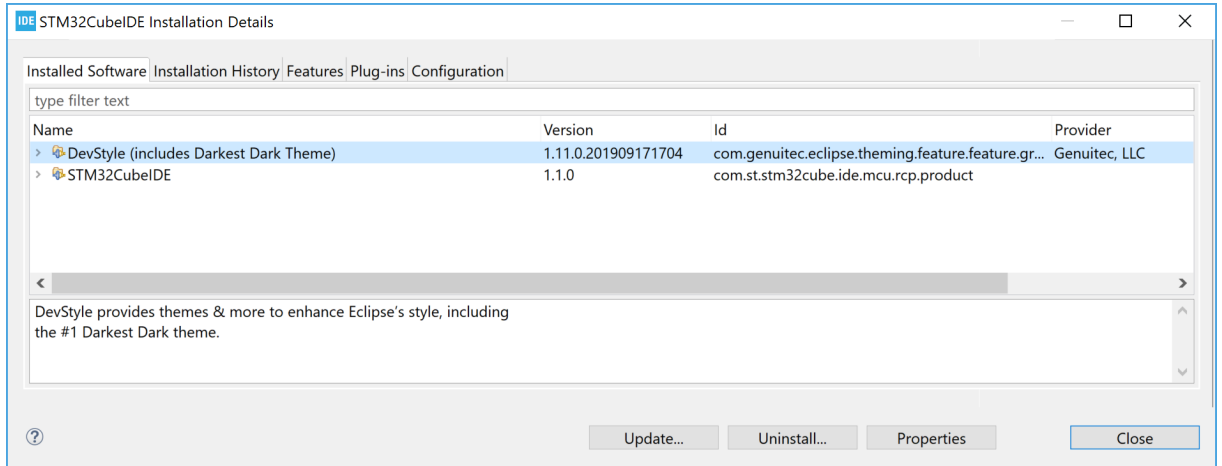
To uninstall a plugin that is no longer needed, select menu **[Help]>[About STM32CubeIDE]**.

Figure 248. About STM32CubeIDE



Press the [**Installation Details**] button to open the *STM32CubeIDE Installation Details* dialog.

Figure 249. Installation details

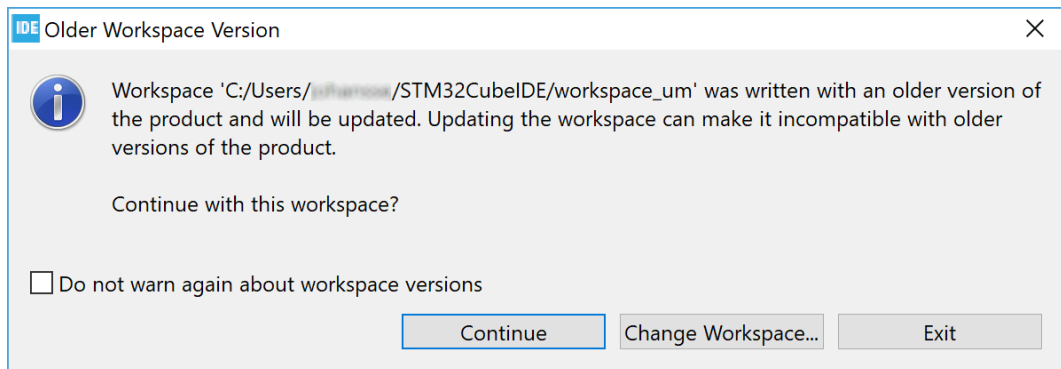


Select the plugin to uninstall in the *Installed Software* tab and press [**Uninstall...**]. Restart STM32CubeIDE when the uninstallation is finished.

10.5 Update to new CDT™

When a new version of *STM32CubeIDE* is installed based on a new version of Eclipse®, CDT™ or both, it is recommended to create a new workspace instead of using a former workspace. The following warning is displayed when trying to use an old workspace with a new *STM32CubeIDE*.

Figure 250. Older workspace version warning



11 References

Table 28. STMicroelectronics reference documents

Reference	Document short name	Description	Document source
[ST-01]	DB3871	STM32CubeIDE data brief	www.st.com
[ST-02]	RN0114	STM32CubeIDE release note	
[ST-03]	UM2553	STM32CubeIDE quick start guide	
[ST-04]	UM2563	STM32CubeIDE installation guide	
[ST-05]	UM2578	Migration guide from TrueSTUDIO® to STM32CubeIDE	
[ST-06]	UM2579	Migration guide from System Workbench to STM32CubeIDE	
[ST-07]	UM2576	STM32CubeIDE ST-LINK GDB server	
[ST-08]	Getting started with projects based on the STM32MP1 Series in STM32CubeIDE ⁽¹⁾		Refer to STM32CubeIDE in the “Tools” section of wiki.st.com/stm32mpu
[ST-09]	AN5361	Getting started with projects based on dual-core STM32H7 microcontrollers in STM32CubeIDE	www.st.com
[ST-10]	AN5394	Getting started with projects based on the STM32L5 Series in STM32CubeIDE	
[ST-11]	AN5564	Getting started with projects based on dual-core STM32WL microcontrollers in STM32CubeIDE	
[ST-12]	AN4296	Use STM32F3/STM32G4 CCM SRAM with IAR Embedded Workbench®, Keil® MDK-ARM, STMicroelectronics STM32CubeIDE and other GNU-based toolchains	
[ST-13]	SLA0048	License agreement applicable to STM32CubeIDE	
[ST-14]	UM1718	STM32CubeMX for STM32 configuration and initialization C code generation	

1. Legacy application note [AN5360](#) remains available on www.st.com.

Table 29. External reference documents

Reference	Description	Document source
[EXT-01]	GNU Assembler	GNU tool suite ⁽¹⁾
[EXT-02]	GNU Compiler Collection	
[EXT-03]	GNU C Library	
[EXT-04]	GNU C Preprocessor	
[EXT-05]	GNU Linker	
[EXT-06]	GNU Binary Utilities	
[EXT-07]	Red Hat Newlib C Library	
[EXT-08]	Red Hat Newlib C Math Library	
[EXT-09]	Newlib nano readme	
[EXT-10]	Debugging with GDB	
[EXT-11]	GDB Quick Reference Card	
[EXT-12]	GNU Tools for STM32 Patch list	<i>Information Center</i>

1. For GNU documentation principles, refer to www.gnu.org.

Revision history

Table 30. Document revision history

Date	Revision	Changes
24-Jul-2020	1	Initial release.
2-Nov-2020	2	Document updated for STM32CubeIDE v1.5.0: <ul style="list-style-type: none"> • Only one toolchain installed by default • The <i>SFRs</i> view displays Arm® Cortex® core registers node • Debug with OpenOCD supports SWV and live expressions • Added <i>Preferences - Build variables</i> • Added <i>Toolchain Manager</i> • Added <i>RTOS-aware debugging</i> with FreeRTOS™ information • Added <i>General debug and run launch flow</i> • Added <i>Post-build with makefile targets</i>
18-Feb-2021	3	Document updated for STM32CubeIDE v1.6.0: <ul style="list-style-type: none"> • Added the <i>Azure RTOS ThreadX</i> section into chapter <i>RTOS-aware debugging</i>. Reorganized the <i>FreeRTOS</i> section • Updated the <i>Toolchain Manager</i> section for the support of local toolchains • Updated the <i>Project C/C++ build settings</i> section, MCU toolchain selection moved • Updated <i>Information Center</i> • Updated the entire document for the “<i>SWV packet</i>” terminology • Updated <i>References</i> • Removed <i>Section 4.3.3 SWV Exception Timeline Graph</i>
5-Jul-2021	4	Document updated for STM32CubeIDE v1.7.0: <ul style="list-style-type: none"> • Added Section 2.7 Thread-safe wizard for empty projects and CDT projects • Added Section 3.8 Import STM32 Cortex-M executable • Added Section 6.3 RTOS-kernel-aware debug • Updated Information Center – Home page • Updated Headless build description • Updated Section 2.5.6 Linker script with new memory map layout figure and additional description • Updated Position-independent code description • Updated debug configuration descriptions for ST-LINK GDB server, OpenOCD and SEGGER in Debug using different GDB servers • Updated FreeRTOS Task List view

Contents

1	Getting started	2
1.1	Product information	2
1.1.1	System requirements	3
1.1.2	Downloading the latest STM32CubeIDE version	3
1.1.3	Installing STM32CubeIDE	3
1.1.4	License	4
1.1.5	Support	4
1.2	Using STM32CubeIDE	4
1.2.1	Basic concept and terminology	4
1.2.2	Starting STM32CubeIDE	5
1.2.3	Help system	6
1.3	<i>Information Center</i>	6
1.3.1	Accessing the <i>Information Center</i>	7
1.3.2	Home page	7
1.3.3	Technical documentation	8
1.3.4	Closing the <i>Information Center</i>	9
1.4	Perspectives, editors and views	9
1.4.1	Perspectives	9
1.4.2	Editors	13
1.4.3	Views	13
1.4.4	<i>Quick Access</i> edit field	15
1.5	Configuration - Preferences	16
1.5.1	Preferences - Editors	17
1.5.2	Preferences - Code style formatter	18
1.5.3	Preferences - Network proxy settings	20
1.5.4	Preferences - Build variables	21
1.6	Workspaces and projects	22
1.7	Managing existing workspaces	22
1.7.1	Backup of preferences for a workspace	23
1.7.2	Copy preferences between workspaces	23
1.7.3	Keeping track of Java heap space	23

1.7.4	Unavailable workspace	23
1.8	STM32CubeIDE and Eclipse® basics	24
1.8.1	Keyboard shortcuts	24
1.8.2	Editor zoom in and zoom out	26
1.8.3	Quickly find and open a file	26
1.8.4	Branch folding	27
1.8.5	Block selection mode	27
1.8.6	Compare files	30
1.8.7	Local file history	32
2	Creating and building C/C++ projects	37
2.1	Introduction to projects	37
2.2	Creating a new STM32 project	37
2.2.1	Creating a new STM32 executable project	37
2.2.2	Creating a new STM32 static library project	42
2.3	Configure the project build setting	44
2.3.1	Project build configuration	44
2.3.2	Project C/C++ build settings	50
2.4	Building the project	57
2.4.1	Building all projects	58
2.4.2	Build all build configurations	58
2.4.3	Headless build	59
2.4.4	Temporary assembly file and preprocessed C code	60
2.4.5	Build logging	60
2.4.6	Parallel build and build behaviour	60
2.4.7	Post-build with makefile targets	61
2.5	Linking the project	61
2.5.1	Run-time library	62
2.5.2	Discard unused sections	64
2.5.3	Page size allocation for malloc	65
2.5.4	Include additional object files	66
2.5.5	Treat linker warnings and errors	66
2.5.6	Linker script	68

2.5.7	Modify the linker script	75
2.5.8	Include libraries	82
2.5.9	Referring to projects	84
2.6	I/O redirection	85
2.6.1	printf() redirection	85
2.7	Thread-safe wizard for empty projects and CDT™ projects	87
2.8	Position-independent code	94
2.8.1	Adding the -fPIE option	95
2.8.2	Run-time library	96
2.8.3	Stack pointer configuration	96
2.8.4	Interrupt vector table	96
2.8.5	Global offset table	96
2.8.6	Interrupt vector table and symbols	97
2.8.7	Debugging position-independent code	97
2.9	Exporting projects	99
2.10	Importing existing projects	101
2.10.1	Importing an STM32CubeIDE project	101
2.10.2	Importing System Workbench and TrueSTUDIO® projects	103
2.10.3	Importing using project files association	106
2.10.4	Prevent “GCC not found in path” error	106
2.11	Toolchain Manager	106
2.11.1	Install new toolchain	109
2.11.2	Manage default toolchain	112
2.11.3	Uninstall toolchain	113
2.11.4	Using local toolchain	114
2.11.5	Network error	118
3	Debugging	119
3.1	Introduction to debugging	119
3.1.1	General debug and run launch flow	120
3.2	Debug configurations	121
3.2.1	Debug configuration	122
3.2.2	Main tab	122

3.2.3	Debugger tab	123
3.2.4	Startup tab	125
3.3	Manage debug configurations	128
3.4	Debug using different GDB servers	129
3.4.1	Debug using the ST-LINK GDB server	129
3.4.2	Debug using OpenOCD and ST-LINK	132
3.4.3	Debug using SEGGER J-Link	134
3.5	Start and stop debugging	136
3.5.1	Start debugging	136
3.5.2	Debug perspective and views	138
3.5.3	Main controls for debugging	139
3.5.4	Run, start and stop a program	141
3.5.5	Set breakpoints	141
3.5.6	Attach to running target	143
3.5.7	Restart or terminate debugging	145
3.6	Debug features	148
3.6.1	Live Expressions view	148
3.6.2	Shared ST-LINK	148
3.6.3	Debug multiple boards	149
3.6.4	STM32H7 multicore debugging	149
3.6.5	STM32MP1 debugging	149
3.6.6	STM32L5 debugging	149
3.7	Run configurations	150
3.8	Import STM32 Cortex [®] -M executable	151
4	Debug with Serial Wire Viewer tracing (SWV)	156
4.1	Introduction to SWV and ITM	156
4.2	SWV debugging	156
4.2.1	SWV debug configuration	156
4.2.2	SWV settings configuration	159
4.2.3	SWV tracing	161
4.3	SWV views	162
4.3.1	SWV Trace Log	163

4.3.2	SWV Exception Trace Log	163
4.3.3	SWV Data Trace	165
4.3.4	SWV Data Trace Timeline Graph	167
4.3.5	SWV ITM Data Console and <code>printf</code> redirection	167
4.3.6	SWV Statistical Profiling	169
4.4	Change the SWV trace buffer size	171
4.5	Common SWV problems	172
5	Special Function Registers (SFRs)	174
5.1	Introduction to SFRs	174
5.2	Using the <i>SFRs</i> view	174
5.3	Updating CMSIS-SVD settings	176
6	RTOS-aware debugging	178
6.1	Azure [®] RTOS ThreadX	178
6.1.1	Finding the views	178
6.1.2	<i>ThreadX Thread List</i> view	178
6.1.3	<i>ThreadX Semaphores</i> view	180
6.1.4	<i>ThreadX Mutexes</i> view	181
6.1.5	<i>ThreadX Message Queues</i> view	182
6.1.6	<i>ThreadX Event Flags</i> view	182
6.1.7	<i>ThreadX Timers</i> view	183
6.1.8	<i>ThreadX Memory Block Pools</i> view	184
6.1.9	<i>ThreadX Memory Byte Pools</i> view	184
6.2	FreeRTOS [™]	185
6.2.1	Requirements	185
6.2.2	Finding the views	187
6.2.3	<i>FreeRTOS Task List</i> view	187
6.2.4	<i>FreeRTOS Timers</i> view	189
6.2.5	<i>FreeRTOS Semaphores</i> view	190
6.2.6	<i>FreeRTOS Queues</i> view	191
6.3	RTOS-kernel-aware debug	191
7	Fault Analyzer	195
7.1	Introduction to the Fault Analyzer	195

7.2	Using the <i>Fault Analyzer</i> view	196
8	<i>Build Analyzer</i>	199
8.1	Introduction to the <i>Build Analyzer</i>	199
8.2	Using the <i>Build Analyzer</i>	199
8.2.1	<i>Memory Regions</i> tab	199
8.2.2	<i>Memory Details</i> tab	200
9	<i>Static Stack Analyzer</i>	207
9.1	Introduction to the <i>Static Stack Analyzer</i>	207
9.2	Using the <i>Static Stack Analyzer</i>	208
9.2.1	Enable stack usage information	209
9.2.2	<i>List</i> tab	210
9.2.3	<i>Call Graph</i> tab	211
9.2.4	Using the filter and search field	212
9.2.5	Copy and paste	214
10	Installing updates and additional Eclipse® plugins	215
10.1	Check for updates	215
10.2	Install from the Eclipse® market place	217
10.3	Install using [Install new software...].	218
10.4	Uninstalling installed additional Eclipse® plugins	220
10.5	Update to new CDT™	221
11	References	222
	Revision history	224
	Contents	225
	List of tables	231
	List of figures	232

List of tables

Table 1.	Examples of toolchain build variables	21
Table 2.	Key shortcut examples	25
Table 3.	Memory map layout	69
Table 4.	<i>Toolchain Manager</i> column details	108
Table 5.	<i>Toolchain Manager</i> button information	108
Table 6.	<i>SWV Trace Log</i> columns details	163
Table 7.	<i>SWV Exception Trace Log – Data</i> columns details	164
Table 8.	<i>SWV Exception Trace Log – Statistics</i> columns details	165
Table 9.	<i>SWV Data Trace</i> columns details	167
Table 10.	<i>SWV Statistical Profiling</i> columns details	171
Table 11.	<i>ThreadX Thread List</i> details	180
Table 12.	<i>ThreadX Semaphores</i> details	181
Table 13.	<i>ThreadX Mutexes</i> details	181
Table 14.	<i>ThreadX Message Queues</i> details	182
Table 15.	<i>ThreadX Event Flags</i> details	183
Table 16.	<i>ThreadX Timers</i> details	183
Table 17.	<i>ThreadX Memory Block Pools</i> details	184
Table 18.	<i>ThreadX Memory Byte Pools</i> details	185
Table 19.	<i>FreeRTOS Task List</i> details	189
Table 20.	<i>FreeRTOS Timers</i> details	190
Table 21.	<i>FreeRTOS Semaphores</i> details	191
Table 22.	<i>FreeRTOS Queues</i> details	191
Table 23.	<i>Memory Regions</i> tab information	200
Table 24.	<i>Memory Regions</i> usage color	200
Table 25.	<i>Memory Details</i> tab information	201
Table 26.	<i>Static Stack Analyzer List</i> tab details	210
Table 27.	<i>Static Stack Analyzer Call Graph</i> tab details	211
Table 28.	STMicroelectronics reference documents	222
Table 29.	External reference documents	223
Table 30.	Document revision history	224

List of figures

Figure 1.	STM32CubeIDE key features	3
Figure 2.	STM32CubeIDE window	5
Figure 3.	STM32CubeIDE Launcher – Workspace selection	6
Figure 4.	Help menu	6
Figure 5.	Help - Information Center menu	7
Figure 6.	Information Center – Home page	7
Figure 7.	Information Center – Technical documentation page	8
Figure 8.	Reset perspective	9
Figure 9.	Toolbar buttons for switching perspective	9
Figure 10.	C/C++ perspective	10
Figure 11.	Debug perspective	11
Figure 12.	Device Configuration Tool perspective	11
Figure 13.	Remote System Explorer perspective	12
Figure 14.	New connection	12
Figure 15.	[Show View] menu	13
Figure 16.	Show View dialog	14
Figure 17.	Quick access	15
Figure 18.	Preferences	16
Figure 19.	Preferences - Text Editors	17
Figure 20.	Preferences - Formatter	18
Figure 21.	Preferences - Code style edit	19
Figure 22.	Preferences - Network Connections	20
Figure 23.	Preferences – Build variables	21
Figure 24.	Pre-build step using build variables	21
Figure 25.	Preferences - Workspaces	22
Figure 26.	Display of Java heap space status	23
Figure 27.	Workspace unavailable	24
Figure 28.	Shortcut keys	24
Figure 29.	Shortcut preferences	25
Figure 30.	Editor with text zoomed in	26
Figure 31.	Editor folding	27
Figure 32.	Editor block selection	28
Figure 33.	Editor text block addition	28
Figure 34.	Editor column block selection	29
Figure 35.	Editor column block paste	30
Figure 36.	Editor - Compare files	31
Figure 37.	Editor - File differences	31
Figure 38.	Local history	32
Figure 39.	Show local history	33
Figure 40.	File history	34
Figure 41.	Compare current history with local history	35
Figure 42.	Compare local file differences	36
Figure 43.	STM32 target selection	38
Figure 44.	STM32 board selection	38
Figure 45.	Project setup	39
Figure 46.	Firmware library package setup	40
Figure 47.	Initialization of all peripherals	40
Figure 48.	STM32CubeMX perspective opening	41
Figure 49.	Project creation started	41
Figure 50.	STM32CubeMX	42
Figure 51.	STM32 static library project	43
Figure 52.	STM32 library and board project	44

Figure 53.	Set the active build configuration using the toolbar	45
Figure 54.	Set active build configuration using right-click	46
Figure 55.	Set active build configuration using menu	47
Figure 56.	<i>Manage Configurations</i> dialog	47
Figure 57.	Create a new build configuration	48
Figure 58.	Updated <i>Manage Configurations</i> dialog	48
Figure 59.	Configuration deletion dialog	49
Figure 60.	Configuration renaming dialog	49
Figure 61.	Properties tabs	50
Figure 62.	Properties configurations	50
Figure 63.	Properties toolchain version	51
Figure 64.	Properties toolchain selection	51
Figure 65.	Properties tool MCU settings	52
Figure 66.	Properties tool MCU post-build settings	53
Figure 67.	Properties tool GCC assembler settings	54
Figure 68.	Properties tool GCC compiler settings	55
Figure 69.	Properties tool GCC linker settings	56
Figure 70.	Properties build steps settings	57
Figure 71.	Project build toolbar	57
Figure 72.	Project build console	58
Figure 73.	Project build all	58
Figure 74.	Project build-all configurations	59
Figure 75.	Headless build	60
Figure 76.	Parallel build	61
Figure 77.	Linker documentation	62
Figure 78.	Linker run-time library	63
Figure 79.	Linker newlib-nano library and floating-point numbers	64
Figure 80.	Linker discard unused sections	65
Figure 81.	Linker include additional object files	66
Figure 82.	Linker fatal warnings	68
Figure 83.	Linker memory output	80
Figure 84.	Linker memory output specified order	80
Figure 85.	Linker memory displaying file <code>readme</code>	81
Figure 86.	Include a library	83
Figure 87.	Add library header files to the include paths	84
Figure 88.	Set project references	85
Figure 89.	Select a wizard	88
Figure 90.	<i>Thread-Safe Solution</i> wizard	89
Figure 91.	Thread-safe source folder location	90
Figure 92.	Thread-safe strategy selection	92
Figure 93.	Thread-safe properties	93
Figure 94.	Thread-safe files	93
Figure 95.	Thread-safe error dialog	94
Figure 96.	Position independent code, <code>-fPIE</code>	95
Figure 97.	Debugging position independent code	98
Figure 98.	Export project	99
Figure 99.	Export dialog	100
Figure 100.	Export archive	101
Figure 101.	Import project	102
Figure 102.	Import dialog	102
Figure 103.	Import projects	103
Figure 104.	Import System Workbench projects (1 of 3)	104
Figure 105.	Import System Workbench projects (2 of 3)	105
Figure 106.	Import System Workbench projects (3 of 3)	105

Figure 107.	Import using project files association	106
Figure 108.	Open <i>Toolchain Manager</i>	107
Figure 109.	<i>Toolchain Manager</i>	107
Figure 110.	Install toolchain.	109
Figure 111.	Check items to install.	109
Figure 112.	Review items to install	110
Figure 113.	Review and accept licenses	110
Figure 114.	Security warning	111
Figure 115.	Restart to apply software update	111
Figure 116.	Toolchain installed.	111
Figure 117.	Default toolchain.	112
Figure 118.	Default toolchain updated	112
Figure 119.	Uninstall toolchain.	113
Figure 120.	Uninstall details	113
Figure 121.	Software updates	114
Figure 122.	Toolchain uninstalled.	114
Figure 123.	Add local toolchain	115
Figure 124.	Specify local toolchain location	116
Figure 125.	Specify local toolchain prefix	116
Figure 126.	Local toolchain added	117
Figure 127.	Edit local toolchain	117
Figure 128.	Toolchain network error	118
Figure 129.	General debug and run launch flowchart	120
Figure 130.	Debug as STM32 MCU	121
Figure 131.	Debug as STM32 MCU menu	122
Figure 132.	Debug configuration main tab	123
Figure 133.	Debug configuration debugger tab	124
Figure 134.	GDB server command line dialog	125
Figure 135.	Debug configuration startup tab	126
Figure 136.	Add/Edit item	127
Figure 137.	Manage debug configurations.	128
Figure 138.	Manage debug configurations toolbar	128
Figure 139.	ST-LINK GDB server debugger tab	130
Figure 140.	OpenOCD debugger tab	133
Figure 141.	SEGGER debugger tab	135
Figure 142.	Debug configurations	137
Figure 143.	Confirm perspective switch.	138
Figure 144.	Debug perspective	138
Figure 145.	[Run] menu	140
Figure 146.	<i>Debug</i> toolbar.	140
Figure 147.	Debug breakpoint	142
Figure 148.	Breakpoint properties	142
Figure 149.	Conditional breakpoint.	143
Figure 150.	Startup tab attach	144
Figure 151.	Reset the chip toolbar	145
Figure 152.	Restart configurations selection	145
Figure 153.	Restart configurations dialog	146
Figure 154.	Restart configurations dialog with additional command	147
Figure 155.	Select restart configuration.	147
Figure 156.	<i>Live Expressions</i>	148
Figure 157.	Live expressions number format	148
Figure 158.	Run configurations startup tab	150
Figure 159.	Cortex [®] -M executable import dialog	151
Figure 160.	STM32 Cortex [®] -M executable dialog	152

Figure 161.	STM32 Cortex [®] -M executable MCU/MPU selection	152
Figure 162.	STM32 Cortex [®] -M CPU and core	153
Figure 163.	Cortex [®] -M debug configuration for imported project	154
Figure 164.	Project explorer view with imported project	155
Figure 165.	SWV core clock	157
Figure 166.	SWV debug configuration	157
Figure 167.	SWV show view	158
Figure 168.	SWV Trace log view	159
Figure 169.	SWV [Configure Trace] toolbar button	159
Figure 170.	SWV settings dialog	159
Figure 171.	SWV [Start/Stop Trace] toolbar button	161
Figure 172.	SWV Trace Log PC sampling	161
Figure 173.	[Remove all collected SWV data] toolbar button	162
Figure 174.	SWV views selectable from the menu	162
Figure 175.	SWV views common toolbar	162
Figure 176.	SWV graph views extra toolbar	163
Figure 177.	SWV Trace Log PC sampling and exceptions	163
Figure 178.	SWV Exception Trace Log – Data tab	164
Figure 179.	SWV Exception Trace Log – Statistics tab	164
Figure 180.	SWV Data Trace configuration	166
Figure 181.	SWV Data Trace	166
Figure 182.	SWV Data Trace Timeline Graph	167
Figure 183.	SWV settings	168
Figure 184.	SWV ITM Data Console	169
Figure 185.	SWV ITM port configuration	169
Figure 186.	SWV PC sampling enable	170
Figure 187.	SWV Statistical Profiling	171
Figure 188.	SWV Preferences	172
Figure 189.	Open the SFRs view using the [Quick Access] field	174
Figure 190.	SFRs view	175
Figure 191.	SFRs view toolbar buttons	176
Figure 192.	SFRs CMSIS-SVD Settings	176
Figure 193.	ThreadX views selectable from the menu	178
Figure 194.	ThreadX Thread List view (default)	179
Figure 195.	ThreadX Thread List view (Stack Usage enabled)	179
Figure 196.	ThreadX Semaphores view	181
Figure 197.	ThreadX Mutexes view	181
Figure 198.	ThreadX Message Queues view	182
Figure 199.	ThreadX Event Flags view	183
Figure 200.	ThreadX Timers view	183
Figure 201.	ThreadX Memory Block Pools view	184
Figure 202.	ThreadX Memory Byte Pools view	185
Figure 203.	FreeRTOS [™] -related views selectable from the menu	187
Figure 204.	FreeRTOS Task List (default)	187
Figure 205.	FreeRTOS [™] Toggle Stack Checking	188
Figure 206.	FreeRTOS Task List (Min Free Stack enabled)	188
Figure 207.	FreeRTOS Task List with ConfigRECORD_STACK_HIGH_ADDRESS enabled	188
Figure 208.	FreeRTOS Timers	189
Figure 209.	FreeRTOS Semaphores	190
Figure 210.	FreeRTOS Queues	191
Figure 211.	RTOS-kernel-aware debug	192
Figure 212.	RTOS-kernel-awareness debug configuration	193
Figure 213.	ThreadX-kernel-awareness debug configuration	193
Figure 214.	ThreadX port configuration	194

Figure 215.	FreeRTOS™ port configuration	194
Figure 216.	Open the <i>Fault Analyzer</i> view	195
Figure 217.	<i>Fault Analyzer</i> view	197
Figure 218.	<i>Fault Analyzer</i> toolbar	198
Figure 219.	Fault analyzer open editor on fault	198
Figure 220.	Fault Analyzer open disassembly on fault.	198
Figure 221.	Build analyzer.	199
Figure 222.	<i>Memory Regions</i> tab	200
Figure 223.	<i>Memory Details</i> tab	201
Figure 224.	<i>Memory Details</i> sorted by size	203
Figure 225.	<i>Memory Details</i> search and filter	203
Figure 226.	Sum of sizes	204
Figure 227.	Show byte count.	204
Figure 228.	Show hex count	205
Figure 229.	Copy and paste	205
Figure 230.	<i>Static Stack Analyzer List</i> tab	207
Figure 231.	<i>Static Stack Analyzer Call Graph</i> tab	208
Figure 232.	Open the <i>Static Stack Analyzer</i> view	208
Figure 233.	Enable generate per function stack usage information	209
Figure 234.	<i>Static Stack Analyzer List</i> tab	210
Figure 235.	<i>Static Stack Analyzer Call Graph</i> tab	211
Figure 236.	Function symbols in <i>Static Stack Analyzer</i>	212
Figure 237.	<i>Static Stack Analyzer List</i> tab using search	213
Figure 238.	<i>Static Stack Analyzer Call Graph</i> using search	213
Figure 239.	Copy and paste	214
Figure 240.	STM32CubeIDE available updates	215
Figure 241.	STM32CubeIDE update details	216
Figure 242.	STM32CubeIDE update review licenses	216
Figure 243.	Eclipse Marketplace menu	217
Figure 244.	Eclipse marketplace	218
Figure 245.	Install new software menu	219
Figure 246.	Install new software.	219
Figure 247.	Install new software from computer	220
Figure 248.	About STM32CubeIDE	220
Figure 249.	Installation details	221
Figure 250.	Older workspace version warning	221

IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, please refer to www.st.com/trademarks. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2021 STMicroelectronics – All rights reserved