Arm® Keil® Studio Visual Studio Code Extensions

User Guide

Non-Confidential

Issue 15
108029_0000_15_en

Arm® Keil® Studio Visual Studio Code Extensions
User Guide

Release information

Document history

Issue	Date	Confidentiality	Change
0000-15	19 June 2024	Non-Confidential	Updates
0000-14	22 May 2024	Non-Confidential	Updates
0000-13	23 April 2024	Non-Confidential	Updates
0000-12	8 April 2024	Non-Confidential	Updates
0000-11	21 March 2024	Non-Confidential	Updates
0000-10	29 February 2024	Non-Confidential	Updates
0000-09	31 January 2024	Non-Confidential	Updates
0000-08	20 December 2023	Non-Confidential	Updates
0000-07	5 December 2023	Non-Confidential	Updates
0000-06	14 November 2023	Non-Confidential	Updates
0000-05	19 October 2023	Non-Confidential	Updates
0000-04	3 October 2023	Non-Confidential	Updates
0000-03	6 September 2023	Non-Confidential	Updates
0000-02	20 July 2023	Non-Confidential	Updates
0000-01	13 July 2023	Non-Confidential	First release

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction

1.1 Conventions

The following subsections describe conventions used in Arm documents.

Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Caution

We recommend the following. If you do not follow these recommendations your system might not work.

Warning

Your system requires the following. If you do not follow these requirements your system will not work.

Danger

You are at risk of causing permanent damage to your system or your equipment, or harming yourself.

Note

This information is important and needs your attention.

Tip

A useful tip that might make it easier, better or faster to perform a task.

Remember

A reminder of something important that relates to the information you are reading.

1.2 Other information

See the Arm website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support.
- Arm® Glossary.

2. Extension pack and extensions

The Arm® Keil® Studio Visual Studio Code extension pack, Arm Keil Studio Pack, provides a comprehensive software development environment for embedded systems and IoT software development on Arm-based microcontroller (MCU) devices. Use the Keil Studio extensions contained in the pack to manage your CMSIS solutions (csolution projects), and to create, build, test, and debug embedded applications on your chosen hardware.

The Keil Studio extensions are part of the Arm Keil Microcontroller Development Kit (MDK). MDK is a collection of software tools for developing embedded applications based on Arm Cortex®-M and Ethos™-U processors. MDK gives you the flexibility to work with a command-line interface (CLI) or an integrated development environment (IDE), or by deploying the tools into a continuous integration workflow.

2.1 Arm Keil Studio Pack

The Arm® Keil® Studio Pack is collection of Visual Studio Code extensions. The pack provides the software development environment for embedded systems and IoT software development on Arm-based microcontroller (MCU) devices.

The Keil Studio Pack contains the following extensions:

- Arm CMSIS Solution (Identifier: `arm.cmsis-csolution`): This extension provides support for working with CMSIS solutions (csolution projects).
- Arm Device Manager (Identifier: `arm.device-manager`): This extension allows you to manage hardware connections for Arm Cortex®-M based microcontrollers, development boards, and debug probes.
- Arm Debugger (Identifier: `arm.arm-debugger`): This extension provides access to the Arm Debugger engine for Visual Studio Code by implementing the Microsoft Debug Adapter Protocol (DAP). Arm Debugger supports connections to physical targets, either through external debug probes such as Arm's ULINK™ family of debug probes, or through on-board low-cost debugging such as ST-Link or CMSIS-DAP based debug probes.
- Arm Environment Manager (Identifier: `arm.environment-manager`): This extension installs the tools that you specify in a manifest file in your environment. For example, you can install Arm Compiler for Embedded, CMSIS-Toolbox, CMake, and Ninja to work with CMSIS solutions.
- Arm Virtual Hardware (Identifier: `arm.virtual-hardware`): This extension allows you to manage Arm Virtual Hardware and run embedded applications on virtual targets. An authentication token is required to access the service. For more details, read the AVH solutions overview.

Arm Debugger is also an extension pack that contains the following extensions:

- Arm Environment Manager (Identifier: `arm.environment-manager`): The Environment Manager is available in the Arm Debugger extension pack if you want to install Arm Debugger and other related extensions without using the Keil Studio Pack.

- Memory Inspector (Identifier: `eclipse-cdt.memory-inspector`): This extension allows you to analyze and monitor the memory contents in an embedded system. It helps you to identify and debug memory-related issues during the development phase of your project.
- Peripheral Inspector (Identifier: `eclipse-cdt.peripheral-inspector`): This extension uses System View Description (SVD) files to display peripheral details. SVD files provide a standardized way to describe the memory-mapped registers and peripherals of a microcontroller or a System-on-Chip (SoC).

Note

- The Arm Virtual Hardware extension is in development, and is not described in this guide.
- The Memory Inspector and the Peripheral Inspector are third-party open-source extensions and are not described in this guide.

You can also install and use the extensions contained in the pack individually. However, Arm recommends installing the Keil Studio Pack in Visual Studio Code Desktop to quickly set up your environment and start working with an example. See the pack README file for more details.

3. Intended use cases for the extensions

The intended use cases for the extensions are as follows:

- **Embedded and IoT software development using CMSIS-Packs and solutions (csolution projects)**: The Common Microcontroller Software Interface Standard (CMSIS) provides driver, peripheral, and middleware support for thousands of MCUs and hundreds of development boards. Using the csolution project format, you can incorporate any CMSIS-Pack based device, board, and software component into your application. For more information about supported hardware for CMSIS projects, go to the Boards and Devices pages on keil.arm.com. For information about CMSIS-Packs, go to open-cmsis-pack.org.
- **Enhancement of a pre-existing Visual Studio Code embedded software development workflow**: You can adapt USB device management and embedded debugging to other project formats (for example CMake) and toolchains without additional overhead. This use case requires familiarity with Visual Studio Code to configure tasks. See the individual extensions for more details.

4. Get started with an example project

Set up your environment and start working with an example.

Note

This section describes working with example solution or µVision projects that you can get from keil.arm.com. If you open a µVision project, Visual Studio Code converts it automatically to csolution format and installs any missing packs. Visual Studio Code also initialises Git and configures a vcpkg instance for the current workspace. Any projects that have the `AC5` compatibility label only use Arm Compiler 5, which does not support automatic conversion. As a workaround, you can update Arm Compiler 5 projects to Arm Compiler 6 in Keil µVision, then convert the projects to csolutions in Visual Studio Code. See Download a Keil µVision example for more information.

You can also create solutions from scratch, or convert your existing µVision projects to solutions. For more information, see Create a solution and Convert a Keil µVision project to a solution.

We recommend installing the Keil Studio Pack in Visual Studio Code Desktop as explained in the README file. The pack installs all the Keil® Studio extensions, as well as the Red Hat YAML and clangd extensions.

Note

If you do not want to use clangd, you can install the Microsoft C/C++ and Microsoft C/C++ Themes extensions instead to enable IntelliSense.

Then:

- Run the setup process using an example solution project from keil.arm.com (recommended).
- Download a Keil µVision `*.uvprojx` project from keil.arm.com and convert it to a solution (alternative).

The examples available on keil.arm.com are shipped with a Microsoft vcpkg manifest file (`vcpkg-configuration.json`). The Environment Manager extension uses the manifest file to acquire and activate the tools that you need to work with solutions using Microsoft vcpkg.

Each example also comes with a `tasks.json` file and a `launch.json` file to build, run, and debug the project.

The tools installed by default are:

- Arm® Compiler for Embedded.
- CMSIS-Toolbox.
- CMake and Ninja.

Finalize the setup of your development environment:

- If you are working behind an HTTP proxy, see Configure an HTTP proxy.
- For more information on the clangd extension and how it adds smart features to your editor, see clangd.

When you are ready:

- Build the example project.
- Explore what you can do with the CMSIS Solution extension:
  - Choose a context for your solution
  - Look at the Solution outline
  - Install CMSIS-Packs and select software components from packs
- Connect your board and run the example on the board.
- Start a debug session.
- Check the serial output.

4.1 Import a solution example

Import a solution example in Visual Studio Code, or download a zip file that contains the solution.

Procedure

1. Go to keil.arm.com.
2. Click the **Hardware** menu and select **Boards**.
3. Search for your board and select it in the **Suggested Boards** list.
4. Find a project in the **Projects** tab.
   The `Keil Studio` compatibility label indicates that the example is compatible with Keil® Studio Cloud and the Keil Studio Visual Studio Code extensions.
5. Move your cursor over **Get Project**, and then click **Open in Keil Studio for VS Code** to import the solution example.
   Alternatively, you can download a zip file that contains the solution with the **Download .zip** option.
6. In the “Open Visual Studio Code?” dialog box that opens at the top of your browser window, click **Open Visual Studio Code**.
7. In the “Allow ‘Arm Keil Studio Pack’ extension to open this URI?” dialog box that opens in Visual Studio Code, click **Open**.
8. Choose a folder to import the project and click **Select as Unzip Destination**.
9. In the “Would you like to open the unzipped folder, or add it to the current workspace?” dialog box, click **Open**.
10. Confirm that the Environment Manager extension can automatically activate the workspace and download the tools specified in your `vcpkg-configuration.json` file.

If there are missing CMSIS-Packs, a pop-up message displays in the bottom right-hand corner with the following message: “Solution [solution-name] requires some packs that are not installed”.

11. Click **Show Missing Packs** to open the **Problems** view.
12. Right-click the error in the **Problems** view and select **Install missing pack**. If there are several packs missing, use **Install all missing packs**.
    You must activate a license to be able to use tools such as Arm® Compiler, Arm Debugger, or Fixed Virtual Platforms in your toolchain. If you have not activated your license after installing the pack, a pop-up message displays in the bottom right-hand corner. See Activate your license to use Arm tools for more details on licensing.
13. Click **Explorer** .
    A `vcpkg-configuration.json` is available. The file records the vcpkg artifacts, such as the compiler toolchain version, that you need to work with your projects. You do not need to do anything to install the tools. Microsoft vcpkg and the Environment Manager extension take care of the setup. See Tools installation with Microsoft vcpkg.

    A `tasks.json` file and a `launch.json` file are also available in the **.vscode** folder. Visual Studio Code uses the `tasks.json` file to build and run the project, and the `launch.json` file for debugging.

4.2 Download a Keil µVision example

When you download and open a Keil® µVision® `*.uvprojx` project, Visual Studio Code automatically converts it to a solution and installs any missing packs. Note that conversion does not work with Arm® Compiler 5 projects. You can download Arm Compiler 5 projects from the website, but you cannot use them with the extensions. Only Arm Compiler 6 projects can be converted. As a workaround, you can update Arm Compiler 5 projects to Arm Compiler 6 in Keil µVision, then convert the projects to solutions in Visual Studio Code. For more help and information on converting to Arm Compiler 6, see the Migrate Arm Compiler 5 to Arm Compiler 6 application note and the Arm Compiler for Embedded Migration and Compatibility Guide.

Procedure

1. Go to keil.arm.com.
2. Connect your board over USB and click **Detect Connected Hardware** in the bottom right-hand corner.
3. Select the device firmware for your board in the dialog box that displays at the top of the window, and then click **Connect**.
4. Click the **Board** link in the pop-up message that displays in the bottom right-hand corner.

    The page for the board opens. Example projects are available in the **Projects** tab.
5. Move your cursor over the **Get Project** button for the project that you want to use and click **Download .zip** to download the Keil µVision `*.uvprojx` example.
6. Unzip the example and open the folder in Visual Studio Code.

The conversion starts immediately, and any required packs that are missing are automatically installed.

A dialog box displays. You can carry out the following tasks:

- Open the solution in a new workspace (**Open** option)
- Open the solution in a new window and new workspace (**Open project in new window** option)

If there are conversion errors, check the `uv2csolution.log` file available.

7. Confirm that the Environment Manager extension can automatically activate the workspace and download the tools specified in your `vcpkg-configuration.json` file.

You must activate a license to be able to use tools such as Arm® Compiler, Arm Debugger, or Fixed Virtual Platforms in your toolchain. If you have not activated your license after installing the pack, a pop-up message displays in the bottom right-hand corner. See Activate your license to use Arm tools for more details on licensing.

The `*.cproject.yml` and `*.csolution.yml` files are available next to the `*.uvprojx` in the **Explorer** .

A `vcpkg-configuration.json` file is available. The file records the vcpkg artifacts, such as the compiler toolchain version, that you need to work with your projects. You do not need to do anything to install the tools. Microsoft vcpkg and the Environment Manager extension take care of the setup. See Tools installation with Microsoft vcpkg.

A `tasks.json` file and a `launch.json` file are also available in the **.vscode** folder. Visual Studio Code uses the `tasks.json` file to build and run the project, and the `launch.json` file for debugging.

4.3 Finalize the setup of your development environment

To finalize the setup of your development environment:

- Configure an HTTP proxy. This step is required only if you are working behind an HTTP proxy.
- The pack installs all the Keil® Studio extensions, as well as the Red Hat YAML and clangd extensions. See clangd for more information on this extension.

4.3.1 Configure an HTTP proxy (optional)

This step is required only if you are working behind an HTTP proxy. You can configure the tools to use an HTTP proxy using the following standard environment variables:

- `HTTP_PROXY`: Set to the proxy used for HTTP requests
- `HTTPS_PROXY`: Set to the proxy used for HTTPS requests

- `NO_PROXY`: Set to include at least `localhost,127.0.0.1` to disable the proxy for internal traffic, which is required for the extension to work correctly

4.3.2 clangd

The clangd extension adds smart features such as code completion, compile errors, and go-to-definition to your editor.

Note

The clangd extension requires the clangd language server. If the server is not found on your PATH, add it with the **clangd: Download language server** command from the Command Palette. Read the clangd extension `README` file for more information.

After clangd has been installed, no extra setup is needed. The CMSIS Solution extension generates a `compile_commands.json` file for each project in a solution whenever a csolution file changes or when you change the context of a solution (**Target Type** and **Build Type** types). A `.clangd` file is kept up to date for each project in the solution. The clangd extension uses the `.clangd` file to locate the `compile_commands.json` files, to provide additional compile flags, and to enable IntelliSense. See the clangd documentation for more details.

Note

To improve IntelliSense with clangd, additional scoped compiler flags (to define certain macros) are added to both your project configuration file (`.clangd`) and your global user configuration file (`config.yaml`). See the clangd documentation for more details.

To turn off the automatic generation of the `.clangd` file and `compile_commands.json` file:

1. Open the settings:
   - On Windows or Linux, go to **File** > **Preferences** > **Settings**.
   - On macOS, go to **Code** > **Settings** > **Settings**.
2. Find the **Cmsis-csolution: Auto Generate Clangd File** and **Cmsis-csolution: Auto Generate Compile Commands** settings and clear their checkboxes.

To turn off the automatic addition of compiler flags for macro defines:

1. Open the settings:
   - On Windows or Linux, go to **File** > **Preferences** > **Settings**.
   - On macOS, go to **Code** > **Settings** > **Settings**.
2. Find the **Cmsis-csolution: Clangd Armclang Macro Query** setting and clear its checkbox.

4.4 Build the example project

Check that your example project builds. You can build your project from the **Explorer** using **Build**, from the **Solution outline**, or from the Command Palette.

Procedure

1. Build the project:
   - From the **Explorer**:
     a. Go to the **Explorer** view.
     b. Right-click the `*.csolution.yml` file and select **Build**.

        These options are also available in the right-click menu:
        - **Clean Output Directories**: cleans the output directories for the active solution
        - **Rebuild**: cleans the output directories before building the cproject
   - From the **Solution outline**:
     a. Click **CMSIS** in the Activity Bar.

        The **Solution outline** opens.

        Note: If you have previously cleared all the projects in the context for your solution, click **Open Build Context Editor** to open the **Build Context** view, where you can select projects. See Choose a context for your solution.
     b. Move your cursor over the **Solution outline**.

        Build icons are available at the solution or project level.
     c. Click **Build**.

        The **Clean Output Directories** and **Rebuild** options are also available with **More Actions** ....

        You can configure a build task in a `tasks.json` file to customise the behaviour of the build button. A `tasks.json` file is provided for all the examples available on keil.arm.com. See Configure a build task for more details.
   - From the Command Palette: **Build**, **Clean Output Directories**, and **Rebuild** can also be triggered from the Command Palette with the **CMSIS: Build**, **CMSIS: Clean Output Directories**, and **CMSIS: Rebuild** commands.

Note

If the build fails with an ENOENT error, follow the instructions in the pop-up message that displays in the bottom right-hand corner for installing CMSIS-Toolbox. See Build fails to find CMSIS-Toolbox causes an ENOENT error for more information.

2. Check the **Terminal** tab to find where the ELF file (`.axf`) was generated.

4.5 Choose a context for your solution

A context is the combination of a target type (build target) and build type (build configuration) for a particular project in your solution.

Read Set a context for your solution for more details.

4.6 Look at the Solution outline

The **Solution outline** presents the content of your solution in a tree view.

Read Use the Solution outline for more details.

4.7 Install CMSIS-Packs and select software components from packs

CMSIS-Packs contain reusable software components that you can use to quickly build projects. CMSIS-Packs are listed in the `csolution.yml` files of solutions. The CMSIS Solution extension seamlessly handles the installation of packs to your pack cache.

See CMSIS-Packs and Install CMSIS-Packs for more details.

The **Software Components** view shows all the software components selected in the active project of your solution.

Read Manage software components for more details.

4.8 Connect your board

Connect your board. See Supported hardware for more details on the development boards, MCUs, and debug probes supported by the extensions.

Procedure

1. Click **Device Manager** in the Activity Bar to open the Device Manager extension.
2. Connect your board to your computer over USB.

The board is detected and a pop-up message displays.

3. Click **OK** in the pop-up message to use the hardware.

   Your board is now ready to be used to run and debug a project.

4.9 Run the solution on your board

Run the solution project on your board.

Procedure

1. Click **CMSIS** in the Activity Bar.
   The **Solution outline** opens.
2. Move your cursor over the **Solution outline**.
   Run icons are available at the solution level or at the project level depending on what you selected in the **Build Context** view for the run configuration. See Set a context for your solution for more details.
3. Click **Run**.
4. If you are using a device with multiple cores and you did not specify a `"processorName"` in the `launch.json` file, and you do not have the CMSIS Solution extension installed, then you must select the appropriate processor for your project in the **Select a processor** drop-down list that displays at the top of the window.
   The project is run on the board.
5. Check the **Terminal** tab.

4.10 Start a debug session

Start a debug session.

Procedure

1. Click **CMSIS** in the Activity Bar.
   The **Solution outline** opens.
2. Move your cursor over the **Solution outline**.
   Debug icons are available at the solution level or at the project level depending on what you selected in the **Build Context** view for the debug configuration. See Set a context for your solution for more details.
3. Click **Debug**.
4. If you are using a device with multiple cores and you did not specify a `"processorName"` in the `launch.json` file, and you do not have the CMSIS Solution extension installed, then you must select the appropriate processor for your project in the **Select a processor** drop-down list that displays at the top of the window.

The **Run and Debug** view displays and the debug session starts. The debugger stops at the `main()` function of your project.

5. Check the **Debug Console** tab to see the debugging output.

Next steps

Look at the Visual Studio Code documentation to learn more about the debugging features available in Visual Studio Code.

5. Arm Environment Manager extension

The Arm® Environment Manager extension allows you to manage environment artifacts, such as a compiler toolchain, using Microsoft vcpkg. The extension uses a vcpkg manifest file to acquire and activate the artifacts that you need to set up your development environment.

The artifacts for your project are stored in the `vcpkg-configuration.json` file in the project source code. This means that the same tools are available to everyone using the project.

Information about vcpkg is available at vcpkg.io and at Microsoft Learn.

Note

If you do not want to use the Environment Manager extension and vcpkg, you can install the artifacts for your project by manually downloading and installing the CMSIS-Toolbox and other required tools such as CMake and Ninja. For more information, see the CMSIS-Toolbox installation instructions in the Open-CMSIS-Pack documentation. See also Add CMSIS-Toolbox to the system PATH for information on how to specify the path of your CMSIS-Toolbox. For other specific cases, see Specific installation use cases.

The Environment Manager extension also includes features to help you license your tools. See Activate your license to use Arm tools for more details.

A full list of commands and settings is available for the **Arm Environment Manager** extension. To view the list, click **Extensions** in the Visual Studio Code Activity Bar. Click **Arm Environment Manager** in the list of extensions, and then click **Features** (Windows) or **Feature Contributions** (macOS).

5.1 Tools installation with Microsoft vcpkg

Microsoft vcpkg works in combination with the Environment Manager extension installed with the pack for the setup of your environment.

Each official Arm example project is shipped with a manifest file (`vcpkg-configuration.json`). The manifest file records the vcpkg artifacts that you need to work with your projects. An artifact is a set of packages required for a working development environment. Examples of relevant packages include compilers, linkers, debuggers, build systems, and platform SDKs.

For more information on vcpkg, see the official Microsoft vcpkg documentation. See also the Microsoft vcpkg-tool repository for more details on artifacts.

Note

If you are using Windows, you must enable long path support when using Keil Studio and the Environment Manager extension. If long paths are not enabled, the downloading and installation of vcpkg artifacts can fail.

Environment Manager detects whether long paths are enabled in the Windows registry, and displays an alert if they are not. To enable long paths in your Windows settings, follow the instructions here: Enable Long Paths in Windows 10, Version 1607, and Later.

5.2 Confirm automatic activation

If you open a new workspace, duplicate an existing workspace, or open an example project from keil.arm.com, the Environment Manager extension automatically activates the workspace and downloads the tools specified in your `vcpkg-configuration.json` file. A dialog box opens, allowing you to confirm the activation. You can open the `vcpkg-configuration.json` file to see what will be installed.

You can also change the automatic activation at any time from the settings.

5.3 Check the tools installed with Microsoft vcpkg

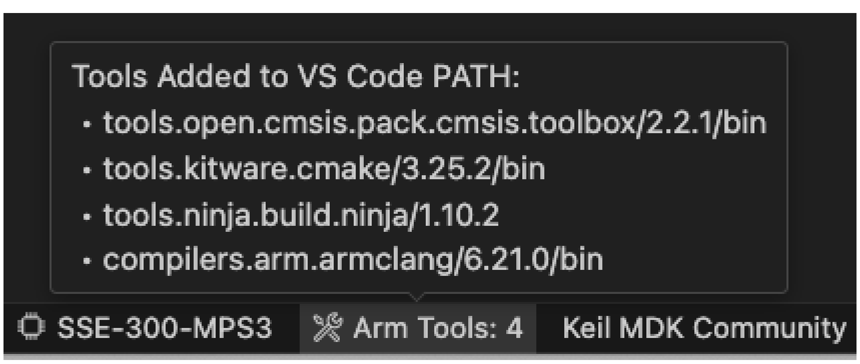
The `vcpkg-configuration.json` manifest file instructs Microsoft vcpkg to install the artifacts. For example:

```
"requires": {
      "arm:tools/open-cmsis-pack/cmsis-toolbox": "2.2.1",
  "arm:compilers/arm/armclang": "6.21.0",
      "microsoft:tools/kitware/cmake": "3.25.2",
      "microsoft:tools/ninja-build/ninja": "1.10.2"
}
```

The artifacts installed with this example manifest file are cmsis-toolbox, armclang (Arm Compiler for Embedded), cmake, and ninja.

Move your mouse over **Arm Tools** in the status bar to see what tools are installed.

Figure 5-1: Arm Tools



You can also click **Arm Tools** in the status bar and select the **View Log** option. This opens the **Output** tab (**Arm Tools** category).

By default, Microsoft vcpkg installs the tools in your `user` folder.

- On Windows: `c:\Users\<user>\.vcpkg\artifacts`
- On Linux: `/home/<user>/.vcpkg/artifacts`
- On macOS: `/Users/<user>/.vcpkg/artifacts`

After Microsoft vcpkg has been activated for a project, any **Terminal** that you open in Visual Studio Code has all the tools added to the PATH by default (Arm Compiler for Embedded, CMSIS-Toolbox, CMake, and Ninja). This process allows you to run the different CMSIS-Toolbox tools such as `cpackget`, `cbuildgen`, `cbuild`, or `csolution`.

5.4 Modify the manifest file manually

To add or change tools in your environment, modify the artifacts contained in the manifest file of your project.

The artifacts provided by Arm are listed on the Arm tools available in vcpkg page on keil.arm.com. The page lists only the versions that are available on the Artifactory repository manager.

Copy the code snippets for the artifacts that you want to install and paste them in the `vcpkg-configuration.json` manifest file of your project in the `"requires":` section, then save the file. The newly added or updated artifacts are automatically downloaded and activated.

See also Use the Arm Tools visual editor as an alternative to editing the manifest file manually.

5.5 Use the Arm Tools visual editor

As an alternative to editing the `vcpkg-configuration.json` manifest file directly, you can use the **Arm Tools** visual editor to add or change tools in your environment.

Procedure

1. Right-click anywhere in the **Explorer** view.
2. From the menu that opens, select **Configure Arm Tools Environment**.

   The **Arm Tools** editor opens.

   You can also open the editor by clicking **Arm Tools** in the status bar and selecting the **Configure Arm Tools Environment** option in the drop-down list that displays at the top of the window.
3. Use the drop-down lists to install or update the tools that you want to use in your environment.

   For example, select version `6.22.0` in the **Arm Compiler for Embedded** drop-down list. Only the versions available on the Artifactory repository manager are exposed in the user interface.
4. If **Auto Save** is not enabled (**File** > **Auto Save**), save your settings.

The newly added or updated tools are automatically downloaded and activated. You can view details of what has been installed in the **Output** tab (**View** > **Output**).

5.6 vcpkg activation options

Several options are available to add a `vcpkg-configuration.json` file to your workspace, to activate, deactivate, or reactivate your environment with Microsoft vcpkg, and to update the vcpkg registries. If you are using an example from keil.arm.com or if you created a solution from scratch from the **Create New Solution** view, your environment is activated by default.

Procedure

1. From the **Explorer**, open your workspace.
2. Right-click the `vcpkg-configuration.json` file.
3. Depending on the activation status of your environment and the **Environment Manager** settings selected, the following options are available:
   - **Configure Arm Tools Environment**: Open the visual editor. Use this option to open the visual editor and select the tools you want to install. See Use the Arm Tools visual editor.
   - **Activate Environment**: Activate the environment. This option is available only if you previously deactivated your environment or if you modified the **Activate On Config Creation** or **Activate On Workspace Open** settings for the **Environment Manager**. Tools are available on the PATH.
   - **Deactivate Environment**: Deactivate the active environment. Tools are also removed from the PATH.
   - **Reactivate Environment**: Deactivate and activate the environment (for example, if you have changed your vcpkg configuration).
   - **Update Tool Registry**: Check for fresh artifacts published in the registries.

   The same options are available when you click **Arm Tools** in the status bar.

   The **View Log** option in the drop-down list opens the **Output** tab to allow you to check what tools have been installed. See Check the tools installed with Microsoft vcpkg.

Note

If your project does not contain a `vcpkg-configuration.json` file, or if you have deactivated the active environment, click **Arm Tools** in the status bar, then select **Add Arm tools Configuration To Workspace** to open the visual editor and select tools.

5.7 Use vcpkg from the command line

You can also use vcpkg from the command line to create reproducible tool installations.

The Arm Developer Learning Paths have an example scenario that shows you how to install and initialize vcpkg, and how to create and use the configuration file. See Install tools on the command line using vcpkg.

5.8 Specific installation use cases

This section describes the following use cases:

- Installing a specific version of Arm® Compiler for Embedded
- Using Arm Compiler for Embedded FuSa
- Using a pre-installed toolchain instead of vcpkg
- Working on a machine with no internet access

5.8.1 Switch to a specific Arm Compiler for Embedded version

To switch to a specific Arm® Compiler for Embedded version, use the Arm Tools visual editor.

Only the versions available on the Artifactory repository manager are exposed in the user interface. You can select versions 6.18.0 and above.

Note

Versions of Arm Compiler for Embedded older than 6.18.0 do not support user-based licensing (UBL). As a consequence, these versions do not work with MDK v6. For more details on user-based licensing support and backwards compatibility, see the User-based licensing User Guide.

To switch to a specific Arm Compiler for Embedded version:

1. Select the version of Arm Compiler for Embedded that you need from the Arm Tools visual editor.

   For example, select version `6.18.0` in the **Arm Compiler for Embedded** drop-down list.
2. Specify the version that you installed in either the `*.csolution.yml` file or the `*.cproject.yml` file of your project.

   For example, for version 6.18.0, add `compiler: AC6@6.18.0` in the `*.csolution.yml` file as explained in the Open-CMSIS-Pack documentation.
3. Restart Visual Studio Code.
4. Right-click the `*.csolution.yml` file and select **Rebuild** to rebuild the project.

5.8.2 Use Arm Compiler for Embedded FuSa

If you have functional safety (FuSa) requirements for your projects, you can use Arm® Compiler for Embedded FuSa.

Arm Compiler for Embedded FuSa is available only on the Product Download Hub (PDH). You need an MDK-Professional license to use it.

Note

Versions of Arm Compiler for Embedded FuSa older than 6.16.2 do not support user-based licensing (UBL). As a consequence, these versions do not work with MDK v6. For more details on user-based licensing support and backwards compatibility, see the User-based licensing User Guide.

To install Arm Compiler for Embedded FuSa version 6.16.2:

1. Download the 6.16.2 version from the Product Download Hub (PDH) and manually install Arm Compiler for Embedded FuSa on your machine.

   Note

   You need an Arm account to access PDH. To download Arm Compiler for Embedded FuSa, your account must be connected with a valid MDK-Professional license.

   ---

2. Specify the version that you installed in either the `*.csolution.yml` file or the `*.cproject.yml` file of your project.

   Add `compiler: AC6@6.16.2` in the `*.csolution.yml` file as explained in the Open-CMSIS-Pack documentation.

3. Add the Arm Compiler for Embedded FuSa toolchain in the global environment variables for the operating system that you are using.

   Set the `AC6_TOOLCHAIN_6_16_2` environment variable to point to the toolchain binaries. See the Open-CMSIS-Pack documentation.

4. Restart Visual Studio Code.
5. Build the project. See Build the example project.

5.8.3 Use a pre-installed toolchain

To use a toolchain that was already installed before installing the Keil Studio Pack, you must deactivate vcpkg to avoid conflicts with your personal setup. If your project does not include a `vcpkg-configuration.json` file, then you do not need to do anything.

Procedure

1. Deactivate vcpkg:
   a. Open the settings:
      - On Windows or Linux, go to **File** > **Preferences** > **Settings**.

    - On macOS, go to **Code** > **Settings** > **Settings**.

   b. Find the **Activate on Workspace Open** setting and clear its checkbox.

2. Make sure that the toolchain is installed correctly and that you have added the toolchain in the global environment variables for the operating system that you are using.
   For example, for Arm Compiler for Embedded version 6.18.0, set the `AC6_TOOLCHAIN_6_18_0` environment variable to point to the toolchain binaries. See the Open-CMSIS-Pack documentation.
3. Restart Visual Studio Code.
4. Use `cbuild list toolchains -v` to check the variable path.

5.8.4 Use the Keil Studio extensions on an air-gapped machine

You can use the **Activate Environment** option (see vcpkg activation options) or run the `vcpkg activate` command from the **Terminal** on a connected machine, and then transfer the vcpkg root directory to the air-gapped machine. This transfers all the required tools to the air-gapped machine. You can then use the air-gapped machine without an internet connection.

6. Arm CMSIS Solution extension

The Arm CMSIS Solution extension provides support for working with CMSIS solutions (csolution projects). The extension manages the information needed to create your solutions.

With the CMSIS Solution extension, you can carry out the following tasks:

- Set a context for your solution
- Use the Solution outline
- Install CMSIS-Packs
- Manage software components
- Use the Configuration Wizard to customize startup code and other configuration files

You can also:

- Create a solution from scratch
- Convert a Keil µVision project to a solution
- Configure a build task
- Initialize your solution
- Use the CMSIS csolution API

Note

For information on working with existing example projects from keil.arm.com instead of creating new projects from scratch, see Get started with an example project.

A full list of commands and settings is available for the **Arm CMSIS Solution** extension. To view the list, click **Extensions** in the Visual Studio Code Activity Bar. Click **Arm CMSIS Solution** in the list of extensions, and then click **Features** (Windows) or **Feature Contributions** (macOS).

6.1 CMSIS solutions

A solution is a container used to organize related projects that are part of a larger application and that can be built separately. See Project Setup for Related Projects for a solution example.

Solutions are defined in YAML format using `*.csolution.yml` files. The `*.csolution.yml` file defines the complete scope of an application and the build order of the projects that the application contains. Individual projects are defined using `*.cproject.yml` files. The `*.cproject.yml` file defines the content of an independent build. Each project corresponds to one binary file (build artifact).

You can edit the `*.csolution.yml` and `*.cproject.yml` files of a solution manually. The Keil Studio Pack includes the Red Hat YAML extension, and the CMSIS Solution extension uses YAML schemas to make the editing of these files easier. See the vscode-yaml repository for more information on the extension.

See the Build Overview of the CMSIS-Toolbox documentation and the Project Examples to understand how solutions and projects are structured. For more information on csolution project files, see CMSIS Solution Project File Format.

6.2 Set a context for your solution

Look at your solution contexts. A context is the combination of a target type and build type for a particular project in your solution.

Procedure

1. Click **CMSIS** in the Activity Bar to open the **CMSIS** view.
2. Choose one of the following options:
    - Click in the **Solution outline** header
    - Select **CMSIS: Manage Solution Settings** from the Command Palette
    - Click in the status bar.

    The **Build Context** view opens.
3. Look at the available contexts for the solution. You can change the target type, the projects included in the build, and the build type. You can also change the run configuration and the debug configuration, or add new configurations.
    - **Active Target**: Select a **Target Type** to specify the hardware to use to build the solution. Some examples are also compatible with Arm® Virtual Hardware (AVH) targets, in which case more options are available. For more details, read the AVH solutions overview.

      Click **Edit targets in the csolution.yml** to specify your target types by editing the YAML file directly.
    - **Active Projects**:
        - **Project Name**: The project or projects included in the build. If you have multiple projects in your solution, you can select the projects to include here.
        - **Build Type**: The build configuration. A build configuration allows you to configure each target type towards specific testing. You can set different build types for different projects in your solution. You can create your own build types as required by your application. Two commonly used examples are **Debug** for a full debug build of the software for interactive debugging, or **Release** for the final code deployment to the systems.

      Click **Edit cproject.yml** next to a project to open the `<project-name>.cproject.yml` file. YAML syntax support helps you with editing.

> **Note**
>
> The projects and build types you can select are defined by contexts for a particular target. Some options might be unavailable if they have been excluded for the target selected. To learn more about contexts and how to modify them, see the Context and Conditional build information in the CMSIS-Toolbox documentation. For example, you can use `for-context` and `not-for-context` to include or exclude target types at the `project:` level in the `*.csolution.yml` file.

- **Run and Debug**:

  Run Configuration and **Debug Configuration**: Choose a run configuration and a debug configuration to use for your solution. You can also select different run and debug configurations for each project included in the solution.

  You can also:

  - Move your mouse over an entry in the list and click the pen icon to edit an existing configuration with the visual editor.
  - Click **+ Add new** to add a new configuration, then:
    - For a run configuration, select the `arm-debugger.flash: Flash Device` task in the drop-down list that displays at the top of the window if you are using the Arm Debugger extension.
    - For a debug configuration, select `Arm Debugger: Attach`, `Arm Debugger: Launch`, or `Arm Debugger: Launch FVP` in the drop-down list that displays in the `launch.json` file if you are using the Arm Debugger extension.

  See Use the Run and Debug Configuration visual editor for your run configuration and Use the Run and Debug Configuration visual editor for your debug configuration for more details.

4. Go to the **Problems** tab and check for errors.
5. Open the `main.c` file and check the IntelliSense features available. Read the Visual Studio Code documentation on IntelliSense to find out about the different features.

6.3 Use the Solution outline

The **Solution outline** presents the content of your solution in a tree view.

Click **CMSIS** in the Activity Bar to open the **CMSIS** view. The **Solution outline** displays on the left.

> **Note**
>
> If you have previously cleared all the projects in the context for your solution, click **Open Build Context Editor** to open the **Build Context** view, where you can select projects. See Choose a context for your solution.

The **Solution outline** shows the projects (cprojects) included in the solution that are selected in the **Build Context** view. Each cproject file contains configuration settings, source code files, build settings, and other project-specific information. The extension uses these settings and files to manage and build a software project for a board or device.

You can have the following details for a project:

- `Groups`: Groups are a way to structure code files into logical blocks. You can add files to groups.

  Move your cursor over a group and click , then select one of these options:

  - **Add New File**: Create a file and add it to the group
  - **Add Existing File**: Select an existing file and add it to the group
  - **Add From Component Code Template**: Select a component code template in the drop-down list and add to the group. A code template is a predefined file included with the software components for your project to help you start developing your project.

  The files you add are listed in the `*.cproject.yml` file of the solution under the `groups` key.
- `Components`: All the software components selected for the project. Components are sorted by component class (Cclass). Code files, user code templates, and APIs from selected components display under their parent components. Click the files, templates, or APIs to open them in the editor.
- `Layers`: The clayer file, `*.clayer.yml`, defines the software layers for the project. A software layer is a set of source files, preconfigured software components, and configuration files. The clayer file can be used by multiple projects. The software components used by each layer in the project appear in the tree view.

The **Solution outline** label displays the name of your active solution. When you move your cursor over the label, you can choose one of the following actions:

- **Build**: Click to build all the projects included in the active solution. You can also build each project individually.
- **Run**: Click to run the solution on your hardware. **Run** icons are also available at the project level depending on what you selected in the **Build Context** view for the run configuration.
- **Debug**: Click to debug the solution. **Debug** icons are also available at the project level depending on what you selected in the **Build Context** view for the debug configuration.
- **Open Csolution File**: Open the main `csolution.yml` file. When you move your cursor over a project or a layer, an **Open File** option is also available.
- **Manage Solution Settings**: Click to set a context for your solution.
- **Collapse All**: Click to close all the entries in the outline.
- **More Actions** :
  - **Clean Output Directories**: Clean the output directories for the active solution
  - **Rebuild**: Clean the output directories before building the projects
  - **Convert uvproj to csolution**: Convert an existing µVision project to a solution

- **New Solution**: Create a solution from scratch
- **Open Solution**: Select the active solution. If you have several solutions in your workspace, this option allows you to switch from one solution to another. The same option is available from the **Explorer** when you right-click the `csolution.yml` file. Select a solution in the drop-down list that displays at the top of the window.

The **Solution outline** displays the selected build type and target type next to each project. You can check which software components are selected for each project. Click to open the **Software Components** view. See Manage software components for more details.

Press **Ctrl+F** (Windows) or **Cmd+F** (macOS) to look for an element in the **Solution outline**.

The `*.csolution.yml`, `*.cproject.yml`, and `*.clayer.yml` file formats are described in the Open-CMSIS-Pack documentation.

6.4 CMSIS-Packs

CMSIS-Packs offer you a quick and easy way to create, build, and debug embedded software applications for Cortex®-M devices.

CMSIS-Packs are a delivery mechanism for software components, device parameters, and board support. A CMSIS-Pack is a file collection that might include:

- Source code, header files, and software libraries - for example, RTOS, DSP, and generic middleware
- Device parameters, such as the memory layout or debug settings, along with startup code and Flash programming algorithms
- Board support, such as drivers, board parameters, and descriptions for debug connections
- Documentation and source code templates
- Example projects that show you how to assemble components into complete working systems

CMSIS-Packs are developed by various silicon and software vendors, covering thousands of different boards and devices. You can also use them to enable life-cycle management of in-house software components.

See the Open-CMSIS-Pack documentation for more details.

Discover new CMSIS-Packs on keil.arm.com/packs. Snippets that you can copy to add a pack to your `csolution.yml` file and to install packs with `cpackget add` are available for each pack.

6.5 Install CMSIS-Packs

If you started from an example available on keil.arm.com, then the CMSIS-Packs you need for the example are already listed in the `csolution.yml` file under the `packs` key. The CMSIS Solution extension scans your pack cache and offers to install any missing packs. See Install missing CMSIS-Packs for more details.

If you need to add CMSIS-Packs in your example solution, or if you are creating a solution from scratch, then you can explore the available CMSIS-Packs on keil.arm.com. See Explore the available CMSIS-Packs for more details.

See also Support for packs to understand the difference between public and private packs and how you can manage packs from the command line.

6.5.1 Install missing CMSIS-Packs

Install the missing CMSIS-Packs for your solution.

Procedure

1. Open the `*.csolution.yml` file for your solution from the **Explorer** view.
   The required packs are listed under the `packs` key of the `csolution.yml` file. If one or more CMSIS-Packs are missing, errors display in the **Problems** view and a pop-up message displays in the bottom right-hand corner with the message “Solution [solution-name] requires some packs that are not installed”.
2. Click **Install**.
   Alternatively, right-click the error in the **Problems** view and select **Install missing pack**. If there are several packs missing, use **Install all missing packs**.

   You can also install missing packs with the **CMSIS: Install required packs for active solution** command from the Command Palette.

6.5.2 Explore the available CMSIS-Packs

Explore the available CMSIS-Packs on keil.arm.com and install them.

Procedure

1. Go to the CMSIS-Packs page on keil.arm.com.
2. Search for a pack and select it in the **Results** list. For example, type `wolfSSL`.
3. Copy the `packs` snippet and update the `packs` key of your `csolution.yml` file in Visual Studio Code.

Figure 6-1: wolfSSL example

4. Install the pack by clicking **Install** in the pop-up message that displays in the bottom right-hand corner.

6.6 Manage software components

The **Software Components** view shows all the software components selected in the active project of a solution.

From this view you can see all the component details, called attributes in the Open-CMSIS-Pack documentation.

You can also carry out the following tasks:

- Modify the software components to include in the project, and manage the dependencies between components for each target type defined in your solution, or for all the target types at once
- Build the solution using different combinations of pack and component versions, and different versions of a toolchain

6.6.1 Open the Software Components view

Describes how to open the **Software Components** view.

Procedure

1. Click **CMSIS** in the Activity Bar to open the **CMSIS** view.

> **Note**
> If you have previously cleared all the projects in the context for your solution, click **Open Build Context Editor** to open the **Build Context** view, where you can select projects. See Choose a context for your solution.

2. Move your cursor over the **Solution outline**, and then click **Manage software components** at the project level.

Results

The **Software Components** view opens.

The default view displays the components available from the packs listed in your solution (**Software packs: Solution** drop-down list and **All** toggle button).

Note

If the view does not display any components, click **Install Missing Packs** to resolve the issue.

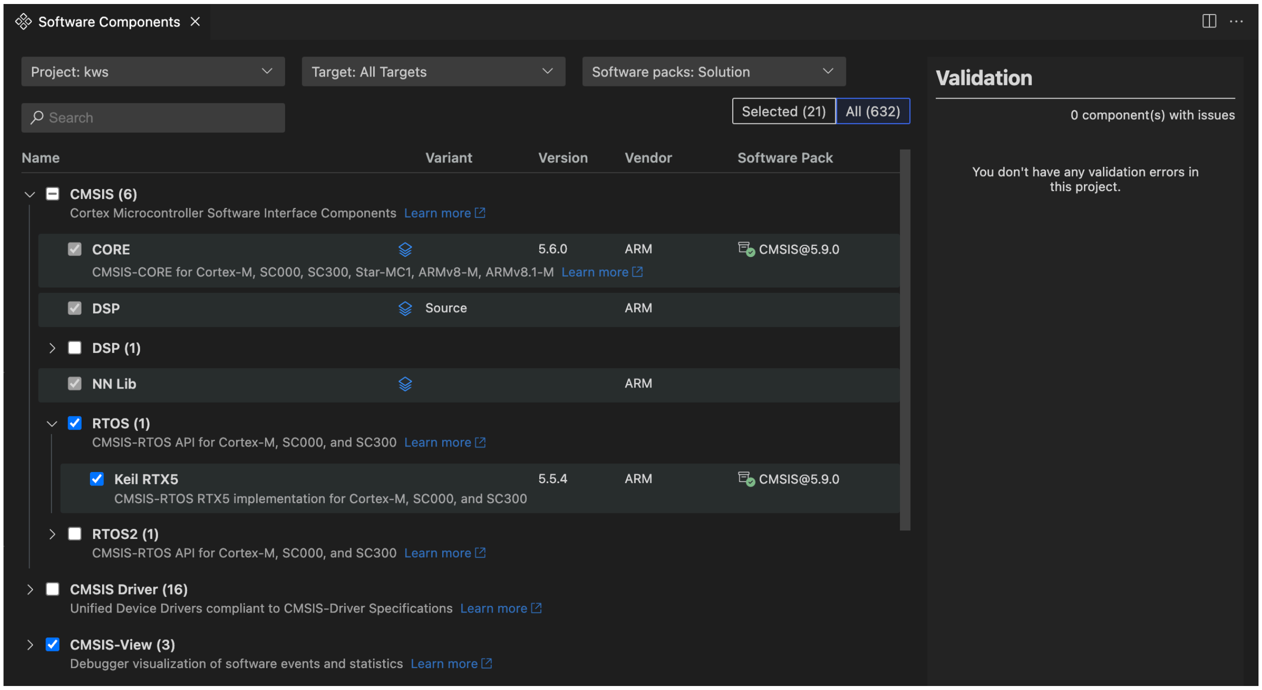
You can use the **Search** field to search the list of components.

With the **Project** drop-down list, select the project for which you want to modify software components.

With the **Target** drop-down list, select **All Targets** or a specific target type to modify software components for all the target types in your solution at once, or for a specific target only.

With the **Software packs** drop-down list, you can filter on the components available from the packs listed in your solution, or display the components from all the packs available in the CMSIS ecosystem.

Figure 6-2: The 'Software Components' view showing all the components available from the packs listed in a solution



The CMSIS-Pack specification states that each software component should have the following attributes:

- Component class (Cclass): A top-level component name (for example, **CMSIS**)
- Component group (Cgroup): A component group name (for example, **CORE** for the **CMSIS** component class)
- Component version (Cversion): The version number of the software component

Optionally, a software component might have these additional attributes:

- Component subgroup (Csub): A component subgroup that is used when multiple compatible implementations of a component are available (for example, **Keil RTX5** under **CMSIS > RTOS2**)
- Component variant (Cvariant): A variant of the software component that is typically used when the same implementation has multiple top-level configurations, like **Library** for **Keil RTX5**
- Component vendor (Cvendor): The supplier of the software component (for example, **ARM**)
- Bundle (Cbundle): Allows you to combine multiple software components into a software bundle. Bundles have a different set of components available. All the components in a bundle are compatible with each other but not with the components of another bundle. For example, **ARM Compiler** for the **Compiler** component class.

Layer icons indicate which components are used in layers. In the current version, layers are read-only, so you cannot select or clear them from the **Software Components** view. Click the layer icon of a component to open the `*.clayer.yml` file or associated files.

Documentation links are available for some components at the class, group, or subgroup level. Click the **Learn more** link of a component to open the related documentation.

6.6.2 Modify the software components in your project

You can add components from all the packs available, not just the packs that are already selected for a project.

Procedure

1. In the **Project** drop-down list, select the project for which you want to modify software components.
2. In the **Target** drop-down list, select a specific target type, or, if you want to modify all the target types at once, select **All Targets** (note that some examples have only one target).
3. In the **Software packs** drop-down list, you can filter on the components available from the packs listed in your solution (**Solution** option), or display the components from all the packs available in the CMSIS ecosystem (**All packs** option).
4. Check that the **All** toggle button is selected to display all the components available, or switch to **Selected** to display only the components that are already selected.
5. Use the checkboxes to select or clear components as required. For some components, you can also select a vendor, variant, or version.
   The `cproject.yml` file is automatically updated.
6. Manage the dependencies between components and solve validation issues from the **Validation** panel.

   Issues are highlighted in red and have an exclamation mark icon next to them. You can remove conflicting components from your selection or add missing component dependencies from a suggested list.
7. If there are validation issues, move your cursor over the issues in the **Validation** panel to get more details. Click the proposed fixes to find the components in the list. In some cases, you might have to choose between different fix sets. Select a fix set in the drop-down list, make the required component choices, and then click **Apply**.
   If a pack is missing in the solution, a “Component’s pack is not included in your solution” message displays in the **Validation** panel. An error also displays in the **Problems** view. See Install CMSIS-Packs for information on how to install CMSIS-Packs.

   There can be other issues such as:

   - A component that you selected is incompatible with the selected hardware and toolchain.
   - A component that you selected has dependencies which are incompatible with the selected hardware and toolchain.
   - A component that you selected has unresolvable dependencies. In such cases, you must remove the component. Click **Apply** from the **Validation** panel.

6.6.3 Undo changes

In the current version, you can undo changes from the **Source Control** view or by directly editing the `cproject.yml` file.

6.7 Use the Configuration Wizard

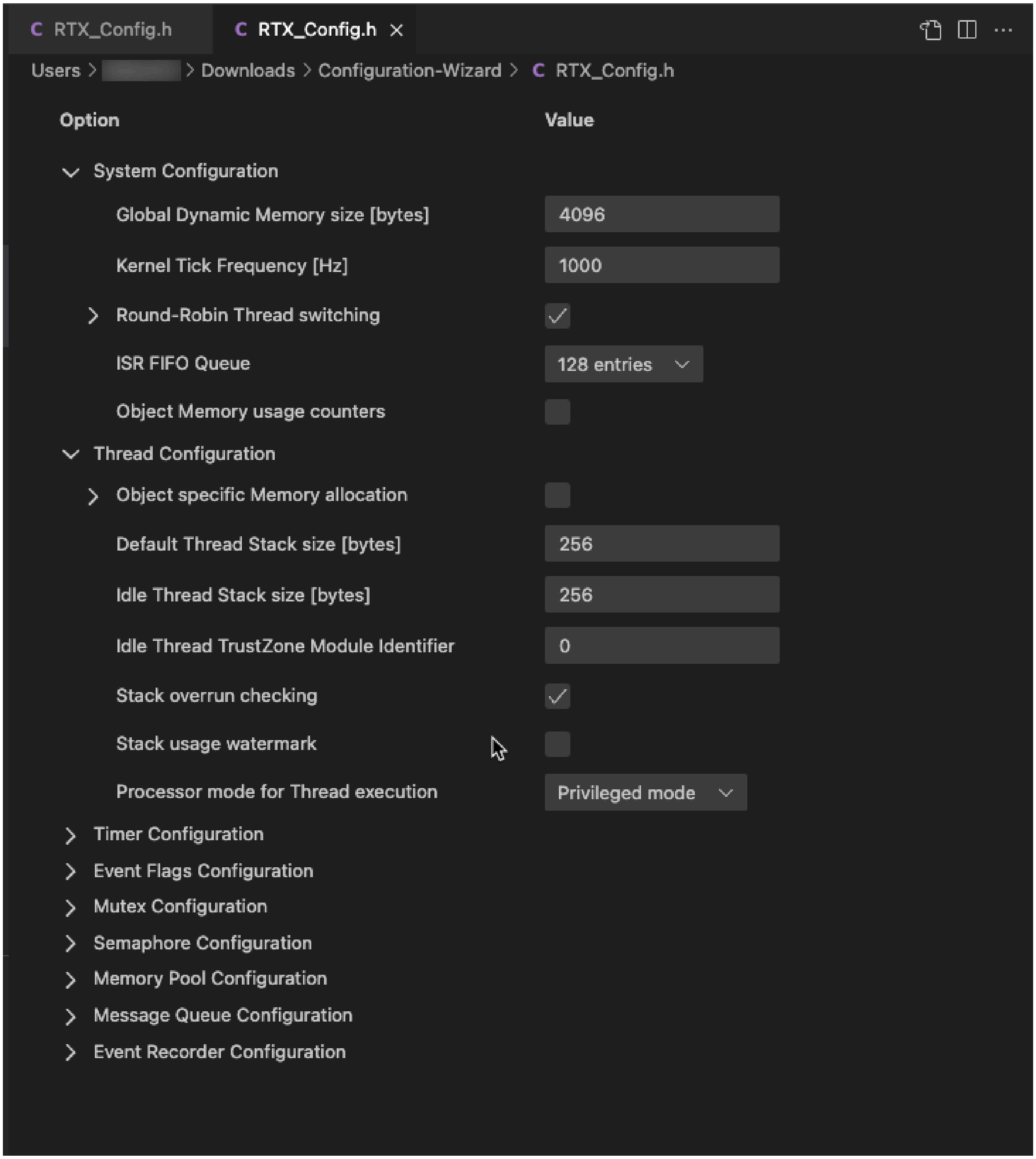
The Configuration Wizard simplifies the customization of startup code and configuration files by providing an intuitive dialog-style interface. It allows you to quickly modify configuration settings without the need for extensive manual editing.

The Configuration Wizard interface is generated from annotations that are included in the configuration files themselves.

These annotations, which are like embedded markup tags, can be already available in the configuration files of software components used in your project, or you can add them yourself. For the set of rules for creating these annotations, see Configuration Wizard Annotations in the Open-CMSIS-Pack documentation.

To open the Configuration Wizard, open a configuration file containing annotations, then click **Open Preview** . The Configuration Wizard displays.

Figure 6-3: Configuration Wizard



Changes you make in the Configuration Wizard are immediately reflected in the source file. You can also edit the source file directly.

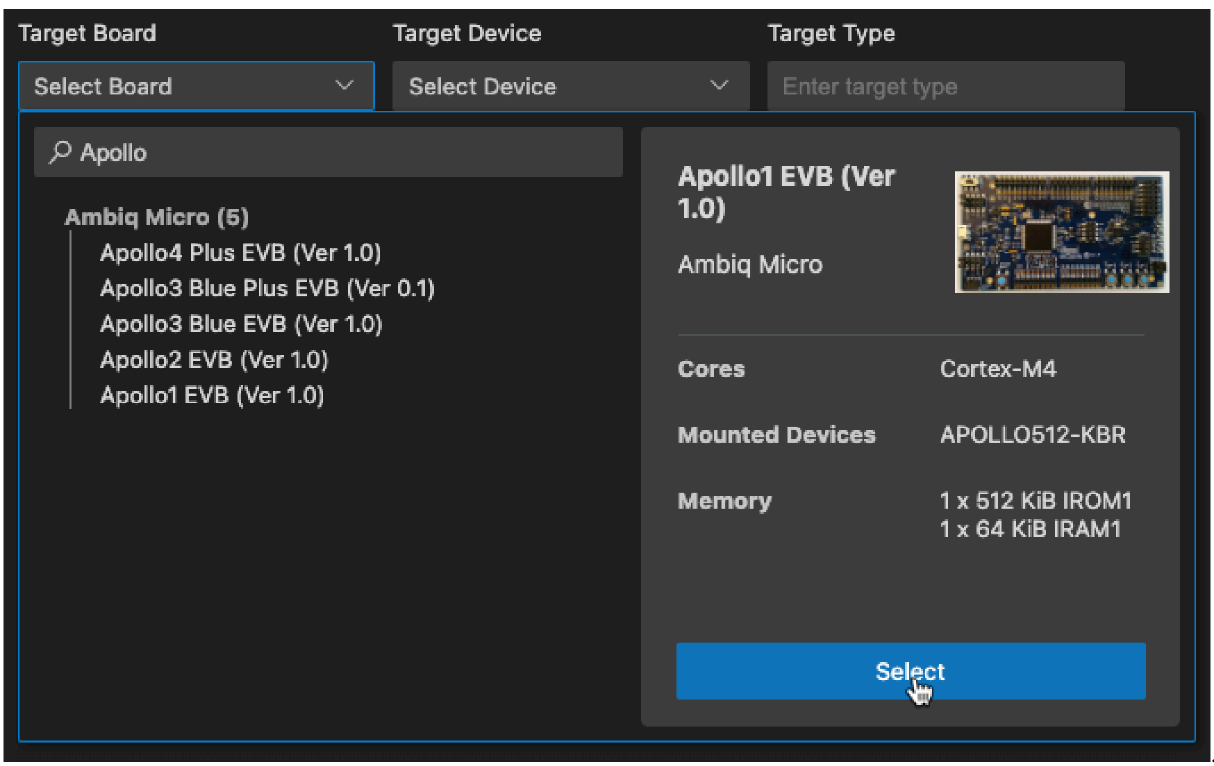
6.8 Create a solution

Create a solution project from scratch.

Procedure

1. To create a solution, either:
   - Click **CMSIS** in the Activity Bar to open the **CMSIS** view. Then:
     - If you are starting from an empty workspace, click **Create a New Solution**.
     - If you already have a solution opened in your workspace and want to create a new one in the same workspace, move your cursor over the **Solution outline**, and then click **More Actions** > **New Solution**.
   - Go to the **File** menu and select **New File...**, then select **CMSIS Solution** in the drop-down list that opens at the top of the window.

   The **Create New Solution** view opens.
2. Click the **Target Board** drop-down list. Enter a search term, and then select a board. A picker shows you the details of the board that you selected.



3. Click **Select**.
   The **Target Device** drop-down list and **Target Type** field are filled in by default with the name of the device mounted on the board that you selected.

   Alternatively, you can directly select a device in the **Target Device** drop-down list.

4. In the **Target Type** field, you can customize the name of the target hardware that is used to deploy the solution. The **Target Type** displays in the **Build Context** view and is set in the `<solution_name>.csolution.yml` file (`target-types: - type:`).
5. Select one of the following options from the **Templates and Examples** drop-down list: Note that the option or options available depend on the board or device that you selected.
    - Create a blank solution
    - Create a TrustZone solution. TrustZone is a hardware-based security feature that provides a secure execution environment on Arm-based processors. It allows the isolation of secure and non-secure zones, enabling the secure processing of sensitive data and applications. If the board or device that you selected is compatible, you can decide if your solution should use the TrustZone technology and define which project in the solution should use secure or non-secure zones.
    - Use an example project as a starting point
6. For blank and TrustZone solutions only, configure the projects in your solution:
    - If you selected `Blank solution`: One project is added for each processor in the target hardware. You can change the project names. You can decide to add `secure` or `non-secure` zones with the **TrustZone** drop-down list if the board or device is compatible. By default, TrustZone is `off`.
    - If you selected `TrustZone solution`: Two projects (a secure project and a non-secure project) are added for each processor in the target hardware that supports TrustZone. You can change the project names. You can also change the zones (`secure` or `non-secure`) in the **TrustZone** drop-down list, or remove TrustZone by selecting `off`.
7. Click **Add Project** to add projects to your solution and configure them. For TrustZone, you can add as many `secure` or `non-secure` projects as you need for a particular processor.
8. For blank and TrustZone solutions only, select a compiler: **Arm Compiler 6**, **GCC**, or **LLVM**.
9. You can change the name for your solution in the **Solution Name** field.
10. Click **Browse** next to the **Solution Location** field and choose where to store the files of the solution using the system dialog box that opens.
11. With the **Initialize Git repository** checkbox, you can initialize the solution as a Git repository. Clear the checkbox if you do not want to turn your solution into a Git repository.
12. Click **Create**.
    The extension creates the solution. Examples available only in `*.uvprojx` format are converted automatically. If there are conversion errors, check the `uv2csolution.log` file available.

    A dialog box displays. You can carry out the following tasks:
    - Open the solution in a new workspace (**Open** option)
    - Open the solution in a new window and new workspace (**Open project in new window** option)
    - Add the solution to the current workspace (**Add project to vscode workspace** option)
13. Select one of the options.
    The extension also generates a `vcpkg-configuration.json` file with the tools that you need to set up your development environment. An **Arm Environment Activation** dialog box displays.

14. Confirm that the Environment Manager extension can automatically activate the workspace and download the tools specified in your vcpkg-configuration.json file.
 Missing CMSIS-Packs are installed automatically. A pop-up message displays in the bottom right-hand corner to confirm the installation.
15. Check that the files for the solution have been created:
 - A vcpkg-configuration.json file
 - A <solution_name>.csolution.yml file
 - One or more <project_name>.cproject.yml files, each available in a separate folder
 - A main.c template file for each project

Next steps

Explore the autocomplete feature available to edit the csolution.yml and cproject.yml files. Read the CMSIS-Toolbox > Build Overview documentation for project examples.

Add CMSIS components with the **Software Components** view. When you add components, the cproject.yml files are updated.

6.9 Convert a Keil µVision project to a solution

You can convert any Keil® µVision® project to a solution from the CMSIS Solution extension. Note that the conversion does not work with Arm® Compiler 5 (AC5) projects. You can download Arm Compiler 5 projects from the website, but you cannot use them with the extensions. Only Arm Compiler 6 projects can be converted. As a workaround, you can update Arm Compiler 5 projects to Arm Compiler 6 in Keil µVision, then convert the projects to solutions in Visual Studio Code. For more help and information on converting to Arm Compiler 6, see the Migrate Arm Compiler 5 to Arm Compiler 6 application note and the Arm Compiler for Embedded Migration and Compatibility Guide.

Procedure

1. Open the project that contains the *.uvprojx that you want to convert in Visual Studio Code.
2. Right-click the *.uvprojx and select **Convert uvproj to csolution** from the **Explorer**.
 The conversion starts immediately.

 Alternatively, if you are starting from an empty workspace, you can click **CMSIS** in the Activity Bar to open the **CMSIS** view. Then choose one of the following two options:
 - Click **Convert a µVision Project** and open your *.uvprojx file to convert it
 - Move your cursor over the **Solution outline**, click **More Actions**, then select **Convert uvproj to csolution** and open your *.uvprojx file to convert it

 A dialog box displays. You can carry out the following tasks:
 - Open the solution in a new workspace (**Open** option)

- Open the solution in a new window and new workspace (**Open project in new window** option)

  You can also run the **CMSIS: Convert uvproj to csolution** command from the Command Palette. In that case, select the `*.uvprojx` that you want to convert on your machine and click **Select**.

  If there are conversion errors, check the `uv2csolution.log` file available.

3. Confirm that the Environment Manager extension can automatically activate the workspace and download the tools specified in your `vcpkg-configuration.json` file.
4. Check the **Output** tab (**View** > **Output**). Select **µVision to Csolution Conversion** in the drop-down list on the right-hand side of the **Output** tab.
   The `*.cproject.yml` and `*.csolution.yml` files are available in the folder where the `*.uvprojx` is stored.

6.10 Configure a build task

In Visual Studio Code, you can automate certain tasks by configuring a file called `tasks.json`. See Integrate with External Tools via Tasks for more details.

With the CMSIS Solution extension, you can configure a build task using the `tasks.json` file to build your projects. When you run the build task, the extension runs `cbuild` with the options that you defined.

Note

As mentioned in Get started with an example project, the examples provided on keil.arm.com are shipped with a `tasks.json` file that already contains some configuration settings to build your project. You can modify the default configuration if needed.

If you are working with an example for which no build task has been configured yet, follow these steps:

1. Go to **Terminal** > **Configure Tasks...**.
2. In the drop-down list that opens at the top of the window, select the **CMSIS Build** task.

   A `tasks.json` file opens with the default configuration.
3. Modify the configuration.

   With IntelliSense, you can see the full set of task properties and values available in the `tasks.json` file. You can bring up suggestions using **Trigger Suggest** from the Command Palette. You can also display the task properties specific to `cbuild` by typing `cbuild --help` in the **Terminal**.
4. Save the `tasks.json` file.

Alternatively, you can define a default build task using **Terminal** > **Configure Default Build Task...**. The **Terminal** > **Run Build Task...** option triggers the execution of default build tasks.

6.11 Initialize your solution

If you have a solution that does not already contain a `vcpkg-configuration.json` file, a `tasks.json` file, and a `launch.json` file, you can use the **Initialize CMSIS project** option to generate these files and start working with your project. Examples from keil.arm.com or solutions created from scratch from the **Create New Solution** view already contain the JSON files required.

Procedure

1. From the **Explorer**, open your workspace.
2. Right-click anywhere in the workspace and select **Initialize CMSIS project**.
   The extension generates a `vcpkg-configuration.json` file, a `tasks.json` file, and a `launch.json` file that are already preconfigured.

6.12 Use the CMSIS csolution API

If you want to create your own Visual Studio Code csolution extension, the CMSIS Solution extension exposes an API that other extensions can use.

For the API specification, see the CMSIS csolution extension API page.

For information about authoring extensions, see the Extension API chapter in the Visual Studio Code documentation.

For solution examples, go to keil.arm.com.

7. Device Manager extension

Look at the hardware supported with the Keil® Studio extensions.

Then, manage your hardware with the Device Manager:

- Connect your hardware
- Edit your hardware
- Open a serial monitor

A full list of commands and settings is available for the **Arm Device Manager** extension. To view the list, click **Extensions** in the Visual Studio Code Activity Bar. Click **Arm Device Manager** in the list of extensions, and then click **Features**.

7.1 Supported hardware

Describes the hardware that the Device Manager extension and other Keil® Studio extensions support.

7.1.1 Supported development boards and MCUs

The extensions support the development boards and MCUs available on keil.arm.com.

7.1.2 Supported debug probes

The following debug probes are supported.

7.1.2.1 WebUSB-enabled CMSIS-DAP debug probes

The extensions support debug probes that implement the CMSIS-DAP protocol, such as:

- The DAPLink implementation: see the ARMmbed/DAPLink repository
- The LPC-Link2 implementation: see the LPC-Link2 documentation
- The Nu-Link2 implementation: see the Nuvoton repository
- The ULINKplus™ (firmware version 2) implementation: see the Keil MDK documentation

See the CMSIS-DAP documentation for general information.

7.1.2.2 ST-LINK debug probes

The extensions support ST-LINK/V2 probes and later, and the ST-LINK firmware available for these probes.

The recommended debug implementation versions of the ST-LINK firmware are:

- For ST-LINK/V2 and ST-LINK/V2-1 probes: J36 and later
- For STLINK-V3 probes: J6 and later

See “Firmware naming rules” in Overview of ST-LINK derivatives for more details on naming conventions.

7.2 Connect your hardware

Describes how to connect your hardware for the first time.

Procedure

1. Click **Device Manager** in the Activity Bar.
2. Connect your hardware to your computer over USB.
   The hardware is detected and a pop-up message displays in the bottom right-hand corner.
3. Click **OK** to use the hardware.

   Alternatively, click **Add Device** Add Device and select your hardware in the drop-down list that displays at the top of the window.

   Your hardware is now ready to be used to run and debug a project.

Next steps

If you need to add more hardware, click **Add Device** in the top right-hand corner.

7.3 Edit your hardware

If your board cannot be detected or if you are using an external debug probe, you can edit the hardware entry from the Device Manager and specify a Device Family Pack (DFP) and a device name retrieved from the pack to be able to work with your hardware. DFPs handle device support.

Procedure

1. Move your cursor over the hardware that you want to edit and click **Edit Device** .
2. Edit the hardware name in the field that displays at the top of the window if needed and press **Enter**. This is the name that displays in the Device Manager.
3. Select a Device Family Pack (DFP) CMSIS-Pack for your hardware in the drop-down list.
4. Select a device name to use from the CMSIS-Pack in the field and press **Enter**.

7.4 Open a serial monitor

Open a serial monitor. The serial output shows the output of your board. You can use the serial output as a debugging tool or to communicate directly with your board.

Procedure

1. Move your cursor over the hardware for which you want to open a serial monitor and click **Open Serial** .
   A drop-down list displays at the top of the window where you can select a baud rate (the data rate in bits per second between your computer and your hardware). To view the output of your hardware correctly, you must select an appropriate baud rate. The baud rate that you select must be the same as the baud rate of your active project.
2. Select a baud rate.
   A **Terminal** tab opens with the baud rate selected.

8. Arm Debugger extension

Run a project on your hardware with Arm Debugger and start an Arm Debugger debug session with the Arm Debugger extension.

For a full list of commands with usage instructions and examples for the Arm Debugger engine, see the Arm Debugger Command Reference guide. You can also view a list of commands and settings in the **Features** tab (Windows) or the **Feature Contributions** tab (macOS) for the extension. Click **Extensions** in the Visual Studio Code Activity Bar, click **Arm Debugger** in the list of extensions, and then click **Features** or **Feature Contributions**.

Note

Most examples provided on keil.arm.com come with `tasks.json` and `launch.json` files that contain run and debug configuration settings. You can modify the default configuration if needed.

8.1 Run your project on your hardware with Arm Debugger

Find out how to configure a task to run your project on your hardware and what the configuration options are.

8.1.1 Configure a task

To run a project on your hardware, you must first configure a task. The task transfers the binary into the appropriate memory locations on the hardware's flash memory.

Use the `arm-debugger.flash: Flash Device` task. The CMSIS-Packs used in your project control the flash download.

Note

Most examples provided on keil.arm.com come with a `tasks.json` file that contains run configuration settings. You can modify the default configuration if needed.

Procedure

1. Open the Command Palette. Search for `Tasks: Configure Task` and then select it.

   Alternatively, go to the **Terminal** menu and select **Configure Tasks...**.
2. Select the `arm-debugger.flash: Flash Device` task (or **Flash Device**).

This task adds default run configuration options in the `tasks.json` file in the `.vscode` folder of the project.

3. Save the `tasks.json` file.

8.1.2 Override or extend the default run configuration options for Arm Debugger

You can override or extend the default configuration options. See Arm Debugger run configuration options.

In order to flash a hardware device, the task configuration must know which CMSIS-Pack to read information from and the device name in the CMSIS-Pack to use. These settings are named `cmsisPack` and `deviceName`, and you can specify them in multiple ways.

If your target hardware is automatically detected, or if you have set the pack and device name for your hardware, the task configuration can automatically pick this up by using the following code:

```
{
    [...]
    "serialNumber": "${command:device-manager.getSerialNumber}",
    "cmsisPack": "${command:cmsis-csolution.getTargetPack}",
    "deviceName": "${command:device-manager.getDeviceName}",
    [...]
}
```

Alternatively, you can specify these settings directly as a full path to the CMSIS-Pack file or a folder on your machine:

```
{
    [...]
    "serialNumber": "${command:device-manager.getSerialNumber}",
    "cmsisPack": "/Users/me/mypack.pack",
    "deviceName": "STM32H745XIHx",
    [...]
}
```

You can also use the short code for the CMSIS-Pack in the format `<vendor>::<pack>@<version>`:

```
{
    [...]
    "serialNumber": "${command:device-manager.getSerialNumber}",
    "cmsisPack": "Keil::STM32H7xx_DFP@3.1.0",
    "deviceName": "STM32H745XIHx",
    [...]
}
```

Note that this code triggers an automatic download of the CMSIS-Pack.

Note

If you do not have the CMSIS Solution extension installed, then you can use:

- `"cmsisPack": "${command:device-manager.getDevicePack}"` instead of `"cmsisPack": "${command:cmsis-csolution.getTargetPack}"`
- `"deviceName": "${command:device-manager.getDeviceName}"` instead of `"deviceName": "${command:cmsis-csolution.getDeviceName}"`

8.1.3 Arm Debugger run configuration options

The extension provides the following run configuration options.

Configuration option	Description
`"cmsisPack"`	Path (file or URL) to a DFP (Device Family Pack) CMSIS-Pack for your hardware. Can be used with: • `cmsis-csolution.getTargetPack`: Gets the DFP CMSIS-Pack for the selected target type in the **CMSIS Solution Build Context** view. `cmsis-csolution.getTargetPack` is specific to your solution. • `device-manager.getDevicePack`: Gets the DFP CMSIS-Pack for the selected device. This command uses the latest pack available in the pack index.
`"connectMode"`	Connection mode. Possible values: • `auto`: Debugger decides • `haltOnConnect`: Halts for any reset before running • `underReset`: Holds external NRST line asserted • `preReset`: Prereset using NRST • `running`: Connects to running target without altering state Default: `auto`
`"dbgconf"`	Path to a `.dbgconf` file to configure CMSIS-Pack debug sequence execution. Requires Arm Debugger v6.1.0 or later.
`"debugClockSpeed"`	Maximum clock frequency for the debug communication. The frequency actually used depends on the debug probe. `auto` uses a target-specific default. Requires Arm Debugger v6.0.2 or later. Possible values: `auto`, `50MHz`, `33MHz`, `25MHz`, `20MHz`, `10MHz`, `5MHz`, `2MHz`, `1MHz`, `500kHz`, `200kHz`, `100kHz`, `50kHz`, `20kHz`, `10kHz`, `5kHz` Default: `auto`
`"debugPortMode"`	Debug port mode to use for the debug connection. Requires Arm Debugger v6.0.2 or later. Possible values: `auto`, `JTAG`, `SWD` Default: `auto`

Configuration option	Description
`"deviceName"`	CMSIS-Pack device name. Can be used with: • `cmsis-csolution.getDeviceName`: Gets the device name from the information available for the probe or board in the `*.csolution.yml` file of your solution • `device-manager.getDeviceName`: Gets the device name from the DFP of the selected device
`"eraseMode"`	Type of flash erase to use. Requires Arm Debugger v6.1.0 or later. Possible values: • `sectors`: Erase only sectors to be programmed • `none`: Skip flash erase Default: `sectors`
`"flms"`	Flash algorithm configurations. Each entry either modifies a default algorithm that is defined in the corresponding DFP (Device Family Pack), or adds an additional algorithm. Use the 'ignore' field to deactivate a default algorithm from the DFP. Requires Arm Debugger v6.1.0 or later. Required value: • "`path`": Relative path to the flash algorithm file in the DFP, or absolute path to a flash algorithm file in your machine's file system Optional values: • "`regionStart`": Start address of the memory region targeted by the flash algorithm in decimal or hexadecimal format. Uses the default start address for the algorithm from the DFP CMSIS-Pack if not set. • "`regionSize`": Size of the memory region targeted by the flash algorithm in decimal or hexadecimal format. Uses the default size for the algorithm from the DFP CMSIS-Pack if not set. • "`ramStart`": Start address of target system's RAM used for execution of flash algorithms. Uses defaults from the DFP CMSIS-Pack if not set. • "`ramSize`": Size of target system's RAM used for execution of flash algorithms. Uses defaults from the DFP CMSIS-Pack if not set. • "`ignore`": Ignores an algorithm as provided in the DFP. Value can be `true` or `false`. If not set, algorithms marked as default in the DFP have an ignore value of `false`, whereas algorithms not marked as default have an ignore value of `true`. Use this option to: ◦ Disable default algorithms from the DFP. For example, to override the default algorithms with a local version. ◦ Enable non-default algorithms. For example, for external flash memories that depend on the target board design. Algorithms are usually marked as default in a DFP if they are expected to be applicable for the majority of use cases.
`"openSerial"`	Baud rate to open the serial output of a device after flash (requires Arm Device Manager). Possible values: `115200`, `57600`, `38400`, `19200`, `9600`, `4800`, `2400`, `1800`, `1200`, `600`
`"pdsc"`	Path (file or URL) to a PDSC file.

Configuration option	Description
`"probe"`	Name of probe to use for the debug connection. Possible values: `ULINKpro`, `ULINKpro D`, `ULINK2`, `CMSIS-DAP`, `ULINKplus`, `ST-Link` Default: `CMSIS-DAP`
`"processorName"`	CMSIS-Pack processor name for multicore devices.
`"program"` or `"programs"`	Path or paths (file or URL) to one or more projects to use in order of loading. Requires Arm Debugger v6.0.2 or later. Can be used with: • `arm-debugger.getApplicationFile`: Returns an AXF or ELF file used for CMSIS run and debug
`"serialNumber"`	Serial number of the connected USB hardware to use. Can be used with: • `device-manager.getSerialNumber`: Gets the serial number of the selected device
`"targetAddress"`	Synonymous with `serialNumber`.
`"targetInitScript"`	Path to a target initialization script (`.ds`/`.py`) executed after connection but before any other operation. Requires Arm Debugger v6.1.1 or later.
`"vendorName"`	CMSIS-Pack vendor name.
`"verifyFlash"`	Verify the contents downloaded to flash. Requires Arm Debugger v6.1.0 or later.
`"workspaceFolder"`	Current Arm Debugger workspace folder. Default: `"${workspaceFolder}"`

Other Visual Studio Code options are also available. Use the **Trigger Suggestions** command (**Ctrl+Space**) to see what is available and read the Visual Studio Code documentation on tasks, as well as the Schema for tasks.json page.

8.1.4 Use the Run and Debug Configuration visual editor for your run configuration

As an alternative to editing the `tasks.json` file of your solution to change the run configuration options, you can use the **Run and Debug Configuration** visual editor.

Procedure

1. To open the editor, either:
    - From the **Explorer**, right-click the `tasks.json` file that is stored in the `.vscode` folder of the solution and select **Open Run and Debug Configuration**.
    - From the **Explorer**, right-click the `tasks.json` file and select **Open With...**, then select **Run and Debug Configuration** in the drop-down list that displays at the top of the window.
    - If the `tasks.json` file is already open in the editor, click **Open Run and Debug Configuration**  in the top right-hand corner.

2. You can define several run configurations in the `tasks.json` file. In the **Selected Configuration** drop-down list, select **New Configuration** to add a new configuration block in the JSON file. You can also click **Duplicate** to duplicate the currently selected configuration and modify it.
3. Modify your run configuration:
    - You can change the name of the configuration in the **Configuration Name** field.
    - **Probe Type**: In the drop-down list, select a type for the debug probe that you are using or the debug unit on your board.
        - Default value: `CMSIS-DAP`. If the Arm Debugger extension cannot set the probe type automatically, the default value is `CMSIS-DAP`.
        - If you have connected a probe or a board over USB to your computer, the Arm Debugger extension sets a probe type based on the serial number of the hardware detected.
    - **Serial Number**: In the drop-down list, select the serial number of the debug probe or debug unit on your board.
        - Default value: `auto`. With `auto`, the serial number of the active device in the Arm Device Manager extension is used by default. The `"${command:device-manager.getSerialNumber}"` command is added in the JSON file for `"serialNumber"`.
        - You can also select the serial number of the active device in the drop-down list, or directly type a serial number.

      Click **Open Arm Device Manager** to check what your active device is.
    - **CMSIS-Pack**: Select the Device Family Pack (DFP) for the target debug probe or board.
        - Default value: `auto (CMSIS Solution)`. The DFP for the active device defined in the `*.csolution.yml` file of your solution is used by default. The `"${command:cmsis-csolution.getTargetPack}"` command is added in the JSON file for `cmsisPack`.
        - `auto (Device Manager)`: The DFP for the active device in the Arm Device Manager extension is used by default. The `"${command:device-manager.getDevicePack}"` command is added in the JSON file for `cmsisPack`.
        - You can also select the DFP for the active device in the drop-down list, or directly type the name of a DFP in the format <vendor>::<pack>@<version>. For example: `ARM::V2M_MPS3_SSE_300_BSP@1.4.0`.
    - **CMSIS-Pack Device Name**: Select the name of the target device (target chip on your board).
        - Default value: `auto (CMSIS Solution)`. The device name is deduced from the information available for the probe or board in the `*.csolution.yml` file of your solution. The `"${command:cmsis-csolution.getDeviceName}"` command is added in the JSON file for `deviceName`.
        - `auto (Device Manager)`: The device name is deduced from the information available for the probe or board in the Arm Device Manager extension. The `"${command:device-manager.getDeviceName}"` command is added in the JSON file for `deviceName`.
        - You can also select the device name in the drop-down list, or directly type the device name. For example: `MPS3_SSE_300`. The device name available in the drop-down list is the one defined in the `*.csolution.yml` file of your solution.

- **Processor Name**: If you are using a device with multiple cores, select the processor to use.
    - Default value: `auto`. With `auto`, the processor name defined in the `*.csolution.yml` file of your solution is used by default. The `"${command:cmsis-csolution.getProcessorName}"` command is added in the JSON file for `"processorName"`.
    - You can also directly type a processor name. For example: `cm4`.
- **Program Files**: One or more programs to run on your hardware
    - Default value: The `${command:arm-debugger.getApplicationFile}` command is added in the JSON file for `"program"` when you add a new configuration block. This command detects the latest AXF or ELF file generated.
    - Click **Add File** to point to a file directly. You can add as many files as you need. Arm Debugger uses the files in the order in which you added them. AXF and ELF files are supported by default. You can also add other file types.
    - Click **Detect File** to add the `${command:arm-debugger.getApplicationFile}` command if it is not available.
    - Move your cursor over the name of the command or a file and click the delete icon  to remove the selection.
- **Connection Mode**: Select a connection mode. The connection mode controls the operations that are run when the debugger connects to the target debug probe or the board.
    - Default value: `auto`. The debugger decides which connect mode to use based on the connected target device. For ST boards, when `auto` is selected, `underReset` is used. For other boards, `haltOnConnect` is used.
    - `haltOnConnect`: Stops the CPU of the target debug probe or board for a reset before the flash download.
    - `underReset`: Asserts the hardware reset during the connection.
    - `preReset`: Triggers a hardware reset pulse before the connection.
    - `running`: Connects to the CPU without stopping the program execution during the connection.
- **Port Mode**: Select a debug port mode to use. A debug port allows you to communicate with and debug microcontrollers or other embedded systems.
    - Default value: `auto`. With `auto`, the debugger decides which debug port mode to use based on the connected target device.
    - `JTAG`: Use the JTAG debug port mode.
    - `SWD`: Use the SWD debug port mode.
- **Clock Speed**: The maximum clock frequency for the debug communication. The clock frequency is the speed at which data is transferred between the debugger and the target device during debugging operations. The frequency actually used depends on the capabilities of the debug probe and might be reduced to the next supported frequency.
    - Default value: `auto`. With `auto`, the debugger decides which clock frequency to use based on the connected target device.

    - Other possible values: `50MHz`, `33MHz`, `25MHz`, `20MHz`, `10MHz`, `5MHz`, `2MHz`, `1MHz`, `500kHz`, `200kHz`, `100kHz`, `50kHz`, `20kHz`, `10kHz`, `5kHz`.

- **Erase Mode**: The type of flash erase to use.
    - Default value: `sectors`. With `sectors`, only the sectors of the flash memory to be programmed are erased. All the data within these specific sectors is erased.
    - `none`: Skip flash erase. The contents of the flash memory are not erased before programming.
- **Verify Flash**: Select this checkbox to verify the contents downloaded to the flash memory during the flash download.
- **Flash Algorithms**: Default flash algorithms are available in the Device Family Pack (DFP) of your solution. You can also create your own algorithms and use them in the configuration. See the Open-CMSIS-Pack documentation for more information. Select the flash algorithms that you want to use or click the checkboxes to clear the selection. Algorithms available and marked as `default` in DFPs are selected by default.

  The following fields are available:
    - **Path**: Relative path to a default flash algorithm file in the DFP, or absolute path to a flash algorithm file in your machine's file system
    - **Region Start**: Start address of the memory region targeted by the selected flash algorithm
    - **Region Size**: Size of the memory region targeted by the selected flash algorithm
    - **Ram Start**: Start address of the target system's RAM used for the execution of the flash algorithm
    - **Ram Size**: Size of the target system's RAM used for the execution of the flash algorithm

  Region Start, **Region Size**, **Ram Start**, and **Ram Size** can be expressed in decimal or hexadecimal format. If they are not set, then the default values from the DFP are used.

  To add your own flash algorithms, edit the `tasks.json` file manually:

  a. Add `"path"` under the `"flms"` key in the JSON file. For example:

```
"flms": [
{
 "path": "<Path to FLM file>"
}
]
```

  b. Fill in the **Region Start**, **Region Size**, **Ram Start**, and **Ram Size** fields from the visual editor, or edit the JSON file directly. For example:

```
"flms": [
{
 "path": "<Path to FLM file>",
 "regionStart": "<Start address of memory region>",
 "regionSize": "<Size of memory region>",
 "ramStart": "<Start address of target system's RAM>",
 "ramSize": "<Size of target system's RAM>"
}
```

```
    ]
```

- **Baud Rate**: Select a baud rate to view the serial output of the target debug probe or board correctly. Possible values: `115200`, `57600`, `38400`, `19200`, `9600`, `4800`, `2400`, `1800`, `1200`, `600`.

4. If **Auto Save** is not enabled (**File** > **Auto Save**), save your changes.
   The `tasks.json` file is updated.

8.1.5 Run your project

Run the project on your hardware.

Before you begin

When you have several solutions grouped in a single folder on your machine, Visual Studio Code does not take into account the `tasks.json` and `launch.json` files that you might have created for each solution. Instead, Visual Studio Code generates new JSON files at the root of the workspace in a `.vscode` folder and ignores the other JSON files.

As a result, you might have issues running or debugging a project.

As a workaround, open one solution first, then add other solutions to your workspace with the **File** > **Add Folder to Workspace** option.

Procedure

1. Check that your hardware is connected to your computer.
2. Open the Command Palette. Search for `Tasks: Run Task` and then select it.
3. Select `arm-debugger.flash: Flash Device` in the drop-down list.
   Alternatively, if you have installed the Keil Studio Pack, go to the **CMSIS** view, open the Build Context view, and check which run configuration is selected. Then, click **Run** in the **Solution outline** header.

   Run icons are available at the solution level or at the project level depending on what you selected in the **Build Context** view for the run configuration. See Set a context for your solution for more details.
4. If you are using a device with multiple cores and you did not specify a `"processorName"` in the `tasks.json` file, and you do not have the CMSIS Solution extension installed, then you must select the appropriate processor for your project in the **Select a processor** drop-down list that displays at the top of the window.
5. Check the **Terminal** tab to verify that the project has run correctly.
   If the Arm Debugger engine cannot be found on your machine, an **Arm Debugger not found** dialog box displays.

   Select one of these options:

   - Click **Install Arm Debugger** to add it to your environment. The `vcpkg-configuration.json` file is updated. Check the Arm tools installed in the status bar Arm Tools: 8.
   - Click **Configure Path** to indicate the path to the Arm Debugger engine in the settings.

8.2 Debug your project with Arm Debugger

Debug a project.

8.2.1 Add configuration

As is the case for running a project, you must first add a launch configuration to debug your project. Creating a launch configuration file allows you to configure and save debugging setup details. Visual Studio Code keeps debugging configuration information in a `launch.json` file. If no configuration is detected, you get an error. You are prompted to open the `launch.json` file and add a launch configuration for Arm Debugger.

Note

Most examples provided on keil.arm.com come with a `launch.json` file that contains debugging configuration settings. You can modify the default configuration if needed.

Procedure

1. Open the Command Palette. Search for `Debug: Add Configuration` and then select it.

   The `launch.json` file opens.

   Alternatively, go to the **Run** menu and select **Add Configuration...**.
2. Select the `Arm Debugger: Attach`, `Arm Debugger: Launch`, or `Arm Debugger: Launch FVP` task.

   This adds default debug configuration options in the `launch.json` file in the `.vscode` folder of the project.
3. Save the `launch.json` file.

Note

If you do not have the CMSIS Solution extension installed, then you can use:

- `"cmsisPack": "${command:device-manager.getDevicePack}"` instead of `"cmsisPack": "${command:cmsis-csolution.getTargetPack}"`
- `"deviceName": "${command:device-manager.getDeviceName}"` instead of `"deviceName": "${command:cmsis-csolution.getDeviceName}"`

8.2.2 Override or extend the default debug configuration options for Arm Debugger

You can override or extend the default configuration options as required. See Arm Debugger debug configuration options for more details.

See also the details provided for the tasks.json file for `cmsisPack` and `deviceName`. In order to debug a hardware device, the launch configuration must know which CMSIS-Pack to read information from and the device name in the CMSIS-Pack to use.

8.2.3 Arm Debugger debug configuration options

The extension provides the following debug configuration options.

Configuration options for debugging with a physical target

Configuration option	Description
`"cdbEntry"`	Arm Debugger Configuration Database Entry to select.
`"cdbEntryParams"`	One or more key/value settings specific to the selected `cdbEntry`. Example: `model_params: -f ${workspaceFolder}/model_config.txt`
`"cmsisDevice"`	Concatenation of CMSIS-Pack name, device vendor, device name, and processor name (if multicore). Deprecated. Use `cmsisPack`, `pdsc`, `vendorName`, `deviceName`, and `processorName` instead.
`"cmsisPack"`	Path (file or URL) to a DFP (Device Family Pack) CMSIS-Pack for your hardware. Can be used with: • `cmsis-csolution.getTargetPack`: Gets the DFP CMSIS-Pack for the selected target type in the **CMSIS Solution Build Context** view. `cmsis-csolution.getTargetPack` is specific to your solution. • `device-manager.getDevicePack`: Gets the DFP CMSIS-Pack for the selected device. This command uses the latest pack available in the pack index.
`"connectMode"`	Connection mode. Possible values: • `auto`: Debugger decides • `haltOnConnect`: Halts for any reset before running • `underReset`: Holds external NRST line asserted • `preReset`: Prereset using NRST • `running`: Connects to running target without altering state Default: `auto`
`"dbgconf"`	Path to a `.dbgconf` file to configure CMSIS-Pack debug sequence execution. Requires Arm Debugger v6.1.0 or later.

Configuration option	Description
"debugClockSpeed"	Maximum clock frequency for the debug communication. Actually used frequency depends on used debug probe. `auto` uses a target-specific default. Requires Arm Debugger v6.0.2 or later. Possible values: `auto`, `50MHz`, `33MHz`, `25MHz`, `20MHz`, `10MHz`, `5MHz`, `2MHz`, `1MHz`, `500kHz`, `200kHz`, `100kHz`, `50kHz`, `20kHz`, `10kHz`, `5kHz` Default: `auto`
"debugFrom"	The symbol the debugger will run to before debugging. Default: `"main"`
"debugInitScript"	Path to a debug initialization script (`.ds`/`.py`) executed after connection and running to `debugFrom`. Requires Arm Debugger v6.1.0 or later.
"debugPortMode"	Debug port mode to use for the debug connection. Requires Arm Debugger v6.0.2 or later. Possible values: `auto`, `JTAG`, `SWD` Default: `auto`
"deviceName"	CMSIS-Pack device name. Can be used with: • `cmsis-csolution.getDeviceName`: Gets the device name from the information available for the probe or board in the `*.csolution.yml` file of your solution • `device-manager.getDeviceName`: Gets the device name from the DFP of the selected device
"pathMapping"	A mapping of remote paths to local paths to resolve source files.
"pdsc"	Path (file or URL) to a PDSC file.
"probe"	Name of probe to use for the debug connection. Possible values: `ULINKpro`, `ULINKpro D`, `ULINK2`, `CMSIS-DAP`, `ULINKplus`, `ST-Link` Default: `CMSIS-DAP`
"processorName"	CMSIS-Pack processor name for multicore devices.
"program" or "programs"	Path or paths (file or URL) to one or more projects to use in order of loading. Requires Arm Debugger v6.0.2 or later. Can be used with: • `arm-debugger.getApplicationFile`: Returns an AXF or ELF file used for CMSIS run and debug
"programMode"	Mode to program an application to a target. Possible values: `auto`, `flash`, `ram`, `mixed` Default: `auto`
"resetAfterConnect"	Resets the device after having acquired control of the processor.

Configuration option	Description
`"resetMode"`	Type of reset to use. Possible values: • `auto`: Debugger decides • `system`: Use ResetSystem sequence • `hardware`: Use ResetHardware sequence • `processor`: Use ResetProcessor sequence Default: `auto`
`"searchPaths"`	Array of paths to source locations.
`"serialNumber"`	Serial number of the connected USB hardware to use. Can be used with: • `device-manager.getSerialNumber`: Gets the serial number of the selected device
`"svd"`	Path (file or url) to an SVD file.
`"targetAddress"`	Synonymous with `serialNumber`.
`"targetInitScript"`	Path to a target initialization script (`.ds/.py`) executed after connection but before any other operation. Requires Arm Debugger v6.1.1 or later.
`"vendorName"`	CMSIS-Pack vendor name.
`"workspaceFolder"`	Current Arm Debugger workspace folder. Default: `"${workspaceFolder}"`

Configuration options for debugging with a virtual target (Fixed Virtual Platforms)

Note

FVPs are natively available on Windows and Linux only. If you are on Mac, you can use Docker to run FVPs in a Linux container. Follow this Arm Developer Learning Path to install Docker and clone the https://github.com/Arm-Examples/FVPs-on-Mac repository.

Configuration option	Description
`"cdbEntry"`	Arm Debugger Configuration Database Entry to select.
`"cdbEntryParams"`	One or more key/value settings specific to the selected `cdbEntry`. Example: `model_params: -f ${workspaceFolder}/model_config.txt`
`"debugFrom"`	The symbol the debugger will run to before debugging. Default: `"main"`
`"debugInitScript"`	Path to a debug initialization script (`.ds/.py`) executed after connection and running to `debugFrom`. Requires Arm Debugger v6.1.0 or later.
`"fvpParameters"`	Path to an FVP parameter configuration file.
`"pathMapping"`	A mapping of remote paths to local paths to resolve source files.

Configuration option	Description
`"program"` or `"programs"`	Path or paths (file or URL) to one or more projects to use in order of loading. Requires Arm Debugger v6.0.2 or later. Can be used with: • `arm-debugger.getApplicationFile`: Returns an AXF or ELF file used for CMSIS run and debug
`"programMode"`	Mode to program an application to a target. Default value: `ram`
`"searchPaths"`	Array of paths to source locations.
`"svd"`	Path (file or url) to an SVD file.
`"targetInitScript"`	Path to a target initialization script (`.ds`/`.py`) executed after connection but before any other operation. Requires Arm Debugger v6.1.1 or later.
`"workspaceFolder"`	Current Arm Debugger workspace folder. Default: `"${workspaceFolder}"`

8.2.4 Use the Run and Debug Configuration visual editor for your debug configuration

As an alternative to editing the `launch.json` file of your solution to change the debug configuration options, you can use the **Run and Debug Configuration** visual editor.

8.2.4.1 Debug configuration for a physical target

This section describes how to define an `Attach` or `Launch` configuration with the **Run and Debug Configuration** visual editor.

Procedure

1. To open the editor, either:
    - From the **Explorer**, right-click the `launch.json` file that is stored in the `.vscode` folder of the solution and select **Open Run and Debug Configuration**.
    - From the **Explorer**, right-click the `launch.json` file and select **Open With...**, then select **Run and Debug Configuration** in the drop-down list that displays at the top of the window.
    - If the `launch.json` file is already open in the editor, click **Open Run and Debug Configuration** in the top right-hand corner.
2. You can define several debug configurations for physical targets in the `launch.json` file. In the **Selected Configuration** drop-down list, select **New Configuration**, then select one of the following options:
    - **Attach**: Use an `Attach` configuration if you want to debug a program that is already running.
    - **Launch**: Select a `Launch` configuration to launch your program in debug mode using a physical target.

See Launch versus attach configurations for explanations of the Launch and Attach core debugging modes in Visual Studio Code.

Selecting a configuration adds a new configuration block in the JSON file.

You can also click **Duplicate**, to duplicate the currently selected configuration and modify it.

3. If you are defining an Attach configuration, modify your debug configuration as follows:
   - If the Arm Debugger engine is running on a distant server, indicate the address of the server in the format ws://<host>:<port> (websocket).
   - If the Arm Debugger engine is running on your machine, use <host>:<port> (socket).
   - **Program Files**: One or more programs to use for debugging
     - Click **Add File** to point to a file directly. You can add as many files as you need. Arm Debugger uses the files in the order in which you added them. AXF and ELF files are supported by default. You can also add other file types.
     - Click **Detect File** to add the ${command:arm-debugger.getApplicationFile} command if it is not available. This command detects the latest AXF or ELF file generated.
     - Move your cursor over the name of the command or a file and click the delete icon to remove the selection.
4. If you are defining a Launch configuration, modify your debug configuration as follows:
   - **Probe Type**: In the drop-down list, select a type for the debug probe that you are using or the debug unit on your board.
     - Default value: CMSIS-DAP. If the Arm Debugger extension cannot set the probe type automatically, the default value is CMSIS-DAP.
     - If you have connected a probe or a board over USB to your computer, the Arm Debugger extension sets a probe type based on the serial number of the hardware detected.
   - **Serial Number**: In the drop-down list, select the serial number of the debug probe or debug unit on your board.
     - Default value: auto. With auto, the serial number of the active device in the Arm Device Manager extension is used by default. The "${command:device-manager.getSerialNumber}" command is added in the JSON file for "serialNumber".
     - You can also select the serial number of the active device in the drop-down list, or directly type a serial number.

     Click **Open Arm Device Manager** to check what your active device is.
   - **CMSIS-Pack**: Select the Device Family Pack (DFP) for the target debug probe or board.
     - Default value: auto (CMSIS Solution). The DFP for the active device defined in the *.csolution.yml file of your solution is used by default. The "${command:cmsis-csolution.getTargetPack}" command is added in the JSON file for cmsisPack.
     - auto (Device Manager): The DFP for the active device in the Arm Device Manager extension is used by default. The "${command:device-manager.getDevicePack}" command is added in the JSON file for cmsisPack.

  - You can also select the DFP for the active device in the drop-down list, or directly type the name of a DFP in the format `<vendor>::<pack>@<version>`. For example: `ARM::V2M_MPS3_SSE_300_BSP@1.4.0`.
- **CMSIS-Pack Device Name**: Select the name of the target device (target chip on your board).
  - Default value: `auto (CMSIS Solution)`. The device name is deduced from the information available for the probe or board in the `*.csolution.yml` file of your solution. The `"${command:cmsis-csolution.getDeviceName}"` command is added in the JSON file for `deviceName`.
  - `auto (Device Manager)`: The device name is deduced from the information available for the probe or board in the Arm Device Manager extension. The `"${command:device-manager.getDeviceName}"` command is added in the JSON file for `deviceName`.
  - You can also select the device name in the drop-down list, or directly type the device name. For example: `MPS3_SSE_300`. The device name available in the drop-down list is the one defined in the `*.csolution.yml` file of your solution.
- **Processor Name**: If you are using a device with multiple cores, select the processor to use.
  - Default value: `auto`. With `auto`, the processor name defined in the `*.csolution.yml` file of your solution is used by default. The `"${command:cmsis-csolution.getProcessorName}"` command is added in the JSON file for `"processorName"`.
  - You can also directly type a processor name. For example: `cm4`.
- **Connection Mode**: Select a connection mode. The connection mode controls the operations that are run when the debugger connects to the target debug probe or the board.
  - Default value: `auto`: The debugger decides which connect mode to use based on the connected target device. For ST boards, when `auto` is selected, `underReset` is used. For other boards, `haltOnConnect` is used.
  - `haltOnConnect`: Stops the CPU of the target debug probe or board for a reset before the flash download.
  - `underReset`: Asserts the hardware reset during the connection.
  - `preReset`: Triggers a hardware reset pulse before the connection.
  - `running`: Connects to the CPU without stopping the program execution during the connection.
- **Reset after connect**: Select this option to reset the device after it has acquired control of the processor.
- **Reset Mode**: Select a reset mode. The reset mode controls the reset operations performed by the debugger.
  - `auto` (default): The debugger decides which reset to use based on information from the CMSIS-Pack.
  - `system`: Uses the ResetSystem sequence from the CMSIS-Pack.
  - `hardware`: Uses the ResetHardware sequence from the CMSIS-Pack.

    - `processor`: Uses the ResetProcessor sequence from the CMSIS-Pack.
- **Debug From**: Select a function from which the debugger should start. Default value: `main`. The debugging session starts and the debugger stops at the `main()` function of the program.
- **Program Mode**: Select a program mode. The program mode defines the type of debugging to use: flash debugging `flash`, RAM debugging `ram`, or both `mixed`. Default value: `auto`. In `auto` mode, the debugger decides.

  The main difference between flash and RAM debugging is in the type of memory used for storing and executing the code during a debugging session:
    - Flash debugging: The code is stored and executed from Flash memory. The debugger internally loads debug information but does not load anything to the target.
    - RAM debugging: The debugger loads the code into RAM after connection to the target system. The code is first copied from its storage location (like Flash memory) into RAM before execution.
- **Port Mode**: Select a debug port mode to use. A debug port allows you to communicate with and debug microcontrollers or other embedded systems.
    - Default value: `auto`. The debugger decides which debug port mode to use based on the connected target device.
    - `JTAG`: Use the JTAG debug port mode.
    - `SWD`: Use the SWD debug port mode.
- **Clock Speed**: The maximum clock frequency for the debug communication. The clock frequency is the speed at which data is transferred between the debugger and the target device during debugging operations. The frequency actually used depends on the capabilities of the debug probe and might be reduced to the next supported frequency.
    - Default value: `auto`. With `auto`, the debugger decides which clock frequency to use based on the connected target device.
    - Other possible values: `50MHz`, `33MHz`, `25MHz`, `20MHz`, `10MHz`, `5MHz`, `2MHz`, `1MHz`, `500kHz`, `200kHz`, `100kHz`, `50kHz`, `20kHz`, `10kHz`, `5kHz`.
- **Program Files**: One or more programs to use for debugging
    - Default value: The `${command:arm-debugger.getApplicationFile}` command is added in the JSON file for `"program"` when you add a new configuration block. This command detects the latest AXF or ELF file generated. Click **Detect File** to add the `${command:arm-debugger.getApplicationFile}` command if it is not available.
    - You can also use the **Add File** button to point to a file directly. You can add as many files as you need. Arm Debugger uses the files in the order in which you added them. AXF and ELF files are supported by default. You can also add other file types.

5. If **Auto Save** is not enabled (**File** > **Auto Save**), save your changes.
   The `launch.json` file is updated.

8.2.4.2 Debug configuration for a virtual target (Fixed Virtual Platforms)

This section describes how to define a `Launch FVP` configuration with the **Run and Debug Configuration** visual editor.

Before you begin

To debug a virtual target using Fixed Virtual Platforms (FVPs) models, you must install models on your machine.

Check that the `vcpkg-configuration.json` file for your project contains `"arm:models/arm/avh-fvp"` in the `"requires":` section.

You can add FVPs with the **Arm Tools** visual editor (**Arm Virtual Hardware for Cortex®-M based on Fast Models** option) or by editing the `vcpkg-configuration.json` file directly.

Note

FVPs are natively available on Windows and Linux only. If you are on Mac, you can use Docker to run FVPs in a Linux container. Follow this Arm Developer Learning Path to install Docker and clone the https://github.com/Arm-Examples/FVPs-on-Mac repository.

Procedure

1. To open the editor, either:
   - From the **Explorer**, right-click the `launch.json` file that is stored in the `.vscode` folder of the solution and select **Open Run and Debug Configuration**.
   - From the **Explorer**, right-click the `launch.json` file and select **Open With...**, then select **Run and Debug Configuration** in the drop-down list that displays at the top of the window.
   - If the `launch.json` file is already open in the editor, click **Open Run and Debug Configuration** in the top right-hand corner.
2. To work with a virtual target, you can define a `Launch FVP` configuration in the `launch.json` file. In the **Selected Configuration** drop-down list, select **New Configuration**, then select **Launch FVP**.
   Selecting a configuration adds a new configuration block in the JSON file.

   You can also click **Duplicate**, to duplicate the currently selected configuration and modify it.
3. Modify your debug configuration as follows:
   - **Configuration Database Entry**: The configuration database is where Arm Debugger stores information about the processors, devices, and boards it can connect to. Select the FVP that you want to use (for example, **MPS2_Cortex_M4**), then select a processor (for example, **Cortex-M4**). The list of FVPs available depends on the `avh-fvp` version specified in the `vcpkg-configuration.json` file for your project.
   - **FVP Parameters**: For more advanced configuration settings, you can generate a list of FVP parameters and modify the arguments listed in the file. Click **Generate File** to generate an `fvp_config.txt` file. Click the pen icon to open the FVP Parameters editor and modify the file. If you already have a file available, click **Select File** to select it.

- **Debug From**: Select a function from which the debugger should start. Default value: `main`. The debugging session starts and the debugger stops at the `main()` function of the program.
- **Program Files**: One or more programs to use for debugging
  - Default value: The `${command:arm-debugger.getApplicationFile}` command is added in the JSON file for `"program"` when you add a new configuration block. This command detects the latest AXF or ELF file generated.
  - Click **Add File** to point to a file directly. You can add as many files as you need. Arm Debugger uses the files in the order in which you added them. AXF and ELF files are supported by default. You can also add other file types.
  - Click **Detect File** to add the `${command:arm-debugger.getApplicationFile}` command if it is not available.
  - Move your cursor over the name of the command or a file and click the delete icon  to remove the selection.

4. If **Auto Save** is not enabled (**File** > **Auto Save**), save your changes.
   The `launch.json` file is updated.

8.2.5 Start an Arm Debugger session

Start a debug session.

Before you begin

When you have several solutions grouped in a single folder on your machine, Visual Studio Code does not take into account the `tasks.json` and `launch.json` files that you might have created for each solution. Instead, Visual Studio Code generates new JSON files at the root of the workspace in a `.vscode` folder and ignores the other JSON files.

As a result, you might have issues running or debugging a project.

As a workaround, open one solution first, then add other solutions to your workspace with the **File** > **Add Folder to Workspace** option.

8.2.5.1 Start a debug session with a physical target

To start a debug session with a physical target, use the following procedure.

Procedure

1. Check that your device is connected to your computer.
2. To start a debug session, go to the **Run and Debug** view  and select the `Arm Debugger` configuration in the list 

. Click **Start Debugging**.

Alternatively, if you have installed the Keil Studio Pack, go to the **CMSIS** view, open the Build Context view, and check which debug configuration is selected. Then, click **Debug** in the **Solution outline** header.

Debug icons are available at the solution level or at the project level depending on what you selected in the **Build Context** view for the debug configuration. See Set a context for your solution for more details.

3. If you are using a device with multiple cores and you did not specify a `"processorName"` in the `launch.json` file, and you do not have the CMSIS Solution extension installed, then you must select the appropriate processor for your project in the **Select a processor** drop-down list that displays at the top of the window.
The **Run and Debug** view displays and the debug session starts. The debugger stops at the `main()` function of the program.
4. Check the **Debug Console** tab to see the debugging output.
If the Arm Debugger engine cannot be found on your machine, an **Arm Debugger not found** dialog box displays.

   Select one of these options:

   - Click **Install Arm Debugger** to add it to your environment. The `vcpkg-configuration.json` file is updated. Check the Arm tools installed in the status bar Arm Tools: 8.
   - Click **Configure Path** to indicate the path to the Arm Debugger engine in the settings.

8.2.5.2 Start a debug session with a virtual target

Start a debug session with a virtual target.

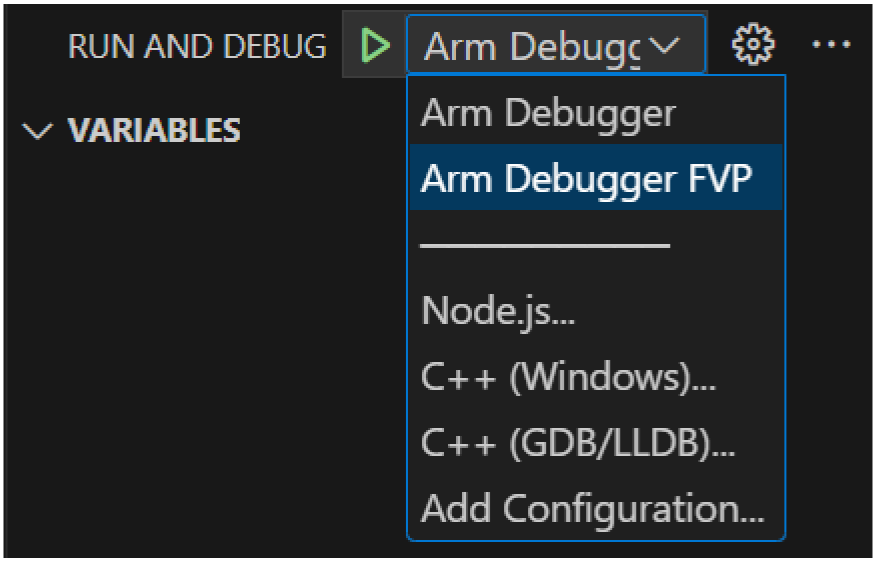
Before you begin

Fixed Virtual Platforms (FVPs) are natively available on Windows and Linux only. If you are on Mac, follow this Arm Developer Learning Path to install Docker and clone the https://github.com/Arm-Examples/FVPs-on-Mac repository.

Procedure

1. Go to the **Device Manager** and select the FVP that you want to use. For example, **MPS2 Cortex M4**.
2. To start a debug session, go to the **Run and Debug** view and select the `Arm Debugger FVP` configuration in the list. Click **Start Debugging**.

Figure 8-1: FVP configuration



Alternatively, if you have installed the Keil Studio Pack, go to the **CMSIS** view, open the Build Context view, and check which debug configuration is selected. Then, click **Debug** in the **Solution outline** header.

Debug icons are available at the solution level or at the project level depending on what you selected in the **Build Context** view for the debug configuration. See Set a context for your solution for more details.

The **Run and Debug** view displays and the debug session starts. The debugger stops at the `main()` function of the program.

3. Check the **Debug Console** tab to see the debugging output.
   If the Arm Debugger engine cannot be found on your machine, an **Arm Debugger not found** dialog box displays.

   Select one of these options:

   - Click **Install Arm Debugger** to add it to your environment. The `vcpkg-configuration.json` file is updated. Check the Arm tools installed in the status bar Arm Tools: 8.
   - Click **Configure Path** to indicate the path to the Arm Debugger engine in the settings.

8.2.6 Set breakpoints

Breakpoints are useful when you know which part of your code you want to examine. To look at values of variables, or to check if a block of code is getting executed, you can set one or more breakpoints to suspend your running code.

See the Visual Studio Code documentation for more details.

Note

With the current version of the Arm Debugger extension, you cannot set breakpoints in assembly files by default. To be able to set breakpoints in assembly files, go to the settings and select **Allow Breakpoints Everywhere**.

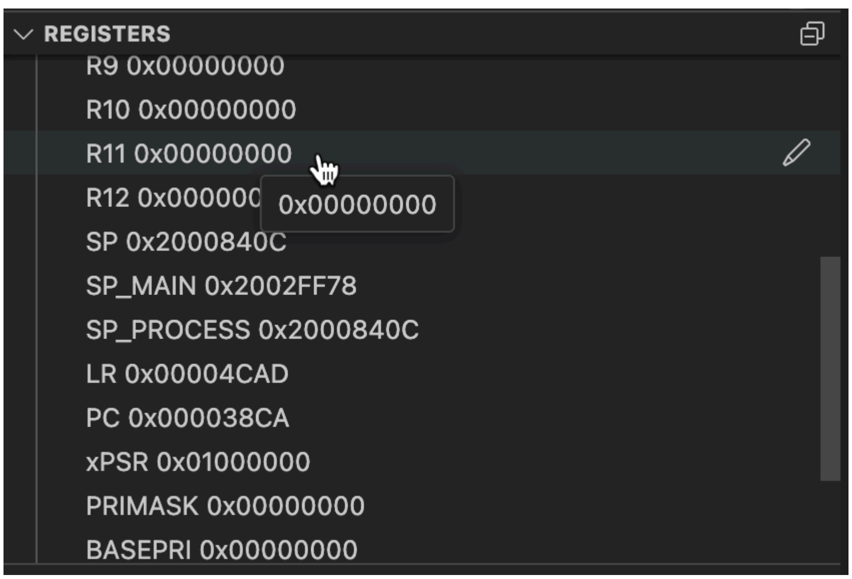
8.2.7 Inspect registers

The **Registers** view displays register contents for the detected processor. Start a debug session as explained in Start an Arm Debugger session to display the **Registers** view in the **Run and Debug** view.

The **Registers** view organizes registers into groups. These groups vary according to the processor type you are using and the system you are debugging. During debugging, register values change as your code executes.

Here is an example of what you can see in the **Registers** view for a Cortex-M4 processor:

Figure 8-2: Registers view for a Cortex-M4 processor



The **Registers** view can include:

- Processor core registers: In Arm processors, each processor core has a set of general-purpose registers that are used for temporary data storage and manipulation during program execution. These registers are used by the processor for various operations, including arithmetic, logical, and data movement instructions. Additionally, Arm processors may also have other specific registers, such as Program Counter (PC) and Stack Pointer (SP), which are essential for managing program flow and maintaining the stack. These registers collectively form the register file of the processor core, providing a fast and efficient means for the processor to store and retrieve data during computation.

- System registers: In Arm processors, system registers are special-purpose registers that control and configure various aspects of a processor's behavior. These registers are part of the Arm architecture and play a crucial role in managing system-level functionality. System registers help control the processor's operating mode, interrupt handling, and other system-related features.
- Floating-Point Unit (FPU) registers: In Arm processors, the FPU is responsible for handling floating-point arithmetic operations. The FPU has its own set of registers distinct from the general-purpose registers. These registers are used to store floating-point numbers and perform operations like addition, subtraction, multiplication, and division on them.

8.2.7.1 Edit registers

Edit registers during a debug session.

Procedure

1. Start a debug session as explained in Start an Arm Debugger session.
2. Click **Pause** to pause the debug session.
   The **Registers** view displays register values that you can edit.
3. Move your cursor over the register values and click the pen icon for the value that you want to update.
4. Enter a value or an expression in the field that opens at the top of the window and press **Enter**.
   If you enter an expression, the result of the expression is written to the registers. For example: Use `$SP+0x20` to add `0x20` to the content of the `SP` register. See the Arm Debugger Command Reference guide for more details on expressions.

   Modified values are highlighted in the **Registers** view.

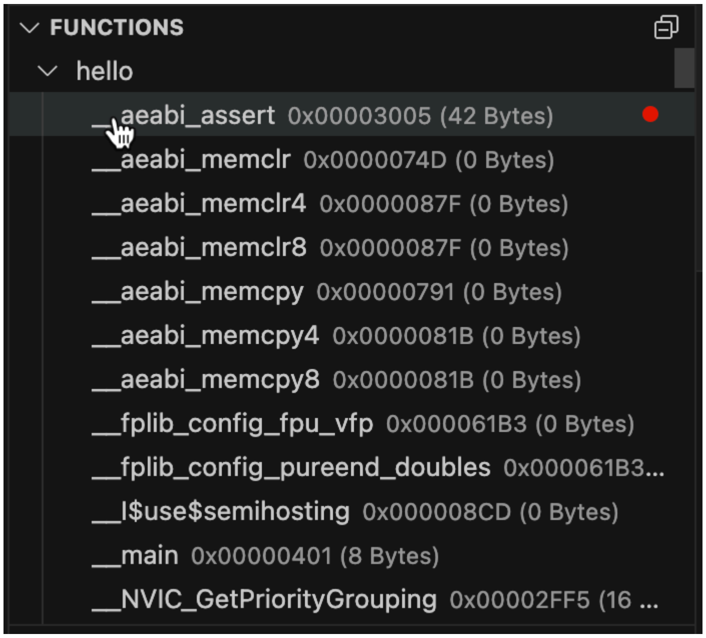
8.2.8 Inspect functions

The **Functions** view displays the main functions in your code, library functions, and user-defined functions. Start a debug session as explained in Start an Arm Debugger session to display the **Functions** view in the **Run and Debug** view.

For each function, the **Functions** view shows the following details:

- The name of the function
- The address where the function is stored
- The size of the function in bytes

Figure 8-3: Functions view



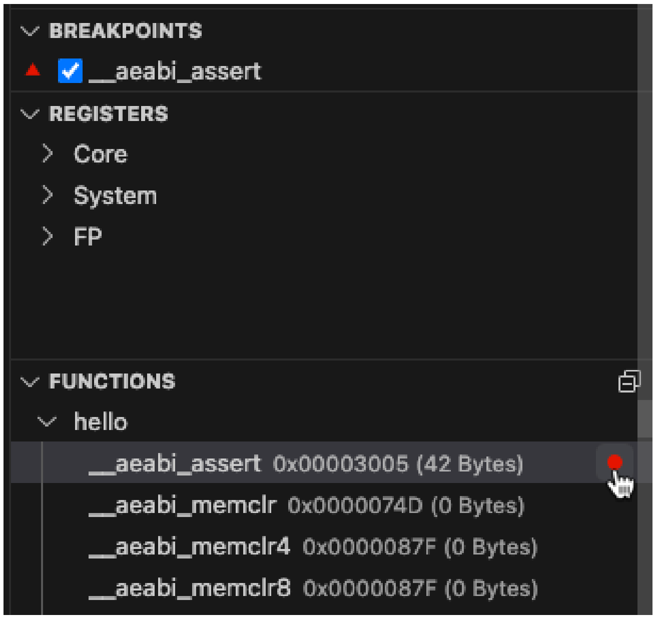
You can add function breakpoints to break execution when a function is called. Breaking execution is useful when you know the function name but not its location.

To add a function breakpoint:

In the **Functions** view, move your cursor over the function for which you want to add a breakpoint and either:

- Click the red dot that displays on the right side of the function name
- Right-click the function and select **Set function breakpoint**.

Figure 8-4: Add a function breakpoint



The function breakpoint displays in the **Breakpoints** view.

From the **Breakpoints** view, you can:

- Remove all breakpoints with  or remove specific breakpoints

Figure 8-5: Remove specific breakpoints

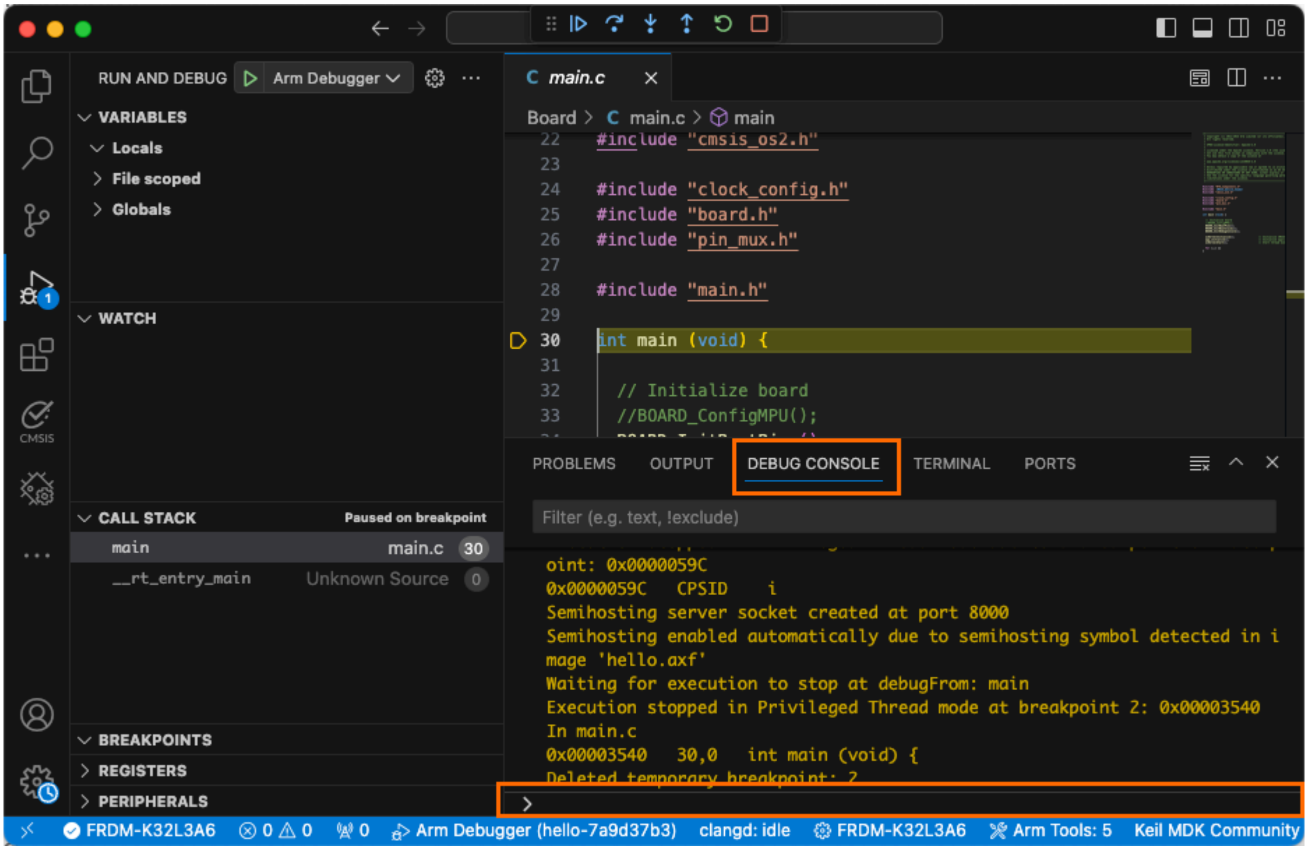
- Add function breakpoints with 
- Activate or deactivate all breakpoints with 

8.2.9 Use the Debug Console

The **Debug Console** shows the debugging output of your project. It displays messages, errors, warnings, and other output generated during a debugging session.

The **Debug Console** automatically displays when you start a debug session. You can also go to **View** > **Debug Console** to display it.

Figure 8-6: Debug Console



8.2.9.1 Run Arm Debugger commands

You can run Arm Debugger commands directly from the **Debug Console**. Type `help` followed by the name of a command in the **Debug Console** prompt to display information on the command and how to use it.

For example, `help step` displays:

```
step

step

    Steps through an application at the source level stopping on the first
    instruction of each source line including stepping into all function calls. You
    must compile your code with debug information to use this command successfully.

    You can modify the behavior of this command with the set step-mode command.

Syntax

    step [<count>]

    Where:

    <count>
        Specifies the number of source lines to execute.
```

```
        Note:
            Execution stops immediately if a breakpoint is reached, even if fewer
 than
            <count> source lines are executed.

Examples

    step                                    # Execute one source line.
    step 5                                  # Execute five source lines.
```

Example:

1. Type `break main.c:10` in the **Debug Console** prompt and press **Enter** to add a breakpoint on line 10 of your `main.c` file.
2. Type `continue` in the prompt and press **Enter** to continue the debugging session. The debugger runs to the first breakpoint it encounters and stops.
3. Type `step` to go to the next line.

All the Arm Debugger commands are also documented in Arm Debugger commands listed in alphabetical order in the Arm Debugger Command Reference.

Type `help` followed by the name of a group in the prompt to display all the commands that are part of that group.

For example, `help group_log` displays:

```
group_log

log

    List of all the Arm Debugger commands that enable you to control runtime
    messages from the debugger.

    log config
        Specifies the type of logging configuration to output runtime messages from
        the debugger.

    log file
        Specifies an output file to receive runtime messages from the debugger.

    Enter help followed by a command name for more information on a specific
    command.
```

The groups are also documented in Arm Debugger commands listed in groups in the Arm Debugger Command Reference.

8.2.9.2 Use expressions

You can evaluate and resolve expressions with the **Debug Console**.

Use `$expr:<expression>` in the **Debug Console** prompt, where `<expression>` can include program symbols or register symbols, and arithmetic or logical operations.

Example with the stack pointer register symbol `$SP` and an arithmetic operation:

1. Type `$expr:$SP` in the **Debug Console** prompt and press **Enter**.

   This expression returns the current value of the stack pointer, for example `537133040`.
2. Type `$expr:$SP+0x20` and press **Enter**.

   This expression returns the current value of the stack pointer + `32`, so `537133072`.

Example with the global variable `SystemCoreClock` and a logical operation:

1. Type `$expr:SystemCoreClock`.

   This expression returns the current value of the global variable `25000000`.
2. Type `$expr:SystemCoreClock == 25000000` and press **Enter**.

   This expression returns `1`, because the result is true.
3. Type `$expr:SystemCoreClock != 25000000` and press **Enter**.

   This returns `0`, because the result is false.

8.2.10 Scope resolution operator

You can use the scope resolution operator (`::`) to access variables and functions in images, files, namespaces, or classes.

The scope resolution operator can be useful if you have to debug a project with multiple AXF or ELF files (for example, a TrustZone example which consists of at least two ELF files). You can explicitly point at a symbol in a specific file (for example, the `main` function) using symbol expressions. See the Arm Debugger Command Reference guide for more details on the scope resolution operator.

For example, to select a function from which the debugger should start using the **Run and Debug Configuration** visual editor, you can specify the following expression in the **Debug From** field:

```
"hello.axf"::main
```

You must put the expression between quotes.

The following line is added in the `launch.json` file:

```
"debugFrom": "\"hello.axf\"::main"
```

Backslashes are used to escape quotes.

You can also use absolute or relative file paths in expressions.

The scope resolution operator is also useful with watch expressions, the **Debug Console**, or function breakpoints.

8.2.11 Next steps

Look at the Visual Studio Code documentation to learn more about the debugging features available in Visual Studio Code.

8.3 Work with scripts

You can use scripts to customize your debugging workflows and automate tasks.

The Arm Debugger extension supports scripting using languages such as standard Python (CPython) and Jython. Jython is a Java implementation of Python and is an ideal choice for larger or more complex scripts.

To call scripts with a `.py` extension in the `tasks.json` file or `launch.json` file of your projects, use the `"targetInitScript"` and `"debugInitScript"` configuration options.

To work with Python and Jython scripts, you must install a supported version of Python on your machine. The Arm Debugger extension supports Python 2.7.2 and above, but not version 3. See the explanations provided in the Visual Studio Code documentation to install a Python interpreter.

Note

On macOS:

The system installation of Python is not supported. It is recommended that you use a package management system like Homebrew.

It is also recommended that you install the Microsoft Python extension in Visual Studio Code to benefit from the rich support for the Python language that it provides. The Microsoft Python extension is installed with Microsoft Pylance, which offers support for IntelliSense, and Microsoft Python Debugger.

After you have installed a version of Python and the Python extension, select the Python version with the **Python: Select Interpreter** command from the Command Palette as explained in the Visual Studio Code documentation. Alternatively, manually specify an interpreter.

9. Activate your license to use Arm tools

You must activate a license to be able to use tools such as Arm® Compiler, Arm Debugger, or Fixed Virtual Platforms (FVPs) in your toolchain.

If you try to use a licensed tool without a license, a **No Arm License** status displays in the status bar and a pop-up message displays.

Errors also appear in the `vcpkg-configuration.json` file and in the **Problems** tab (**View** > **Problems**).

1. Click **Manage Arm license** in the pop-up that displays in the bottom right-hand corner.
2. Select one of the following options in the drop-down list at the top of the window:
   - **Activate Arm Keil MDK Community Edition**: Select this option to switch to the Keil® MDK Community Edition license. You can use this license only for non-commercial projects.
   - **Activate or manage Arm licenses**: Select this option to switch to a commercial license such as Keil MDK Professional Edition or a Keil MDK Essential Edition. This option opens an **Arm License Management Utility** window where you can provide a product activation code or use a license server to activate your license.

Note

To have access to the **Arm License Management Utility** window and manage your license, you can also use the **Environment: Manage tool licenses** command from the Command Palette.

For further details on license activation, see Activate your product using an activation code and Activate your product using a license server in the User-based Licensing User Guide.

The Backwards compatibility topic also explains how you can license older versions of MDK (MDK 5.36 and earlier), PK51, PK166, and DK251 using a product license that includes Keil MDK Professional.

9.1 Troubleshoot expired or cache-expired licenses

If you try to use a licensed tool with a license that is expired or cache-expired, a warning displays in the status bar and a pop-up message displays in the bottom right-hand corner.

Note

Cache-expired licenses happen when your local license could not be renewed, either because of network issues, lack of space on your device, or issues with your permissions.

---

1. Click **Manage Arm license** in the pop-up.
2. Depending on your license, one of the following options displays in the drop-down list at the top of the window:

- If your license has expired, a **Get help for expired license** option displays. Select this option to view information on the steps that you need to take.
- If your license is cache-expired, a **Get help for cache-expired license** option displays. Select this option to view information on the steps that you need to take.

10. Use CMSIS-Toolbox from the command line

CMSIS-Toolbox is a set of command-line tools that are integrated into the Keil® Studio extensions. You can also use them as standalone tools from the command line.

If you used an official example from keil.arm.com and installed the Keil Studio Pack as recommended, then CMSIS-Toolbox is already available on your machine as explained in Get started with an example project.

The main tools that CMSIS-Toolbox provides and that you can use with the command line are:

- `cpackget`: Pack Manager. Used to install and manage CMSIS-Packs in your development environment.
- `cbuild`: Build invocation. Used to orchestrate the build process that translates a project to an executable binary image. `cbuild` invokes the different tools (`csolution`, `cpackget`, and `cbuildgen`) and launches the CMake compilation process.
- `csolution`: Project Manager. Used to create build information for embedded applications that consist of one or more related projects.

The Build Tools page describes how to use these tools with the command line.

10.1 Add CMSIS-Toolbox to the system PATH

The Environment Manager extension installs CMSIS-Toolbox and adds the tools into the Visual Studio Code system PATH.

If you install CMSIS-Toolbox without using the Environment Manager extension and vcpkg, add the installation path to the system PATH, or use the **Cmsis-csolution: Cmsis Toolbox Path** setting to add the path.

10.2 Support for packs

CMSIS-Packs (also often referred to as software packs) contain everything that you need to work with specific microcontroller families or development boards.

You can work with different types of packs:

- Public packs. These are packs that Arm or silicon and software vendors created and that are publicly available. Public packs are available from the CMSIS-Packs page on keil.arm.com.
- Private packs. These are packs that you have created but not shared yet, or packs that others shared with you privately. These can be local packs available on your system or remote packs available on the web.

This section gives you an overview on how to manage the different types of packs.

Note

The Open-CMSIS-Pack documentation describes the different ways of adding or removing packs from the command line in detail. See Adding packs and Removing packs.

10.2.1 Add public packs

You can use the functionality available in the CMSIS Solution extension to install public packs. See Install CMSIS-Packs for more details.

Alternatively, you can use the `cpackget add` command from the **Terminal** to install the latest published version of public packs listed in the package index of a vendor. A package index file lists all the CMSIS-Packs hosted and maintained by a vendor. See the Open-CMSIS-Pack documentation for more information on package index files.

Tip

Explore the available CMSIS-Packs on keil.arm.com/packs and use the snippets available to update your `csolution.yml` file and install packs with `cpackget add`.

For example, the following command installs the latest public version of a public pack:

```
cpackget pack add Vendor::PackName
```

Where:

- `Vendor`: Is the name of the vendor who created the CMSIS-Pack
- `PackName`: Is the name of the CMSIS-Pack

After running `cpackget add`, reload Visual Studio Code to update the data that displays in the user interface.

10.2.2 Add private local packs

To work with a CMSIS-Pack that you created locally, use the `cpackget add` command from the **Terminal** and reload Visual Studio Code so that the CMSIS csolution extension knows about the registered pack. Components from the pack appear in the **Software Components** view, and the file validation takes the new pack into account.

For example, the following command registers a local pack using a PDSC (pack description) file:

```
cpackget add /path/to/Vendor.PackName.pdsc
```

Where:

- `Vendor`: Is the name of the vendor who created the CMSIS-Pack
- `PackName`: Is the name of the CMSIS-Pack

PDSC files contain information about the content of packs.

After running `cpackget add` to add packs to the pack root folder, reload Visual Studio Code to update the data that displays in the user interface.

If you cannot see the components from the pack or packs that you have just added in the **Software Components** view, check the **Cmsis-csolution: Pack Cache Path** setting and the `CMSIS_PACK_ROOT` environment variable.

10.2.3 Add private remote packs

To install a remote pack available on the web, use the `cpackget add` command and the URL of the pack.

For example, the following command installs a pack version that can be downloaded from the web:

```
cpackget add https://vendor.com/example/Vendor.PackName.x.y.z.pack
```

Where:

- `Vendor`: Is the name of the vendor who created the CMSIS-Pack
- `PackName`: Is the name of the CMSIS-Pack
- `x.y.z`: Is the specific version of the pack that you want to install

After running `cpackget add`, reload Visual Studio Code to update the data that displays in the user interface.

10.2.4 Remove packs

To remove packs from your system, use `cpackget rm`.

For example, the following command removes a specific pack version:

```
cpackget rm Vendor.PackName.x.y.z
```

Where:

- `Vendor`: Is the name of the vendor who created the CMSIS-Pack
- `PackName`: Is the name of the CMSIS-Pack
- `x.y.z`: Is the specific version of the pack that you want to remove

After running `cpackget rm`, reload Visual Studio Code to update the data that displays in the user interface.

11. Known issues and troubleshooting

Describes known issues with the Keil® Studio extensions and how to troubleshoot some common issues.

11.1 Known issues

Here are the known issues.

Arm CMSIS Solution extension

The CMSIS Solution extension has the following known issues:

- No support for cdefaults.yml. The **Software Components** view and validation do not use the compiler set in the cdefaults file.

11.2 Troubleshooting

Provides solutions to some common issues you might experience when you use the extensions.

11.2.1 Build fails to find CMSIS-Toolbox causes an ENOENT error

The solution build fails with an ENOENT error because the extension cannot find the CMSIS-Toolbox.

Solution

Follow the instructions in the pop-up message.

If the Environment Manager is installed, but the environment does not contain CMSIS-Toolbox:

- Add CMSIS-Toolbox to the `vcpkg-configuration.json` file (**Add to Vcpkg** option). This installs CMSIS-Toolbox with the Environment Manager.
- Alternatively, install CMSIS-Toolbox manually and add it to the PATH, or configure the path in the settings (**Open Installation Documentation** option).

If the Environment Manager is not installed:

- Install the Environment Manager from the **Extensions** view (**Install Environment Manager** option) and create a `vcpkg-configuration.json` file. Click **Arm Tools** in the status bar, then select **Add Arm tools Configuration To Workspace** to open the visual editor and select tools. This creates a `vcpkg-configuration.json` file you can save for your project.
- Alternatively, install CMSIS-Toolbox manually and add it to the PATH, or configure the path in the settings (**Open Installation Documentation** option).

The CMSIS-Toolbox documentation describes how to install CMSIS-Toolbox manually.

11.2.2 Download and installation of vcpkg artifacts fails on Windows

With the Arm Environment Manager extension, downloading and installing vcpkg artifacts fails on Windows due to the length of resulting path names in the default installation folder.

Solution

Enable long path support in your Windows settings, as described here: Enable Long Paths in Windows 10, Version 1607, and Later.

11.2.3 Build fails to find toolchain

With the CMSIS Solution extension, errors such as `ld: unknown option: --cpu=Cortex-M4` appear in the build output. In this example, CMSIS-Toolbox is trying to use the system linker rather than Arm® Compiler's armlink.

Solution

1. If you have installed the CMSIS Solution extension separately rather than by using the Keil Studio Pack, make sure that you follow the instructions for installing and setting up CMSIS-Toolbox. In particular, make sure that the `CMSIS_COMPILER_ROOT` environment variable is set correctly. Alternatively, you can install the Keil Studio Pack to benefit from an automated setup with Microsoft vcpkg.
2. Clean the solution. In particular, delete the `out` and `tmp` directories.
3. Run the build again.

11.2.4 Connected development board or debug probe not found

You have connected your development board or debug probe, but the Device Manager extension cannot detect the hardware.

Solution

- Run **Device Manager** (Windows), **System Information** (Mac), or a Linux system utility tool like **hardinfo** (Linux), and then check for warnings beside your hardware. Warnings can indicate that hardware drivers are not installed. If necessary, obtain and install the appropriate drivers for your hardware.
- On Windows: ST development boards and probes require extra drivers. You can download them from the ST site.
- On Windows: Check if you have an Mbed™ serial port driver installed on your machine. The Mbed serial port driver is required with Windows 7 only. Serial ports work out of the box with Windows 8.1 or newer. The Mbed serial port driver breaks native Windows functionality for updating drivers as it claims all the boards with a DAPLink firmware by default. Arm recommends that you uninstall the driver if you do not need it. Alternatively, you can disable it.

  You can either:

- Uninstall the Mbed serial port driver (recommended): Open a command prompt as an administrator. Find and delete the `mbedserial_x64.inf` and `mbedcomposite_x64.inf` drivers.

  ```
  pnputil /enum-drivers
  pnputil /delete-driver {oemnumber.inf} /force
  ```

  Then, connect your hardware using a USB cable and open the Windows Device Manager. In `Ports (COM & LPT)` and `Universal Serial Bus controllers`, find the `mbed` entries and uninstall both by right-clicking them. Finally, disconnect and reconnect your hardware.
- Disable the Mbed serial port driver: Open the Windows Device Manager. In `Ports (COM & LPT)`, find the Mbed Serial Port. Right-click it and select **Properties**. Select the **Driver** tab and click **Update Driver**. Click **Browse my computer for drivers** and then click **Let me pick from a list of available drivers on my computer**. Select `USB Serial Device` instead of `mbed Serial Port`.

- On Linux: udev rules grant permission to access USB boards and devices. You must install udev rules to be able to build a project and run it on your hardware or debug a project.

  Clone the pyOCD repository, then copy the rules files which are available in the udev folder to `/etc/udev/rules.d/` as explained in the README.md file. Follow the instructions in the `README` file.

  After installing the udev rules, your connected hardware is detectable in the Device Manager extension. You might still encounter a permission issue when accessing the serial output. If this is the case, run `sudo adduser "$USER" dialout`, and then restart your machine.
- Check that the firmware version of your board or debug probe is supported and update the firmware to the latest version. See Out-of-date firmware for more details.
- Your board or device might be claimed by other processes or tools (for example, if you are trying to access a board or device with several instances of Visual Studio Code, or with Visual Studio Code and another IDE).
- Activate the **Manage All Devices** setting. This setting allows you to select any USB hardware connected to your computer. By default, the Device Manager extension gives you access only to hardware from known vendors.
  1. Open the settings:
     - On Windows or Linux, go to: **File** > **Preferences** > **Settings**.
     - On macOS, go to: **Code** > **Settings** > **Settings**.
  2. Find the **Device-manager: Manage All Devices** setting and select its checkbox.

11.2.5 Out-of-date firmware

You have connected your development board or debug probe and a pop-up message appears mentioning that the firmware is out of date.

Solution

Update the firmware of the board or debug probe to the latest version:

- DAPLink. If you cannot find your board or probe on `daplink.io`, then check the website of the manufacturer for your hardware.
- ST-LINK.
- For other WebUSB-enabled CMSIS-DAP firmware updates, please contact your board or debug probe vendor.

Note

If you are using an FRDM-KL25Z board and the standard DAPLink firmware update procedure does not work, follow this procedure (requires Windows 7 or Windows XP).

For more information on firmware updates, see also the Debug Probe Firmware Update Information Application Note.

12. Submit feedback

If you have suggestions or if you have discovered an issue with any of the Keil® Studio extensions, please report them to us. Go to the keil.arm.com support page and use the links provided in the **Keil Studio for VS Code** category.