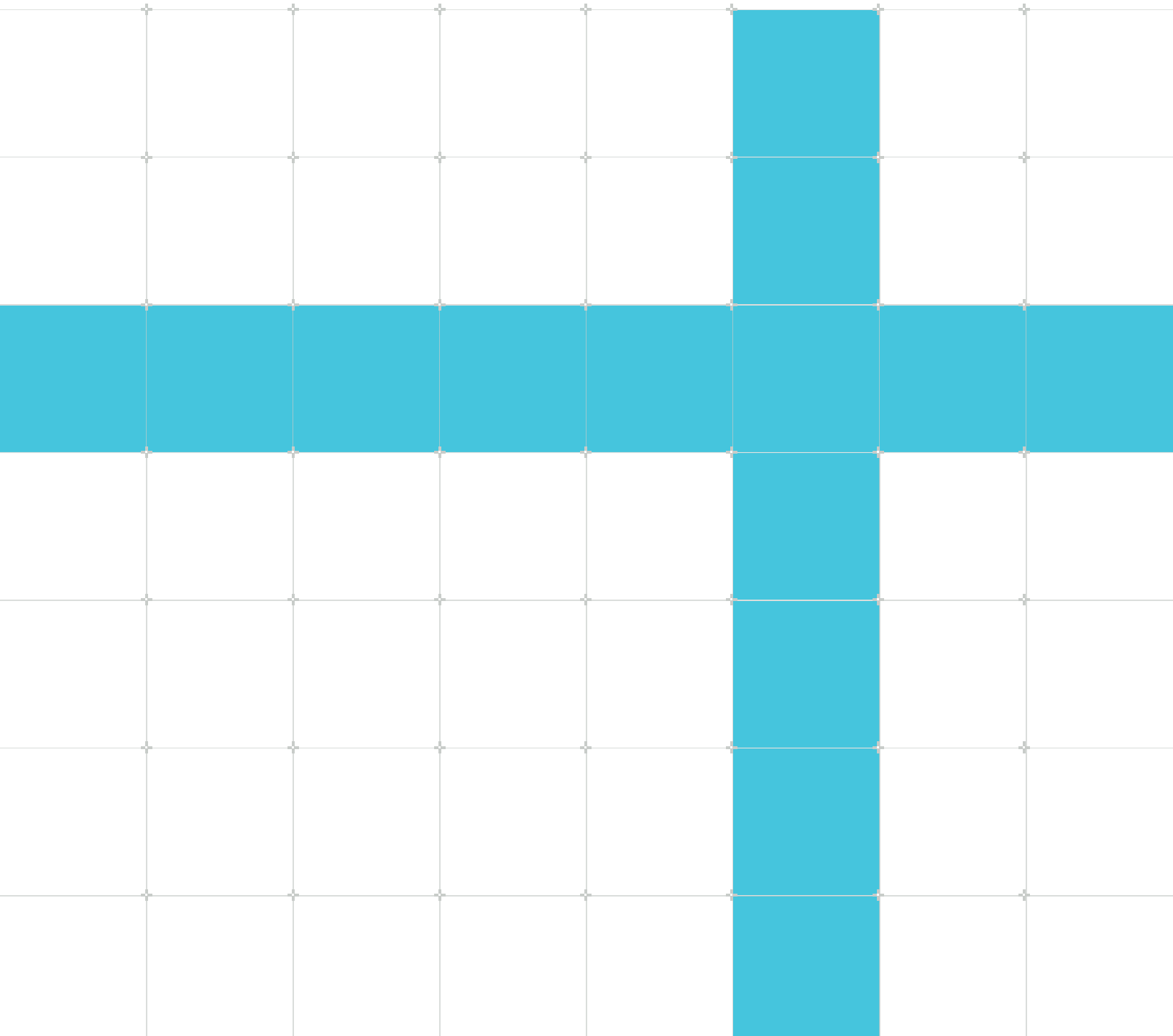
Arm® Keil® Studio Visual Studio Code Extensions

User Guide

Issue 18
108029_0000_18_en



Arm® Keil® Studio Visual Studio Code Extensions User Guide

This document (108029_0000_18_en) was issued on 2024-11-06. There might be a later issue at https://developer.arm.com/documentation/108029

See also: Proprietary notice | Product and document information | Useful resources

Start reading

If you prefer, you can skip to the start of the content.

Intended audience

This book is written for all developers who are involved in the development of embedded, IoT, and Machine Learning software for Cortex-M devices.

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.

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.

Contents

1. 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.

1.1 Arm Keil Studio Pack

The Arm® Keil® Studio Pack is a 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, also known as 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 through external debug probes like Arm's ULINK™ family of debug probes, or through on-board low-cost debugging like 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 the 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. The Memory Inspector 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

- This guide does not describe the Arm Virtual Hardware extension, which is in development.
- This guide does not describe the Memory Inspector and the Peripheral Inspector, which are third-party open-source extensions.

You can also install and use the extensions contained in the pack individually. However, we recommend 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.

2. 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, also known as 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 and toolchains without additional overhead. This use case requires familiarity with Visual Studio Code to configure tasks. See the individual extensions for more details.

3. 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, CMSIS Solution extension converts it automatically to csolution format and installs any missing packs. CMSIS Solution extension also initializes 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, 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 include 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.

3.1 Import a solution example

Import a solution example 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 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.
    Missing CMSIS-Packs are installed automatically.

You must activate a license to use tools like 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.

11. Click **Explorer**  .
A `vcpkg-configuration.json` is available. The file records the vcpkg artifacts that you need to work with your projects. Microsoft vcpkg and the Environment Manager extension take care of the setup automatically. 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.

3.2 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, update Arm Compiler 5 projects to Arm Compiler 6 in Keil µVision, then convert the projects to solutions in Visual Studio Code. For more information, 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.

3.3 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.

3.3.1 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

3.3.2 clangd

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


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 you install clangd, you do not need any extra setup. The CMSIS Solution extension generates a `compile_commands.json` file for each project in a solution whenever a csolution file changes. It also generates the file when you change the context of a solution, that is, the **Target Type** and **Build Type** types. CMSIS Solution extension keeps a `.clangd` file 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, CMSIS Solution extension adds scoped compiler flags to define certain macros to both your `.clangd` project configuration file and your `config.yaml` global user configuration file. See the clangd documentation for more details.

To turn off the automatic generation of the `.clangd` file and `compile_commands.json` file, and 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 Solution: Generate Clang Setup** setting and clear its checkbox.

3.4 Build the example project

Check that your example project builds. You can build your project from the **Explorer** using **Build solution**, 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 solution**.

These options are also available in the right-click menu:

- **Clean all out and tmp directories**: cleans the output and tmp directories for the active solution
- **Rebuild solution**: cleans the output directories before building the cproject

- From the **Solution outline**:

  a. Click **CMSIS**  in the Activity Bar.

  The **Solution outline** opens. A **Build solution** icon is available in the **Solution outline** header.

  b. Click **Build solution**  .

  The **Clean all out and tmp directories** and **Rebuild solution** options are also available with **Views and More Actions**  .

  You can configure a build task in a `tasks.json` file to customize the behavior of the build button. All the examples on keil.arm.com include a `tasks.json` file. See Configure a build task for more details.

- From the Command Palette: You can also trigger **Build solution**, **Clean all out and tmp directories**, and **Rebuild solution** with the **CMSIS: Build**, **CMSIS: Clean all out and tmp directories**, and **CMSIS: Rebuild solution** 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 and causes an ENOENT error for more information.

---

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

3.5 Choose a context for your solution

A context is the combination of a target type, known as a build target, and a build type, known as a build configuration. This context is specific to a particular project in your solution.

Read Set a context for your solution for more details.

3.6 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.

3.7 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.

3.8 Connect your board

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

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 Device Manager detects the board and displays a pop-up message.
3. Click **OK** in the pop-up message to use the hardware.

   You can now use your board to run and debug a project.

3.9 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. A **Run** icon is available in the **Solution outline** header.
2. Click **Run**.
3. If you are using a multicore device, select a processor from the drop-down list at the top of the window.

Note

You do not need to select a processor if you specified a `"processorName"` in the `launch.json` file and you installed the CMSIS Solution extension.

The project runs on the board.

4. Check the **Terminal** tab.

3.10 Start a debug session

Start a debug session.

Procedure

1. Click **CMSIS** in the Activity Bar.
   The **Solution outline** opens. A **Debug** icon is available in the **Solution outline** header.
2. Click **Debug**.
3. If you are using a multicore device, select a processor from the drop-down list at the top of the window.

Note

You do not need to select a processor if you specified a `"processorName"` in the `launch.json` file and you installed the CMSIS Solution extension.

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

4. To see the debugging output, check the **Debug Console** tab.

Next steps

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

4. Arm Environment Manager extension

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

Your project artifacts are stored in the `vcpkg-configuration.json` file in the project source code, so the same tools are available to everyone using the project.

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

Note

Alternatively, you can install the artifacts for your project by manually downloading and installing the CMSIS-Toolbox and other required tools. 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).

4.1 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 includes a manifest file, `vcpkg-configuration.json`. This file records the vcpkg artifacts required 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.

4.2 Confirm automatic activation

The Environment Manager extension automatically activates the workspace and downloads the tools specified in your `vcpkg-configuration.json` file when you perform the following actions:

- Open a new workspace
- Duplicate an existing workspace
- Open an example project from keil.arm.com

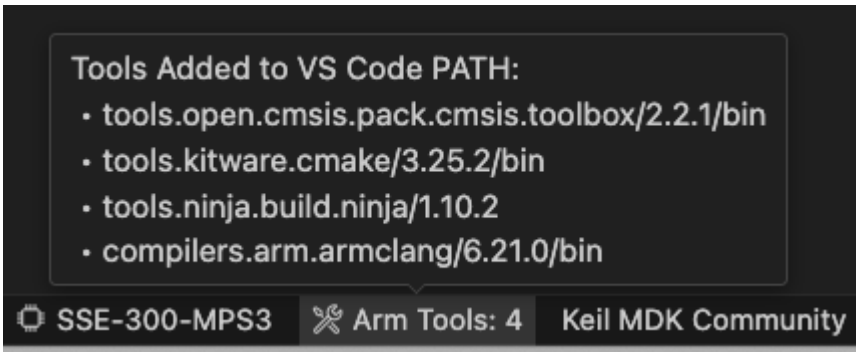
A dialog box opens, allowing you to confirm the activation. Open the `vcpkg-configuration.json` file to see what will be installed. You can also change the automatic activation at any time from the settings.

4.3 Check the tools installed with Microsoft vcpkg

The `vcpkg-configuration.json` manifest file instructs Microsoft vcpkg to install tool artifacts like compilers, models, and tools.

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

Figure 4-1: Arm Tools



You can also click **Arm Tools** in the status bar and select the **View Log** option. This option opens the **Output** tab for the **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. This process allows you to run the different CMSIS-Toolbox tools such as `cpackget`, `cbuildgen`, `cbuild`, or `csolution`.

4.4 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.

Copy the code snippets for the artifacts that you want to install, then paste them in the `"requires":` section of your `vcpkg-configuration.json` file. Save the file to download and activate the newly added or updated artifacts automatically.

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

4.5 Use the Configure Arm Tools Environment visual editor

As an alternative to editing the `vcpkg-configuration.json` manifest file directly, you can use the **Configure Arm Tools Environment** 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 **Configure Arm Tools Environment** editor opens.

   Alternatively, click **Arm Tools** in the status bar and select **Configure Arm Tools Environment** from the drop-down list 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, save your changes. You can enable or disable **Auto Save** from the **File** menu.

   The newly added or updated tools are automatically downloaded and activated. You can view details of what has been installed in the **Output** tab. To open the **Output** tab, select **Output** from the **View** menu.

4.6 vcpkg activation options

Several options are available to manage your environment with Microsoft vcpkg. If you are using an example from keil.arm.com, or if you created a solution from scratch using **Create Solution**, 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 Configure Arm Tools Environment 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 and remove tools 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. Next, select **Add Arm tools Configuration To Workspace** to open the visual editor and select the tools.

4.7 Use vcpkg from the command line

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

The Arm Developer Learning Paths provide a guide on installing and initializing vcpkg and using a configuration file. See Install tools on the command line using vcpkg.

4.8 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

4.8.1 Switch to a specific Arm Compiler for Embedded version

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

Only the versions available on the Artifactory repository manager are exposed in the user interface.


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 Configure Arm Tools Environment visual editor.

   For example, select version `6.22.0` in the **Arm Compiler for Embedded** drop-down list.
2. Restart Visual Studio Code.
3. Right-click the `*.csolution.yml` file and select **Rebuild solution** to rebuild the project.

4.8.2 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 the 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.

4.8.3 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.
3. Make sure that you have added the toolchain to the global environment variables for your operating system.
   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.
4. Restart Visual Studio Code.
5. Use `cbuild list toolchains -v` to check the variable path.

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

Use the following procedure to transfer all the required tools to an air-gapped machine:

1. Use the **Activate Environment** option or run the `vcpkg-shell activate` command from the **Terminal** on a connected machine.
2. Transfer the vcpkg root directory to the air-gapped machine.

You can then use the air-gapped machine without an internet connection.

See vcpkg activation options for more details.

5. Arm CMSIS Solution extension

The Arm CMSIS Solution extension provides support for working with CMSIS solutions, otherwise known as csolution projects. The extension manages the information needed to create your solutions.

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

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

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).

5.1 CMSIS solutions

A solution is a container used to organize related projects that are part of a larger application and that you can build 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.

5.2 Select a solution from the workspace

If you have several solutions in your workspace, the **Select solution from workspace** option enables you to set the active solution and switch between solutions.

You can set the active solution either from the **Explorer** view or from the **Solution outline** view.

- To set the active solution from the **Explorer** view, right-click the folder containing your csolution project files, or the `csolution.yml` file itself, and select **Select solution from workspace**.
- To open a project from the **Solution outline** view:
  - Click **CMSIS** in the Activity Bar to open the **CMSIS** view. The **Solution outline** displays on the left.
  - In the header next to the solution name, click **Views and More Actions** ..., and then select **Select solution from workspace**.
  - Choose a solution from the list that opens at the top of the window. The **CMSIS** view loads and activates your chosen solution.

5.3 Set a context for your solution

Look at your solution contexts. A context set 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 **Manage Solution** 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 and debug configurations, 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 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.

     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 an option in the drop-down list in the `launch.json` file if you are using the Arm Debugger extension:
         - `Arm Debugger: Attach`

- `Arm Debugger: Launch`
- `Arm Debugger: Launch FVP`

See Modify the run configuration options with the Run Configuration visual editor and Modify the debug configuration options with the Debug Configuration visual editor for more details.

4. Go to the **Problems** tab at the bottom of the window and check for errors. If the **Problems** tab is not visible, go to the **View** menu and click **Problems**, or press Ctrl+Shift+M (Windows) or Cmd+Shift+M (macOS).
5. Open the `main.c` file and check the IntelliSense features available. To find out about the different features, read the Visual Studio Code documentation on IntelliSense.

5.4 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.

The **Solution outline** shows the projects (cprojects) included in the solution that are selected in the **Manage Solution** 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:

- User files: User files are the files that you add to the project and that you can edit. For example, a README or code files. You can organize user files using groups, and add files, sub-groups, or component code templates 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 New Group**: Add a sub-group to the group selected
  - **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. You can also manually add files under the `groups` key. They display as groups without names in the **Solution outline**. For example:

```
groups:
    - files:
        - file: README.md
```

- **constructed-files**: Contains files such as the `RTE_Components.h`, `Pre_Include_Global.h`, and `Pre_Include_Local_Component_h` header files that are generated for each context. See RTE_Components.h, Pre_Include_Global_h, and Pre_Include_Local_Component_h for more details.
- **linker**: Contains a linker script file and a <regions>.h file (or other user-defined header files). See Linker Script Management for more details.
- **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.
- **Layer Type:<type>** or **Layer:<base-filename>**: 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. Multiple projects can use the clayer file. The software components used by each layer in the project appear in the tree view.

  If you are using a generator to configure your device or board, then a **Run Generator** option is available from the generator component entry under **Layer** > **Components** to start a generator session. For more details, see Generator Support.

The **Solution outline** label displays the name of your active solution. You can choose one of the following actions next to the solution name:

- **Build solution**: Click  to build the projects included in the context set that you defined in the **Manage Solution** view.
- **Run**: Click  to run the solution on your hardware.
- **Debug**: Click  to debug the solution.
- **Open csolution.yml 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 collapse all the entries in the outline.
- **Views and More Actions**  :
  - **Clean all out and tmp directories**: Clean the output and tmp directories for the active solution
  - **Rebuild solution**: Clean the output directories before building the projects
  - **Refresh**: Refresh the Explorer view
  - **Convert uVision project**: Convert an existing µVision project to a solution
  - **Create new solution**: Create a solution from scratch
  - **Select solution from workspace**: Select the active solution. If you have several solutions in your workspace, this option allows you to switch switch between solutions. The same option is available from the **Explorer** when you right-click the folder that contains

your csolution project files. You can also access this option when you right-click the `csolution.yml` file itself. The active solution is shown as highlighted.

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 Open-CMSIS-Pack documentation gives more information on the `*.csolution.yml`, `*.cproject.yml`, and `*.clayer.yml` file formats.

5.5 CMSIS-Packs

CMSIS-Packs are 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. Examples include RTOS (Real-Time Operating System), DSP (Digital Signal Processing), and generic middleware.
- Device parameters, for example the memory layout or debug settings, along with startup code and Flash programming algorithms
- Board support, for example 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

Various silicon and software vendors develop CMSIS-Packs, 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.

5.6 Install CMSIS-Packs

If you use an example from keil.arm.com, the CMSIS-Packs that you need are already listed in the `csolution.yml` file under the `packs` key. The CMSIS Solution extension scans your pack cache and installs any missing packs.

If you need to add CMSIS-Packs, or if you are creating a solution from scratch, 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.

5.6.1 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 5-1: wolfSSL example

4. Save the `csolution.yml` file.
   Missing CMSIS-Packs are installed automatically.

Note

If you have deactivated the automatic download of packs with the **Download Packs** setting, then errors display in the **Output** tab for the **CMSIS Build Manager** category. Use the `cpackget pack add` command to add packs manually.

5.7 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
- 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

5.7.1 Open the Software Components view

This section describes how to open the **Software Components** view.

Procedure

1. Click **CMSIS** in the Activity Bar to open the **CMSIS** view.
2. Move your cursor over the **Solution outline**. 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-name>** drop-down list and **All** toggle button).

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

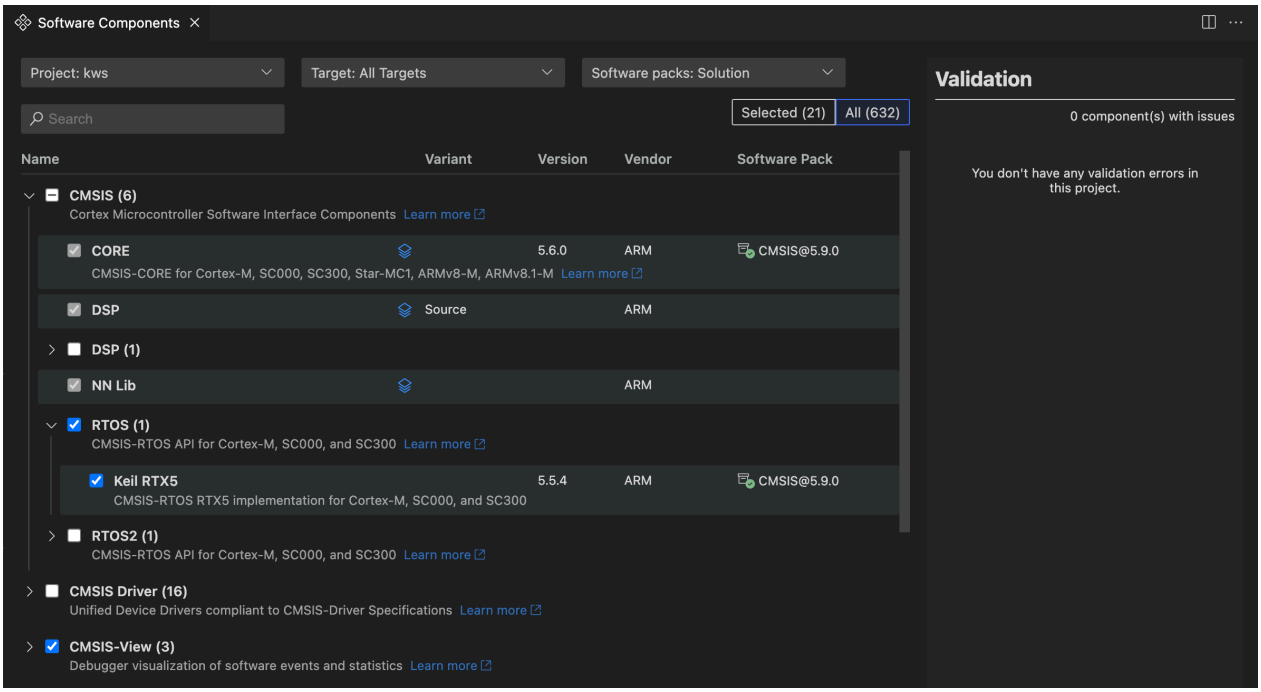
You can use the **Search** field to search the list of components.

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 installed packs.

Figure 5-2: 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 must 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.

5.7.2 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. 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 with the **Solution: <Solution-name>** option. You can display the components from all installed packs with the **All installed packs** option.
4. Check that the **All** toggle button is selected to display all the components available. 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 also be 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.

5.7.3 Undo changes

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

5.8 Use the Configuration Wizard

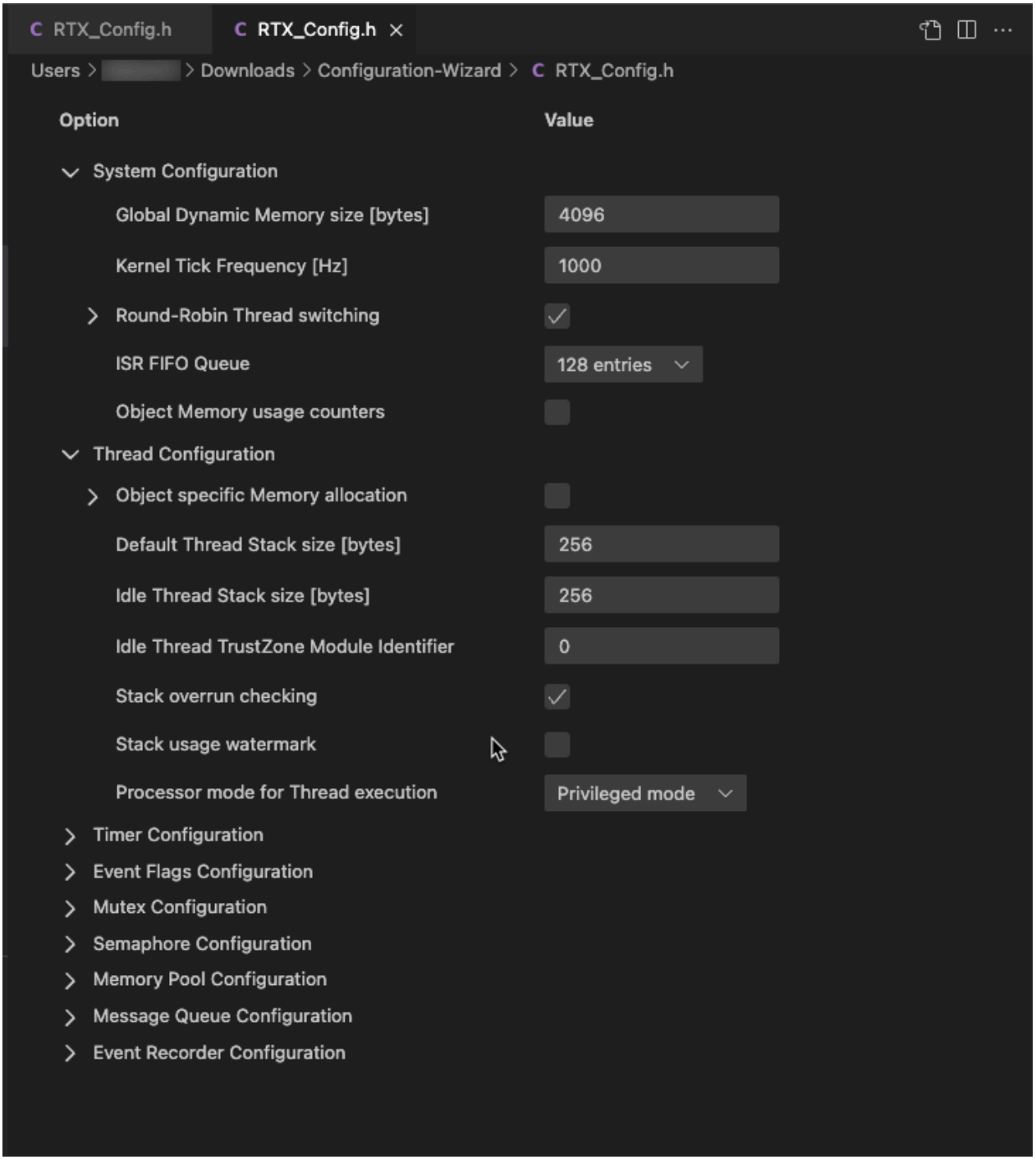
The Configuration Wizard simplifies the customization of startup code and configuration files by providing an intuitive dialog-style interface. The wizard allows you to quickly modify configuration settings without the need for extensive manual editing.

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**  .

Figure 5-3: Configuration Wizard



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

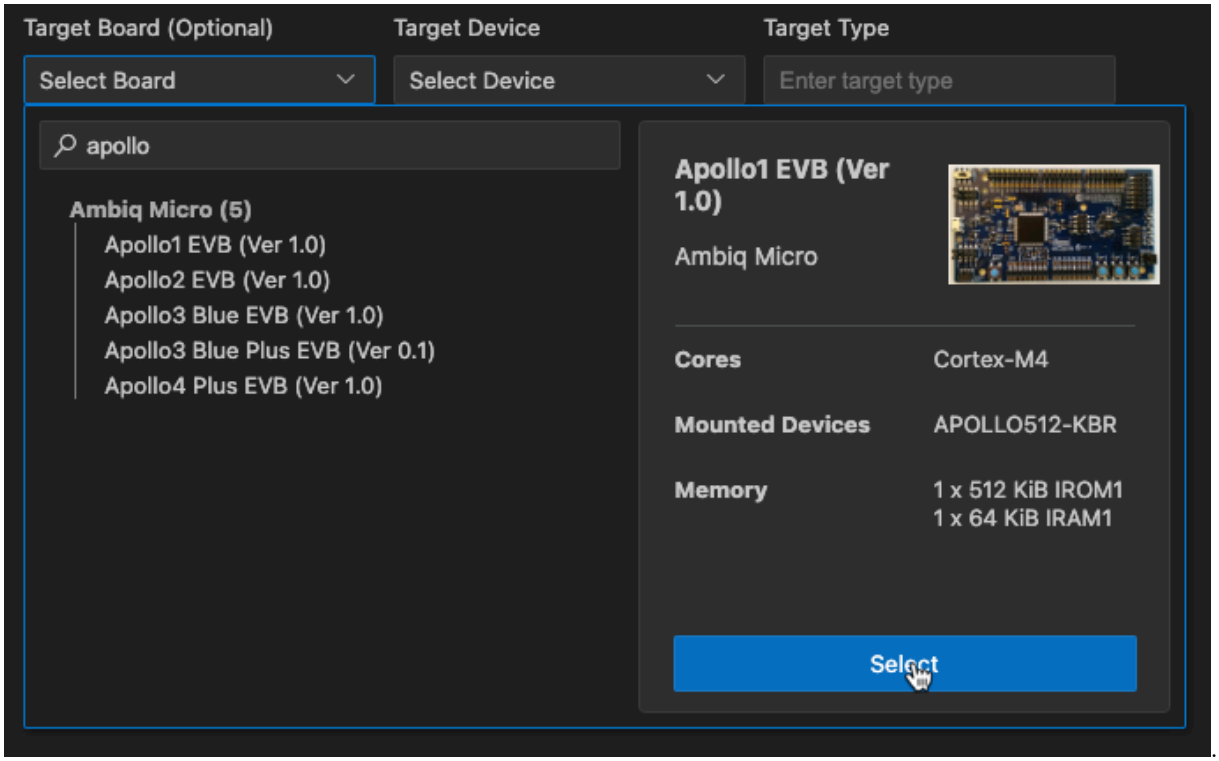
5.9 Create a solution

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**. Next, click **Views and More Actions** > **Create new solution**.
   - Go to the **File** menu and select **New File...**, then select **Create new solution** in the drop-down list that opens at the top of the window.

   The **Create 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**.
   By default, the **Target Device** drop-down list and **Target Type** field show 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, without selecting a board first.

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 **Manage Solution** view. The `target-types: - type:` section of the `<solution_name>.csolution.yml` file is updated automatically when you set a target type in the user interface.
5. Select one of the following options from the **Templates, Reference Applications, and Examples** drop-down list. Note that the option or options available depend on the board or device that you selected. If there are many examples available, enter a search term, and then select an example.
   - **Templates**: Templates are projects that you can use to get started. Templates do not include application-specific code.
     - 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 whether to use TrustZone, and define whether projects in the solution use secure or non-secure zones.
   - **Reference examples**: Use a reference application. Examples are only available if you selected a board in the **Target Board** drop-down list. Reference applications are not dependent on specific hardware. You can deploy them to various evaluation boards using additional software layers that provide driver APIs for specific target hardware. Layers are provided using Board Support Packs (BSPs).

     Reference applications are available with the MDK-Middleware software pack. These examples show you how to use software components for IPv4 and IPv6 networking, USB Host and Device communication, and file system for data storage. The examples use board layers. See MDK Middleware Reference Applications and the MDK-Middleware repository and documentation for more details.

     Other reference applications that illustrate how to match sensor shields and boards are also available with the Sensor SDK pack. The examples use board and shield layers. See Sensor Reference Applications and the Sensor-SDK-Example repository for more details.
   - **CMSIS solution examples**: Use a CMSIS solution example. CMSIS solution examples are created for a specific hardware or evaluation board. The examples are complete projects that directly interface with board and device peripherals.
   - **µVision examples**: Use a µVision example in `*.uvprojx` format as a starting point. µVision examples are converted automatically.
6. For blank and TrustZone solutions only:
   a. Configure the projects in your solution:
      - If you selected `Blank solution`: CMSIS Solution extension adds one project 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`: CMSIS Solution extension adds a secure project and a non-secure project for each processor in the target hardware that supports TrustZone. You can change the project names. You can also change the `secure` or `non-secure` zones in the **TrustZone** drop-down list, or remove TrustZone by selecting `off`.

b. 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.

c. Select a compiler: **Arm Compiler 6**, **GCC**, or **LLVM**.

7. You can change the name for your solution in the **Solution Name** field.
8. Click **Browse** next to the **Solution Location** field and choose where to store the files of the solution using the system dialog box.
9. 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.
10. Click **Create**.
    The extension creates the solution and automatically converts examples that are available only in `*.uvprojx` format. If there are conversion errors, check the `uv2csolution.log` file.

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

    - Open the solution in a new workspace with the **Open** option
    - Open the solution in a new window and new workspace with the **Open project in new window** option
    - Add the solution to the current workspace with the **Add project to vscode workspace** option
11. 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.
12. 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.
13. 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. For reference applications only, each `cproject.yml` file contains a `$Board-Layer$` variable. For reference applications with sensor shields, each `cproject.yml` file contains a `$Shield-Layer$` variable too (`layers: - layer:`). These variables are not yet defined.
    - A main `<filename>.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.

See Configure a solution to add board and shield layers to your reference application. You can also select a compiler for reference applications and other solution types.

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

5.10 Configure a solution

If you have not already set a compiler, select a compiler for your solution from the **Configure Solution** view. If you created a reference application from a reference example, you can also add layers to your solution from the same view.

Procedure

1. Click **CMSIS** in the Activity Bar to open the **CMSIS** view.
2. Make sure that your solution is the active solution in the **Solution outline**, otherwise use the **Select solution from workspace** option in **Views and More Actions** ....
3. Run the **CMSIS: Configure Solution** command from the Command Palette.
4. If you are working with a reference application, the **Add Software Layer** area displays, showing the software layers that you can use. Layers are available from the Board Support Packs (BSPs) installed on your machine.

   Note

   - Not all BSPs have board layers.
   - Not all layers are compatible with the connections that your reference application requires.
   - The CMSIS-Packs which contain reference applications and layers generally provide an `Overview.md` file where the connections are detailed.

   If there are no compatible layers, errors display.

   a. Click **Next** to display the different options available.

   b. You can indicate where the layers should be copied to in the **Board-Layer** and **Shield-Layer** fields. Click **Default** to reset the paths to their default values.

   c. If no compiler is set for the reference application, the **Select Compiler** area displays under the layers selection and shows the compilers available in your environment. Select a compiler. For example, **AC6**.
5. If you are working with another solution type, only the **Select Compiler** area displays. Select a compiler.
6. Click **OK**.
   For reference applications only, a `Board.clayer.yml` file and a `Shield.clayer.yml` file are added in the folders that you selected. The files are available from the **Explorer**. Layers are automatically added in the `csolution.yml` file of your solution under `target-types: variables:`.

For all solution types, the compiler is added with the `compiler:` key.

5.11 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 information, see the Migrate Arm Compiler 5 to Arm Compiler 6 application note and the Arm Compiler for Embedded Migration and Compatibility Guide.

Procedure

1. Drag and drop the folder that contains the `*.uvprojx` that you want to convert onto the Visual Studio Code workspace to open it. Alternatively, import a µVision project from keil.arm.com, or clone a project from GitHub.
2. Right-click the `*.uvprojx` and select **Convert uVision project** 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 **Views and More Actions**  , then select **Convert uVision project** 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 with the **Open** option
   - Open the solution in a new window and new workspace with the **Open project in new window** option

   You can also run the **CMSIS: Convert uVision project** 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.
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. In the **View** menu, select **Output** to open the **Output** tab. Select **µVision to Csolution Conversion** in the drop-down list on the right side of the **Output** tab. The `*.cproject.yml` and `*.csolution.yml` files are available in the folder where the `*.uvprojx` is stored.

5.12 Configure a build task

In Visual Studio Code, you can automate certain tasks by configuring a `tasks.json` file. 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

The examples on keil.arm.com include 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 that does not have a build task configured, 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.

5.13 Initialize your solution

If your solution does not already contain a `vcpkg-configuration.json`, `tasks.json`, and `launch.json` files, use the **Initialize CMSIS project** option to generate them. Examples from keil.arm.com or solutions created from scratch from the **Create 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.

5.14 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.

6. Arm 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**.

6.1 Supported hardware

This section describes the hardware that the Device Manager extension and other Keil® Studio extensions support.

6.1.1 Supported development boards and MCUs

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

6.1.2 Supported debug probes

The following debug probes are supported.

6.1.2.1 WebUSB-enabled CMSIS-DAP debug probes

The extensions support debug probes that implement the CMSIS-DAP protocol, for example:

- 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 ULINKplus documentation

See the CMSIS-DAP documentation for general information.

6.1.2.2 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.

6.2 Connect your hardware

This section 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 a USB connection.
   The Device Manager detects the hardware and a pop-up message displays in the bottom right-hand corner.
3. Click **OK** to use the hardware.

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

   You can now use your hardware to run and debug a project.

Next steps

To add more hardware, click **Add Device**  in the top right-hand corner.

6.3 Edit your hardware

If the Device Manager cannot detect your hardware or if you are using an external debug probe, you can edit the hardware entry from the Device Manager. You can 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**. The hardware name 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**.

6.4 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 baud rate is 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.

7. Arm Debugger extension

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

To run a project on your hardware, you must first configure a task. Run configuration details are saved in the `tasks.json` file of your project. For debugging, you also must add a launch configuration to your project first. Debug configuration details are saved in the `launch.json` file.

You can configure the `tasks.json` and `launch.json` files manually. See Configure a task and Add a configuration for more details.

You can also use the **Run Configuration** and **Debug Configuration** visual editors to quickly create the configurations you need. See Modify the run configuration options with the Run Configuration visual editor and Modify the debug configuration options with the Debug Configuration visual editor.

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.

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**.

7.1 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.

7.1.1 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.

7.1.2 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.

For CMSIS use cases, 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, the task configuration can automatically pick this up. Use 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 have not installed the CMSIS Solution extension, then for CMSIS use cases 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}"`

7.1.3 Arm Debugger run configuration options

The extension provides the following run configuration options.

Configuration option	Description
`"cmsisPack"`	The file path or URL path to a DFP (Device Family Pack) CMSIS-Pack for your hardware. You can use this option with: • `cmsis-csolution.getTargetPack`: Gets the DFP CMSIS-Pack for the selected target type in the **CMSIS Solution Manage Solution** 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"`	The connection mode. Possible values: • `auto`: The debugger decides • `haltOnConnect`: Halts for any reset before running • `underReset`: Holds external NRST line asserted • `preReset`: Prereset using NRST • `running`: Connects to a running target without altering the state Default: `auto`
`"dbgconf"`	The path to a `.dbgconf` file to configure CMSIS-Pack debug sequence execution. Requires Arm Debugger v6.1.0 or later.

Configuration option	Description
`"debugClockSpeed"`	The maximum clock frequency for the debug communication. The frequency that is actually used depends on the debug probe that you use. `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"`	The 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"`	The CMSIS-Pack device name. You can use this option 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"`	The type of flash erase to use. Requires Arm Debugger v6.1.0 or later. Possible values: • `sectors`: Erase only the sectors to be programmed • `none`: Skip flash erase Default: `sectors`

Configuration option	Description
`"flms"`	Flash algorithm configurations. Each entry either modifies a default algorithm that is defined in the corresponding DFP (Device Family Pack), or adds an algorithm. Use the 'ignore' field to deactivate a default algorithm from the DFP. Requires Arm Debugger v6.1.0 or later. Required value: • "`path`": A relative path to the flash algorithm file in the DFP, or an absolute path to a flash algorithm file in the file system of your machine Optional values: • "`regionStart`": The start address of the memory region targeted by the flash algorithm in decimal or hexadecimal format. If not set, uses the default start address for the algorithm from the DFP CMSIS-Pack. • "`regionSize`": The size of the memory region targeted by the flash algorithm in decimal or hexadecimal format. If not set, uses the default size for the algorithm from the DFP CMSIS-Pack. • "`ramStart`": The start address of target system's RAM used for execution of flash algorithms. If not set, uses defaults from the DFP CMSIS-Pack. • "`ramSize`": The size of target system's RAM used for execution of flash algorithms. If not set, uses defaults from the DFP CMSIS-Pack. • "`ignore`": Ignores an algorithm as provided in the DFP. The value can be `true` or `false`. If not set, algorithms marked as default in the DFP have an ignore value of `false`, and 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"`	The 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"`	The file path or URL path to a PDSC file.
`"probe"`	The name of a probe to use for the debug connection. Possible values: `ULINKpro`, `ULINKpro D`, `ULINK2`, `CMSIS-DAP`, `ULINKplus`, `ST-Link` Default: `auto`. If the Arm Debugger extension cannot set the probe type automatically, the default value is `CMSIS-DAP`.
`"processorName"`	The CMSIS-Pack processor name for multicore devices.
`"program"` or `"programs"`	One or more file paths or URL paths to one or more projects to use in order of loading. Requires Arm Debugger v6.0.2 or later. You can use this option with: • `arm-debugger.getApplicationFile`: Returns an AXF or ELF file used for CMSIS run and debug
`"serialNumber"`	The serial number of the connected USB hardware to use. You can use this option with: • `device-manager.getSerialNumber`: Gets the serial number of the selected device
`"targetAddress"`	Synonymous with `serialNumber`.

Configuration option	Description
`"targetInitScript"`	The path to a target initialization script (`.ds/.py`) executed after connection but before any other operation. Requires Arm Debugger v6.1.0 or later.
`"vendorName"`	The CMSIS-Pack vendor name.
`"verifyFlash"`	Verify the contents downloaded to flash. Requires Arm Debugger v6.1.0 or later.
`"workspaceFolder"`	The current Arm Debugger workspace folder. Default: `"${workspaceFolder}"`

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

7.1.4 Modify the run configuration options with the Run Configuration visual editor

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

Procedure

1. There are 3 ways to open the editor:
   - From the **Explorer**, right-click the `tasks.json` file in the `.vscode` folder of the solution and select **Open Run Configuration**.
   - From the **Explorer**, right-click the `tasks.json` file. Select **Open With...**, and then select **Run 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 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. To duplicate the currently selected configuration and modify it, click **Duplicate**.
3. You can change the name of the configuration in the **Configuration Name** field.
4. Modify your run configuration. See Run configuration options in the visual editor for more details.
5. If **Auto Save** is not enabled, save your changes. You can enable or disable **Auto Save** from the **File** menu.
   The Arm Debugger extension updates the `tasks.json` file.

7.1.4.1 Run configuration options in the visual editor

Modify the run configuration options for your solution.

Probe / Unit	Action
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: `auto`. If the Arm Debugger extension cannot set the probe type automatically, the default value is `CMSIS-DAP`. • You can connect a probe or a board to your computer over USB. In this case, 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 Arm Debugger extension uses the serial number of the active device in the Arm Device Manager extension by default. The Arm Debugger extension adds the `"${command:device-manager.getSerialNumber}"` command in the JSON file for `"serialNumber"`. • You can also select the serial number of the active device in the drop-down list. Alternatively, type a serial number. To check what your active device is, click **Open Arm Device Manager**.

Target	Action
CMSIS-Pack	Select the Device Family Pack (DFP) for the target debug probe or board. • Default value: `auto (CMSIS Solution)`. The Arm Debugger extension uses the DFP for the active device defined in the `*.csolution.yml` file of your solution. The Arm Debugger extension adds the `"${command:cmsis-csolution.getTargetPack}"` command in the JSON file for `cmsisPack`. • `auto (Device Manager)`: The Arm Debugger extension uses the DFP for the active device in the Arm Device Manager extension. The Arm Debugger extension adds the `"${command:device-manager.getDevicePack}"` command in the JSON file for `cmsisPack`. • You can also select the DFP for the active device in the drop-down list. Alternatively, 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, that is, the target chip on your board. • Default value: `auto (CMSIS Solution)`. The Arm Debugger extension detects the device name from the information available for the probe or board in the `*.csolution.yml` file of your solution. The Arm Debugger extension adds the `"${command:cmsis-csolution.getDeviceName}"` command in the JSON file for `deviceName`. • `auto (Device Manager)`: The Arm Debugger extension detects the device name from the information available for the probe or board in the Arm Device Manager extension. The Arm Debugger extension adds the `"${command:device-manager.getDeviceName}"` command in the JSON file for `deviceName`. • You can also select the device name in the drop-down list. Alternatively, 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	For multicore devices, select the processor to use. • Default value: `auto (CMSIS Solution)`. The Arm Debugger extension uses the processor name that is defined in the `*.csolution.yml` file of your solution. The Arm Debugger extension adds the `"${command:cmsis-csolution.getProcessorName}"` command in the JSON file for `"processorName"`. • You can also directly type a processor name. For example: `cm4`.

Application	Action
Program Files	One or more programs to run on your hardware • Default value: The Arm Debugger extension adds the `${command:arm-debugger.getApplicationFile}` command in the JSON file for `"program"` when you add a new configuration block. This command detects the latest AXF or ELF file generated. • To point to a file directly, click **Add File**. You can add as many files as you need. The Arm Debugger extension uses the files in the order in which you added them. The Arm Debugger extension supports AXF and ELF files by default. You can add other file types. • To add the `${command:arm-debugger.getApplicationFile}` command if it is not available, click **Detect File**. • To remove the selection, move your cursor over the name of the command or file and click the **Delete Program File** icon.

Run	Action
Connection Mode	Select a connection mode. The connection mode controls the operations that 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 you select `auto`, the debugger uses `underReset`. For other boards, the debugger uses `haltOnConnect`. • `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`.

Flash Setup	Action
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 Setup	Action
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**: A relative path to a default flash algorithm file in the DFP, or an absolute path to a flash algorithm file in the file system of your machine • **Region Start**: The start address of the memory region targeted by the selected flash algorithm • **Region Size**: The size of the memory region targeted by the selected flash algorithm • **RAM Start**: The start address in the RAM of the target system used for the execution of the flash algorithm • **RAM Size**: The size of the RAM in the target system 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 you do not set these fields, then the Arm Debugger extension uses the default values from the DFP. To add your own flash algorithms, edit the `tasks.json` file manually: 1. Add `"path"` under the `"flms"` key in the JSON file. 2. Fill in the **Region Start**, **Region Size**, **RAM Start**, and **RAM Size** fields from the visual editor, or edit the JSON file directly.

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

7.1.5 Run your project

Run the project on your hardware.

Before you begin

When you have several solutions in one folder, Visual Studio Code ignores the `tasks.json` and `launch.json` files that you 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 installed the Keil Studio Pack, go to the **CMSIS** view, open the **Manage Solution** view, and check which run configuration is selected. Then, click **Run** in the **Solution outline** header.

4. If you are using a multicore device and you did not specify a `"processorName"` in the `tasks.json` file, and the CMSIS Solution extension is not installed, select the appropriate processor for your project in the **Select a processor** drop-down list at the top of the window.
5. To verify that the project has run correctly, check the **Terminal** tab.
   If the Arm Debugger engine cannot be found on your machine, an **Arm Debugger not found** dialog box displays.

   Select one of these options:

   - To add Arm Debugger to your environment, click **Install Arm Debugger**. The `vcpkg-configuration.json` file is updated. Check the Arm tools installed in the status bar 

.
   - To indicate the path to the Arm Debugger engine in the settings, click **Configure Path**.

7.2 Debug your project with Arm Debugger

Debug a project.

Several debug configurations are available. For CMSIS use cases, use the `Arm Debugger: Launch`, `Arm Debugger: Launch FVP`, or `Arm Debugger: Attach` task.

7.2.1 Add a configuration

To debug your project, you must first add a launch configuration to enable you to configure and save debug setup details. Visual Studio Code keeps debug configuration information in a `launch.json` file. If the system does not detect a configuration, 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 debug configuration settings. You can modify the default configuration.

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: Launch`, `Arm Debugger: Launch FVP`, or `Arm Debugger: Attach` task for CMSIS use cases.

   This task 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 have not installed the CMSIS Solution extension, then for CMSIS use cases 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}"`

7.2.2 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 - CMSIS use cases for more details.

For CMSIS use cases, see also the details provided for the tasks.json file for `cmsisPack` and `deviceName`. 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.

7.2.3 Arm Debugger debug configuration options - CMSIS use cases

The extension provides the following debug configuration options.

Launch configuration options for debugging with a physical target

Configuration option	Description
`"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"`	The file path or URL path to a DFP (Device Family Pack) CMSIS-Pack for your hardware. You can use this option with: • `cmsis-csolution.getTargetPack`: Gets the DFP CMSIS-Pack for the selected target type in the **CMSIS Solution Manage Solution** 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.

Configuration option	Description
`"connectMode"`	The connection mode. Possible values: • `auto`: The debugger decides • `haltOnConnect`: Halts for any reset before running • `underReset`: Holds external NRST line asserted • `preReset`: Prereset using NRST • `running`: Connects to a running target without altering the state Default: `auto`
`"dbgconf"`	The path to a `.dbgconf` file to configure CMSIS-Pack debug sequence execution. Requires Arm Debugger v6.1.0 or later.
`"debugClockSpeed"`	The maximum clock frequency for the debug communication. The frequency that is actually used depends on the debug probe that you use. `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"`	The path to a debug initialization script (`.ds/.py`) executed after connection and running to `debugFrom`. Requires Arm Debugger v6.1.0 or later.
`"debugPortMode"`	The 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"`	The CMSIS-Pack device name. You can use this option 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"`	The file path or URL path to a PDSC file.
`"probe"`	The name of a probe to use for the debug connection. Possible values: `ULINKpro`, `ULINKpro D`, `ULINK2`, `CMSIS-DAP`, `ULINKplus`, `ST-Link` Default: `auto`. If the Arm Debugger extension cannot set the probe type automatically, the default value is `CMSIS-DAP`.
`"processorName"`	The CMSIS-Pack processor name for multicore devices.

Configuration option	Description
`"program"` or `"programs"`	One or more file paths or URL paths to one or more projects to use in order of loading. Requires Arm Debugger v6.0.2 or later. You can use this option 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.
`"resetMode"`	Type of reset to use. Possible values: • `auto`: The debugger decides • `system`: Use ResetSystem sequence • `hardware`: Use ResetHardware sequence • `processor`: Use ResetProcessor sequence Default: `auto`
`"searchPaths"`	Array of paths to source locations.
`"serialNumber"`	The serial number of the connected USB hardware to use. You can use this option with: • `device-manager.getSerialNumber`: Gets the serial number of the selected device
`"svd"`	The file path or URL path to an SVD file.
`"targetAddress"`	Synonymous with `serialNumber`.
`"targetInitScript"`	The path to a target initialization script (`.ds/.py`) executed after connection but before any other operation. Requires Arm Debugger v6.1.0 or later.
`"vendorName"`	The CMSIS-Pack vendor name.
`"workspaceFolder"`	The current Arm Debugger workspace folder. Default: `"${workspaceFolder}"`

Attach configuration options for debugging with a physical target

Configuration option	Description
`address`	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).
`"debugInitScript"`	The path to a debug initialization script (`.ds/.py`) executed after connection and running to `debugFrom`. Requires Arm Debugger v6.1.0 or later.
`"pathMapping"`	A mapping of remote paths to local paths to resolve source files.
`"svd"`	The file path or URL path to an SVD file.

Configuration option	Description
`"targetInitScript"`	The path to a target initialization script (`.ds/.py`) executed after connection but before any other operation. Requires Arm Debugger v6.1.0 or later.

Launch FVP 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 a Mac, you can run FVPs in a Linux container with Docker. To install Docker and clone the https://github.com/Arm-Examples/FVPs-on-Mac repository, follow this Arm Developer Learning Path.

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 ${workspaceFolder}/model_config.txt`
`"debugFrom"`	The symbol the debugger will run to before debugging. Default: `"main"`
`"debugInitScript"`	The path to a debug initialization script (`.ds/.py`) executed after connection and running to `debugFrom`. Requires Arm Debugger v6.1.0 or later.
`"fvpParameters"`	The path to an FVP parameter configuration file.
`"pathMapping"`	A mapping of remote paths to local paths to resolve source files.
`"program"` or `"programs"`	One or more file paths or URL paths to one or more projects to use in order of loading. Requires Arm Debugger v6.0.2 or later. You can use this option with: • `arm-debugger.getApplicationFile`: Returns an AXF or ELF file used for CMSIS run and debug
`"programMode"`	Mode to program an application to a target. Default value: `ram`
`"searchPaths"`	Array of paths to source locations.
`"svd"`	The file path or URL path to an SVD file.
`"targetInitScript"`	The path to a target initialization script (`.ds/.py`) executed after connection but before any other operation. Requires Arm Debugger v6.1.0 or later.
`"workspaceFolder"`	The current Arm Debugger workspace folder. Default: `"${workspaceFolder}"`

7.2.4 Modify the debug configuration options with the Debug Configuration visual editor

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

7.2.4.1 Debug configuration for a physical target

This section describes how to define a `Launch` or an `Attach` configuration (CMSIS use cases) with the **Debug Configuration** visual editor.

Procedure

1. There are 3 ways to open the editor:
   - From the **Explorer**, right-click the `launch.json` file in the `.vscode` folder of the solution and select **Open Debug Configuration**.
   - From the **Explorer**, right-click the `launch.json` file. Select **Open With...**, and then select **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 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:
   - `Launch` debugging mode: For CMSIS use cases, select a `Launch` configuration to launch your program in debug mode using a physical target.
   - `Attach` debugging mode: For CMSIS use cases, select an `Attach` configuration if you want to debug a program that is already running.

     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.

   To duplicate the currently selected configuration and modify it, click **Duplicate**.
3. You can change the name of the configuration in the **Configuration Name** field.
4. Modify your debug configuration.
   - If you are defining a **Launch** configuration, see Launch configuration for more details.
   - If you are defining an **Attach** configuration, see Attach configuration for more details.
5. If **Auto Save** is not enabled, save your changes. You can enable or disable **Auto Save** from the **File** menu.
   The Arm Debugger extension updates the `launch.json` file.

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

This section describes how to define a `Launch FVP` configuration (CMSIS use cases) with the **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 **Configure Arm Tools Environment** visual editor or by editing the `vcpkg-configuration.json` file directly. Select the **Arm Virtual Hardware for Cortex®-M based on Fast Models** option with the visual editor.

Note

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

Procedure

1. There are 3 ways to open the editor:
   - From the **Explorer**, right-click the `launch.json` file in the `.vscode` folder of the solution and select **Open Debug Configuration**.
   - From the **Explorer**, right-click the `launch.json` file. Select **Open With...**, and then select **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 Debug Configuration**  in the top right-hand corner.
2. To work with a virtual target, you can define a `Launch FVP` configuration for CMSIS use cases 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.
   To duplicate the currently selected configuration and modify it, click **Duplicate**.
3. You can change the name of the configuration in the **Configuration Name** field.
4. Modify your debug configuration. See Launch FVP configuration for more details.
5. If **Auto Save** is not enabled, save your changes. You can enable or disable **Auto Save** from the **File** menu.
   The Arm Debugger extension updates the `launch.json` file.

7.2.4.3 Debug configuration options in the visual editor

Modify the debug configuration options for your solution.

Launch configuration

Probe / Unit	Action
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: `auto`. If the Arm Debugger extension cannot set the probe type automatically, the default value is `CMSIS-DAP`. • You can connect a probe or a board to your computer over USB. In this case, 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 Arm Debugger extension uses the serial number of the active device in the Arm Device Manager extension by default. The Arm Debugger extension adds the `"${command:device-manager.getSerialNumber}"` command in the JSON file for `"serialNumber"`. • You can also select the serial number of the active device in the drop-down list. Alternatively, type a serial number. To check what your active device is, click **Open Arm Device Manager**.

Target	Action
CMSIS-Pack	Select the Device Family Pack (DFP) for the target debug probe or board. • Default value: `auto (CMSIS Solution)`. The Arm Debugger extension uses the DFP for the active device defined in the `*.csolution.yml` file of your solution. The Arm Debugger extension adds the `"${command:cmsis-csolution.getTargetPack}"` command in the JSON file for `cmsisPack`. • `auto (Device Manager)`: The Arm Debugger extension uses the DFP for the active device in the Arm Device Manager extension. The Arm Debugger extension adds the `"${command:device-manager.getDevicePack}"` command in the JSON file for `cmsisPack`. • You can also select the DFP for the active device in the drop-down list. Alternatively, 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, that is, the target chip on your board. • Default value: `auto (CMSIS Solution)`. The Arm Debugger extension detects the device name from the information available for the probe or board in the `*.csolution.yml` file of your solution. The Arm Debugger extension adds the `"${command:cmsis-csolution.getDeviceName}"` command in the JSON file for `deviceName`. • `auto (Device Manager)`: The Arm Debugger extension detects the device name from the information available for the probe or board in the Arm Device Manager extension. The Arm Debugger extension adds the `"${command:device-manager.getDeviceName}"` command in the JSON file for `deviceName`. • You can also select the device name in the drop-down list. Alternatively, 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.

Application	Action
Program Files	One or more programs to use for debugging • Default value: The Arm Debugger extension adds the `${command:arm-debugger.getApplicationFile}` command in the JSON file for `"program"` when you add a new configuration block. This command detects the latest AXF or ELF file generated. • To point to a file directly, click **Add File**. You can add as many files as you need. The Arm Debugger extension uses the files in the order in which you added them. The Arm Debugger extension supports AXF and ELF files by default. You can add other file types. • To add the `${command:arm-debugger.getApplicationFile}` command if it is not available, click **Detect File**. • To remove the selection, move your cursor over the name of the command or file and click the **Delete Program File** icon.
Processor Name	For multicore devices, select the processor to use. • Default value: `auto (CMSIS Solution)`. The Arm Debugger extension uses the processor name that is defined in the `*.csolution.yml` file of your solution. The Arm Debugger extension adds the `"${command:cmsis-csolution.getProcessorName}"` command in the JSON file for `"processorName"`. • You can also directly type a processor name. For example: `cm4`.

Debug	Action
Connection Mode	Select a connection mode. The connection mode controls the operations that 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 you select `auto`, the debugger uses `underReset`. For other boards, the debugger uses `haltOnConnect`. • `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 to start the debugger. 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`. The default value is `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 into RAM before execution.

Debug	Action
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`.
Target Initialization Script	The path to a target initialization script (`.ds/.py`) executed after connection but before any other operation. Requires Arm Debugger v6.1.0 or later.
Debug Initialization Script	The path to a debug initialization script (`.ds/.py`) executed after connection and running to `debugFrom`. Requires Arm Debugger v6.1.0 or later.

Attach configuration

Connection	Action
Address	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).

Debug	Action
Target Initialization Script	The path to a target initialization script (`.ds/.py`) executed after connection but before any other operation. Requires Arm Debugger v6.1.0 or later.
Debug Initialization Script	The path to a debug initialization script (`.ds/.py`) executed after connection and running to `debugFrom`. Requires Arm Debugger v6.1.0 or later.

Launch FVP configuration

Target	Action
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. You can filter the list of FVPs to only display the models installed on your machine with the **Show only installed** checkbox.
FVP Parameters	For more advanced configuration settings, you can generate a list of FVP parameters and modify the arguments that are listed in the file. To generate an `fvp_config.txt` file, click **Generate File**. To open the FVP Parameters editor and modify the file, click the pen icon. If you already have a file available, click **Select File** to select it.

Application	Action
Program Files	One or more programs to use for debugging • Default value: The Arm Debugger extension adds the `${command:arm-debugger.getApplicationFile}` command in the JSON file for `"program"` when you add a new configuration block. This command detects the latest AXF or ELF file generated. • To point to a file directly, click **Add File**. You can add as many files as you need. The Arm Debugger extension uses the files in the order in which you added them. The Arm Debugger extension supports AXF and ELF files by default. You can add other file types. • To add the `${command:arm-debugger.getApplicationFile}` command if it is not available, click **Detect File**. This command detects the latest AXF or ELF file generated. • To remove the selection, move your cursor over the name of the command or file and click the **Delete Program File** icon.

Debug	Action
Debug From	Select a function from which to start the debugger. Default value: `main`. The debugging session starts and the debugger stops at the `main()` function of the program.

7.2.5 Start an Arm Debugger session

Start a debug session.

Before you begin

When you have several solutions in one folder, Visual Studio Code ignores the `tasks.json` and `launch.json` files that you 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.

7.2.5.1 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 a debug configuration in the list 

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

3. If you are using a multicore device and you did not specify a `"processorName"` in the `launch.json` file, and the CMSIS Solution extension is not installed, select the appropriate processor for your project in the **Select a processor** drop-down list 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. To see the debugging output, check the **Debug Console** tab.
   If the Arm Debugger engine cannot be found on your machine, an **Arm Debugger not found** dialog box displays.

   Select one of these options:

   - To add Arm Debugger to your environment, click **Install Arm Debugger**. The `vcpkg-configuration.json` file is updated. Check the Arm tools installed in the status bar 

.
   - To indicate the path to the Arm Debugger engine in the settings, click **Configure Path**.

7.2.5.2 Start a debug session with a virtual target

Start a debug session with a virtual target.

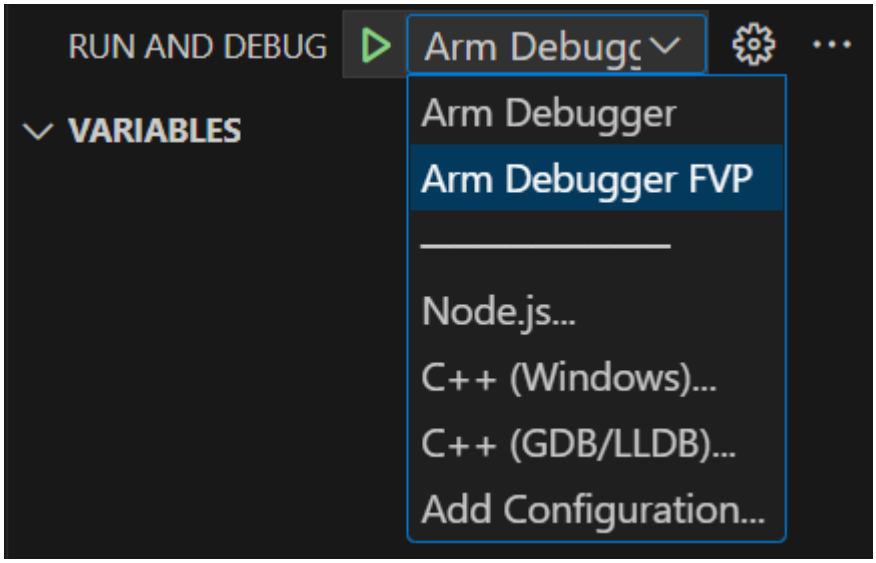
Before you begin

Fixed Virtual Platforms (FVPs) are natively available on Windows and Linux only. If you are on a 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 a debug configuration in the list. Click **Start Debugging**.

Figure 7-1: FVP configuration



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

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

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

   Select one of these options:

   - To add Arm Debugger to your environment, click **Install Arm Debugger**. The `vcpkg-configuration.json` file is updated. Check the Arm tools installed in the status bar 

.
   - To indicate the path to the Arm Debugger engine in the settings, click **Configure Path**.

7.2.6 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 being run, 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**.

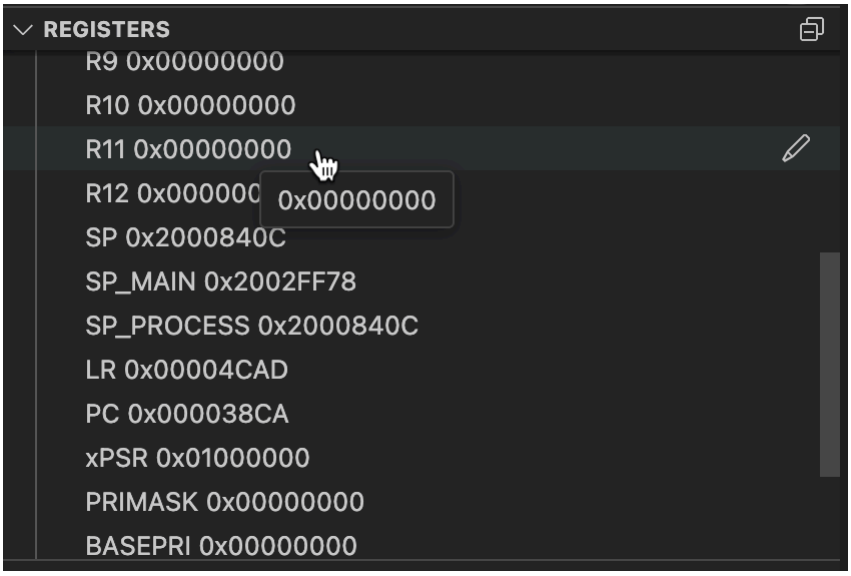
7.2.7 Inspect registers

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

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 runs.

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

Figure 7-2: 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. The processor uses these registers for various operations, including arithmetic, logical, and data movement instructions. Additionally, Arm processors can also have other specific registers, for example the Program Counter (PC) and Stack Pointer (SP). These registers are essential for managing program flow and maintaining the stack. These registers collectively form the register file of the processor core. The register file provides 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 the behavior of a processor. These registers are part of the Arm architecture and play a crucial role in managing system-level functionality. System registers help to control the operating mode of the processor, 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 to perform operations like addition, subtraction, multiplication, and division on them.

7.2.7.1 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, `$SP+0x20` adds `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.

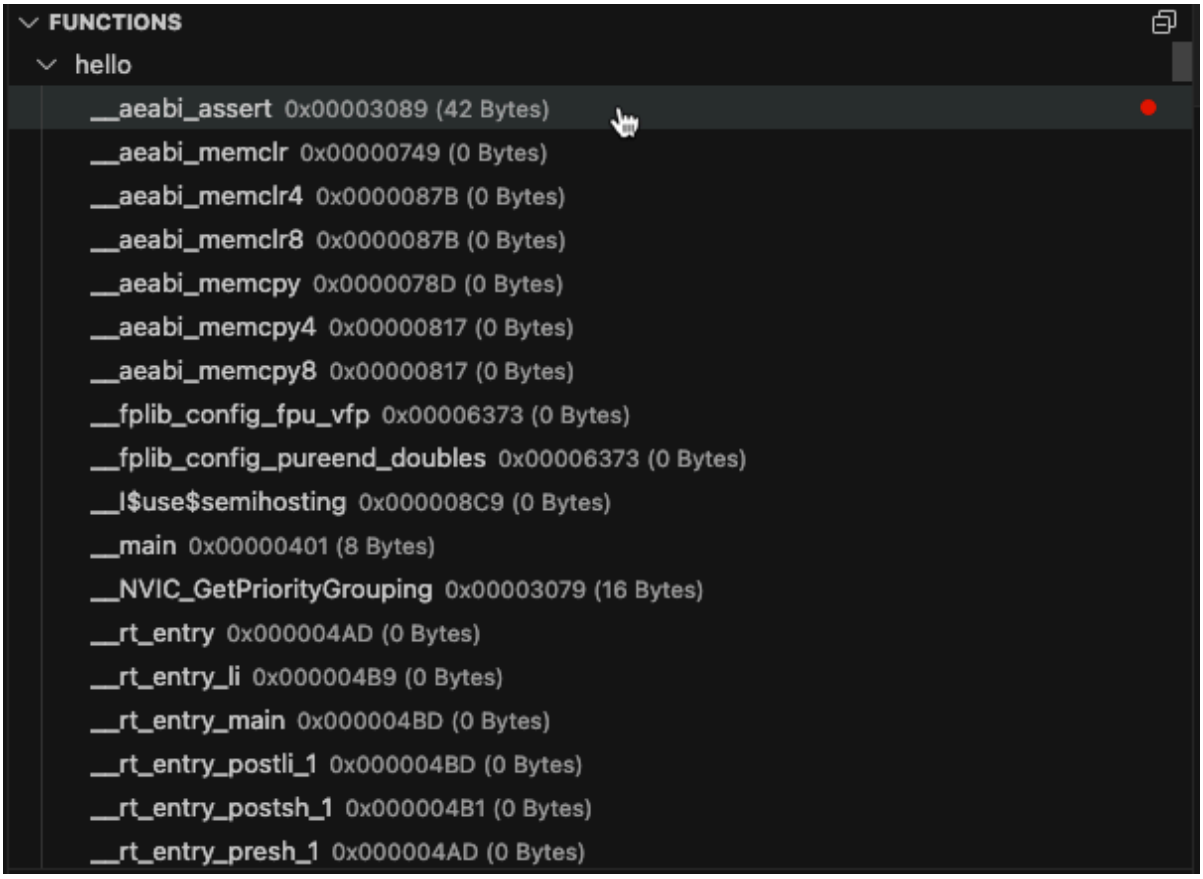
7.2.8 Inspect functions

The **Functions** view displays the main functions in your code, library functions, and user-defined functions. To display the **Functions** view in the **Run and Debug** view, start a debug session. See Start an Arm Debugger session for more information.

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

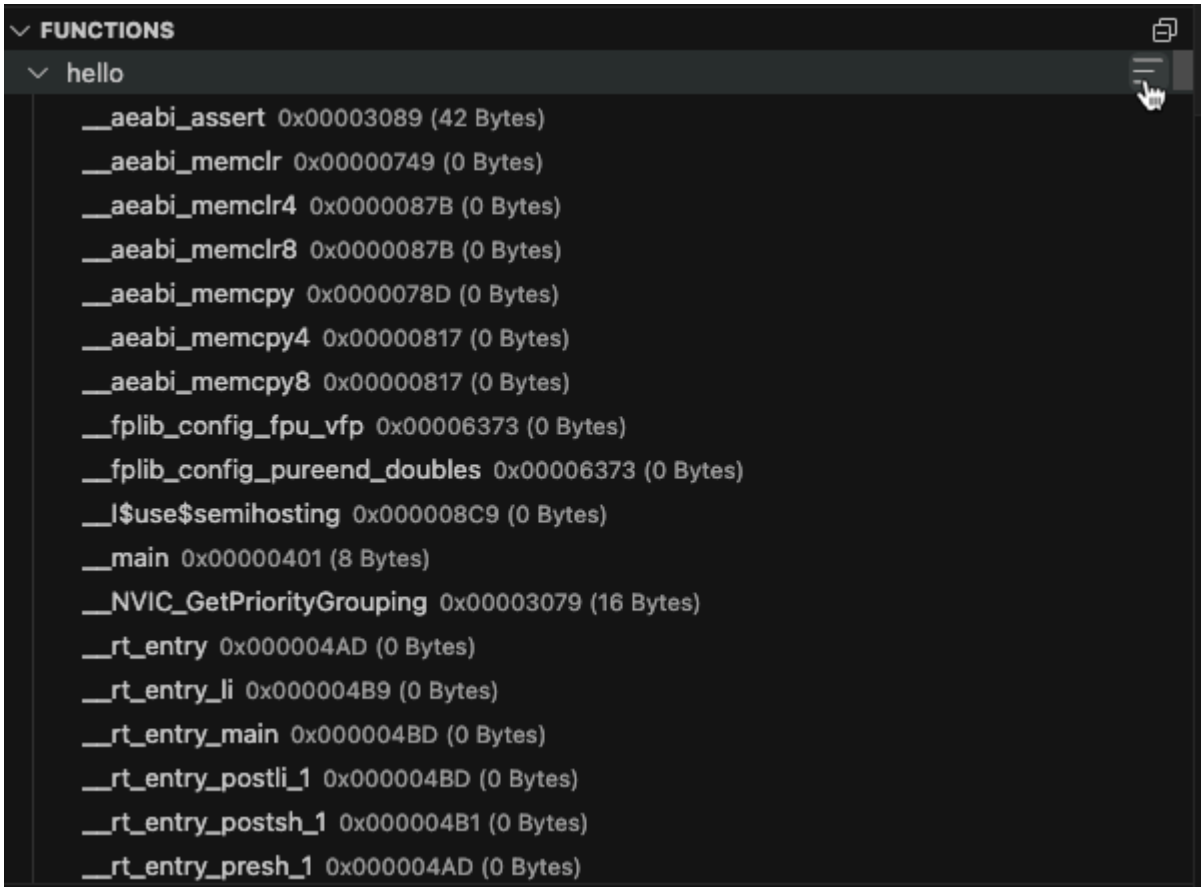
Figure 7-3: Functions view



You can sort functions by a name, an address, or by size:

1. Move your cursor over the name of the solution in the **Functions** view and click **Sort**.

Figure 7-4: Functions view Sort button



2. Select an option in the drop-down list that displays at the top of the window.
   - **Sort By Label**
   - **Sort By Address (Desc)**
   - **Sort By Function Size (Desc)**

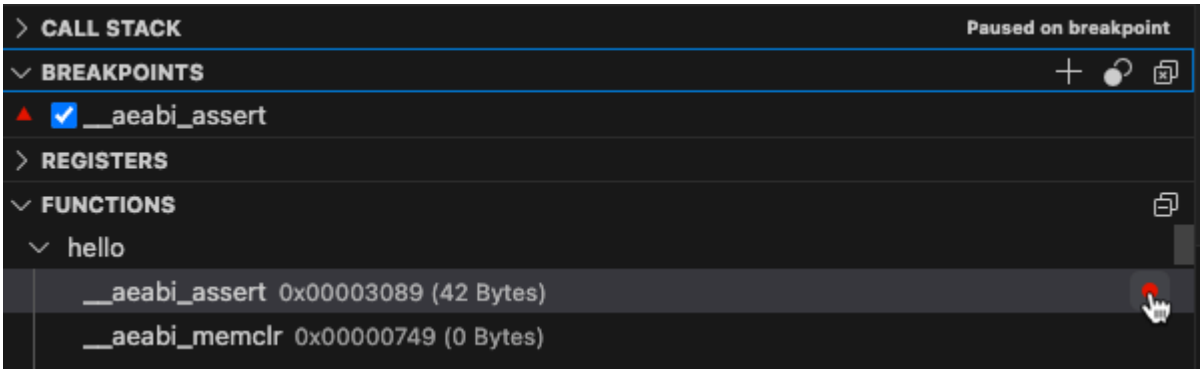
You can add function breakpoints to break execution when a function is called. Breaking execution is useful when you know the function name but not its location.

To add a function breakpoint:

In the **Functions** view, move your cursor over the function for which you want to add a breakpoint and choose one of the following actions:

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

Figure 7-5: Add a function breakpoint



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

From the **Breakpoints** view, you can carry out the following tasks:

- Remove all breakpoints with or remove specific breakpoints

Figure 7-6: Remove specific breakpoints

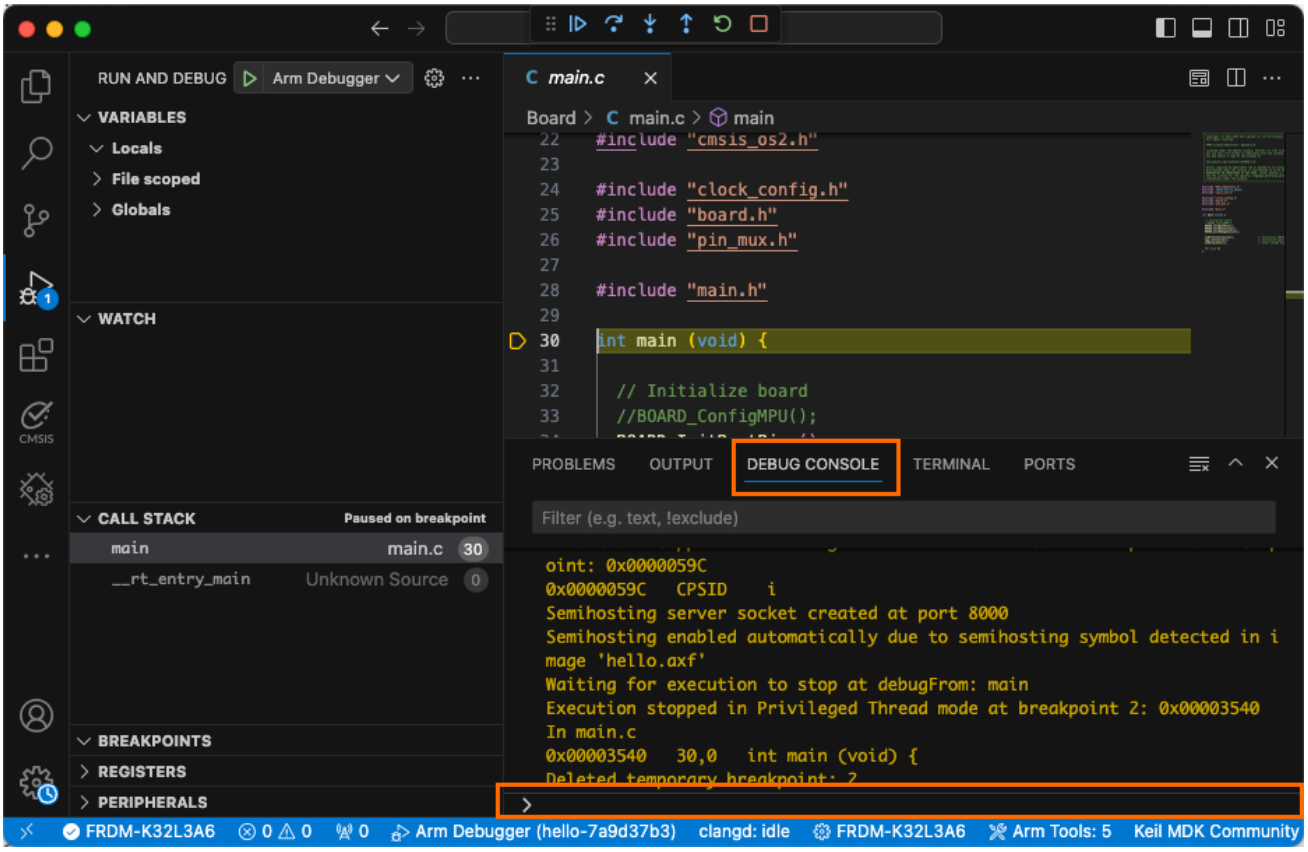
- Add function breakpoints with
- Activate or deactivate all breakpoints with

7.2.9 Use the Debug Console

The **Debug Console** shows the debugging output of your project. The console 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 7-7: Debug Console



7.2.9.1 Run Arm Debugger commands

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

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.

7.2.9.2 Use expressions

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

Use `$expr:<expression>` in the **Debug Console** prompt, where `<expression>` can include the following information:

- program symbols or register symbols

- 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 expression returns `0`, because the result is false.

7.2.10 Scope resolution operator

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

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 to start the debugger with the **Debug Configuration** visual editor, 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.

7.2.11 Next steps

To learn more about the debug features available in Visual Studio Code, see the Visual Studio Code documentation.

7.3 Work with scripts

Use scripts to customize your debugging workflows and automate tasks.

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

You can write advanced scripts or use the Jython templates available, and then run the scripts from the **Debug Console**. Alternatively, use the `"targetInitScript"` and `"debugInitScript"` configuration options to call the scripts from the `tasks.json` or `launch.json` files in your project.

7.3.1 Prerequisites

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. To install a Python interpreter, see the explanations provided in the Visual Studio Code documentation.

Note

macOS does not support the system installation of Python. We recommend that you use a package management system like Homebrew.

We recommend that you install the Microsoft Python extension in Visual Studio Code for rich Python language support. 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.

7.3.2 Use advanced scripts or the default Jython templates

Write your own Jython scripts or use the default templates provided to help you get started, then run the scripts.

Procedure

You can either:

- Create a Jython file with the `.py` extension. See About Jython scripts and Jython script concepts and interfaces in the Arm Development Studio User Guide for more details.
- Use one of the templates provided.

a. Go to the **File** menu. Select **New File...**, and then select **Jython Template** in the drop-down list that opens at the top of the window.

b. Select either the **Basic template** or the **Advanced template** option in the drop-down list.

   The basic template contains module imports that are typically placed at the top of the Jython script.

   The advanced template acquires a connection to the target, loads an application, runs to the start of the application, and reads some register values.

c. Edit the file to add the scripting commands that you need.

d. Save the template in the same folder as your project.

Next steps

You can run Jython scripts from the **Debug Console** with the `source` command.

For example:

```
source myScripts\myFile.py        # Run a Jython script from file myFile.py.
```

Both relative and absolute paths are supported.

See the Arm Debugger Command Reference guide for more details.

You can also use the `"targetInitScript"` and `"debugInitScript"` configuration options to call scripts from the `tasks.json` or `launch.json` files in your project.

For example:

```
"targetInitScript": "myScripts\myFile.py"
```

Both relative and absolute paths are supported.

See Arm Debugger run configuration options and Arm Debugger debug configuration options - CMSIS use cases.

7.4 Arm Debugger extension settings

The Arm Debugger extension has the following settings.

Name	Description
Application Pattern	Regular expression to specify which files should be considered as application files by the `arm-debugger.getApplicationFile` command. The default is `**/*.{axf,elf}`.
Debugger Path	The path to the Arm Debugger executable. This setting overrides any path set automatically in your Arm Tools Environment.
Experimental Features	Option to enable preview features that have not reached production quality yet.
Logging Verbosity	Verbosity level of the debugger logging. This impacts the output printed in the **Output** tab (**View** > **Output**) for the **Arm Debugger** output channel. Arm Debugger also generates additional log files for each debug session if the **Logging Verbosity** setting is set to `debug`. The log files are named as follows: `armdbg-<date>_<time>.log`.
Pack Asset Url	URL to download CMSIS DFPs (Device Family Packs) from if they are not already available on your machine.

7.4.1 Access the settings

Access the Arm Debugger settings.

Procedure

1. Open the settings:
   - On Windows or Linux, go to **File** > **Preferences** > **Settings**.
   - On macOS, go to **Code** > **Settings** > **Settings**.
2. Go to the **Extensions** category and click **Arm Debugger**.

8. Activate your license to use Arm tools

MDK and the various development tools Arm provides are exclusively available through user-based licensing (UBL).

To use MDK and tools like Arm® Compiler, Arm Debugger, or Fixed Virtual Platforms (FVPs) in your toolchain, you must activate a license.

If you 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. To open the **Problems** tab, select **Problems** from the **View** menu.

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**: Switch to the Keil® MDK Community Edition license. You can use this license only for non-commercial projects.
    - **Activate or manage Arm licenses**: 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 activate your license with a license server.


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. After you activate a license, its name displays in the status bar. Click the license name to open the **Arm License Management Utility** window.

For further details, see Activate your product using an activation code and Activate your product using a license server.

The Backwards compatibility topic also explains how you can license older versions of Keil software using a product license that includes Keil MDK Professional.

8.1 Troubleshoot expired or cache-expired licenses

If you use a licensed tool with an expired or cache-expired license, a warning displays in the status bar. In addition, a pop-up message displays in the bottom right-hand corner.


Note

Cache-expired licenses happen when your local license could not be renewed. This issue might occur 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.

9. 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, then CMSIS-Toolbox is already available on your machine. For more details, see Get started with an example project.

The main command-line tools that CMSIS-Toolbox provides 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 `csolution`, `cpackget`, and `cbuildgen` tools 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.

9.1 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.

9.2 Support for packs

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

9.2.1 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, 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 interface.

9.2.2 Add private local packs

To work with a CMSIS-Pack that you created locally, use the `cpackget add` command from the **Terminal**. Then, reload Visual Studio Code so that the CMSIS Solution 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 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_PACK_ROOT` environment variable. You can check the environment variables of your system relevant for CMSIS-Toolbox with:

```
cbuild list environment
```

9.2.3 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 you can download 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.

9.2.4 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.

10. Known issues and troubleshooting

This section describes known issues with the Keil® Studio extensions and how to troubleshoot some common issues.

10.1 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.

10.2 Troubleshooting

This section provides solutions to some common issues you might experience when you use the extensions.

10.2.1 Build fails to find CMSIS-Toolbox and 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:

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

If the Environment Manager is not installed:

- Install the Environment Manager from the **Extensions** view using the **Install Environment Manager** option. Next, 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 process creates a `vcpkg-configuration.json` file that you can save for your project.
- Alternatively, use the **Open Installation Documentation** option to install CMSIS-Toolbox manually and add it to the PATH, or configure the path in the settings.

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

10.2.2 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 long path names in the default installation folder.

Solution

Enable long path support in your Windows settings. See Enable Long Paths in Windows 10, Version 1607, and Later for more details.

10.2.3 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 instead of the Arm linker, armlink, that is included with Arm® Compiler.

Solution

1. If you have installed the CMSIS Solution extension separately instead of using the Keil Studio Pack, follow the instructions for installing and setting up CMSIS-Toolbox. In particular, make sure that you set the `CMSIS_COMPILER_ROOT` environment variable correctly.

   You can check the environment variables of your system relevant for CMSIS-Toolbox with:

   ```
   cbuild list environment
   ```

   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.

10.2.4 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** on Windows, **System Information** on a Mac, or a Linux system utility tool like **hardinfo**, 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 because it claims all the boards with a DAPLink firmware by default. We recommend 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 right-click them both to uninstall 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. To build and run a project on your hardware or to debug a project, you must install udev rules.

  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. If you still encounter a permission issue when accessing the serial output, 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. This might happen if you are accessing a board or device with several instances of Visual Studio Code, or with different IDEs.
- 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.

10.2.5 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. Note that ST development boards and probes on Windows require extra drivers. You can download them from the ST site.
- For other WebUSB-enabled CMSIS-DAP firmware updates, 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.

11. Submit feedback

If you have suggestions or if you discover an issue with any of the Keil® Studio extensions, get in touch with us. Go to https://www.keil.arm.com/support and use the links in the **Keil Studio for VS Code** category.

Proprietary Notice

This document is protected by copyright and other related rights and the use or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm Limited ("Arm"). No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether the subject matter of this document infringes any third party patents.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant

export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Product and document information

Read the information in these sections to understand the release status of the product and documentation, and the conventions used in Arm documents.

Product status

All products and services provided by Arm require deliverables to be prepared and made available at different levels of completeness. The information in this document indicates the appropriate level of completeness for the associated deliverables.

Product completeness status

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

Revision history

These sections can help you understand how the document has changed over time.

Document release information

The Document history table gives the issue number and the released date for each released issue of this document.

Document history

Issue	Date	Confidentiality	Change
0000-18	6 November 2024	Non-Confidential	Updates
0000-17	10 October 2024	Non-Confidential	Updates
0000-16	25 July 2024	Non-Confidential	Updates
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

Issue	Date	Confidentiality	Change
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

Change history

For information about the functional changes to the Arm® Keil® Studio Visual Studio Code extensions, see the **Change Log** for each extension in Visual Studio Code. Click **Extensions** in the Visual Studio Code Activity Bar. Then select an extension and click **CHANGELOG**.

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	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.

Convention	Use
<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**.

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

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

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

This information is important and needs your attention.

This information might help you perform a task in an easier, better, or faster way.

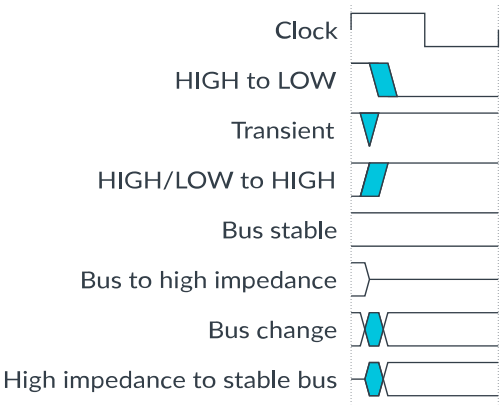
This information reminds you of something important relating to the current content.

Timing diagrams

The following figure explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.

Figure 1: Key to timing diagram conventions



Signals

The signal conventions are:

Signal level

The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

Lowercase n

At the start or end of a signal name, n denotes an active-LOW signal.

Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Arm product resources	Document ID	Confidentiality
Arm Keil Microcontroller Development Kit (MDK) Getting Started Guide	109350	Non-Confidential
Arm Keil Studio Cloud User Guide	102497	Non-Confidential
µVision User Guide	101407	Non-Confidential