Arm Keil Studio Visual Studio Code Extensions

User Guide

Non-Confidential

Issue 03

108029_0000_03_en

Arm Keil Studio Visual Studio Code Extensions
User Guide

Release information

Document history

Issue	Date	Confidentiality	Change
0000-01	13 July 2023	Non-Confidential	First release
0000-02	20 July 2023	Non-Confidential	Updates
0000-03	6 September 2023	Non-Confidential	Updates

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

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

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

Inclusive language commitment

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

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

Contents

1. Introduction

1.1 Conventions

The following subsections describe conventions used in Arm documents.

Glossary

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

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

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
monospace	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
monospace <u>underline</u>	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.
Caution	Recommendations. Not following these recommendations might lead to system failure or damage.
Warning	Requirements for the system. Not following these requirements might result in system failure or damage.
Danger	Requirements for the system. Not following these requirements will result in system failure or damage.
Note	An important piece of information that needs your attention.

Convention	Use
Tip	A useful tip that might make it easier, better or faster to perform a task.
Remember	A reminder of something important that relates to the information you are reading.

1.2 Other information

See the Arm website for other relevant information.

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

2. Extension packs and extensions

The Keil Studio Visual Studio Code extension packs provide a comprehensive software development environment for embedded systems and IoT software development on Arm-based microcontroller (MCU) devices.

The extension packs are installed and used with Visual Studio Code or Visual Studio Code for the Web. An extension pack is a set of related extensions that are installed together.

There is one extension pack for Visual Studio Code desktop (**Keil Studio Pack**), and one extension pack for Visual Studio Code for the Web (**Keil Studio Pack For Web**).

Each pack contains the following extensions:

Extension included in	Desktop pack	Web pack
Arm CMSIS csolution	√	x
Arm Device Manager	√	√
Arm Embedded Debugger	√	√
Arm Environment Manager	√	x
Arm Remote Build	√	√
Arm Virtual Hardware	√	√

Note

The **Arm CMSIS csolution** and **Arm Environment Manager** extensions are only available with Visual Studio Code desktop.

The extensions are:

- **Arm CMSIS csolution** (Identifier: `arm.cmsis-csolution`): This extension provides support for working with CMSIS solutions (csolution projects).
- **Arm Device Manager** (Identifier: `arm.device-manager`): This extension allows you to manage hardware connections for Arm Cortex-M based microcontrollers, development boards and debug probes.
- **Arm Embedded Debugger** (Identifier: `arm.embedded-debug`): This extension allows you to run and debug projects on Arm Cortex-M based microcontrollers, development boards and debug probes implementing the Microsoft Debug Adapter Protocol (DAP).
- **Arm Environment Manager** (Identifier: `arm.environment-manager`): This extension installs the tools you specify in a manifest file in your environment. For example, Arm Compiler for Embedded, CMSIS-Toolbox, CMake, and Ninja can be installed to work with CMSIS solutions.
- **Arm Remote Build** (Identifier: `arm.remote-build`): This extension allows you to undertake remote builds of projects for Arm Cortex-M based microcontrollers. The extension only works with standalone CMSIS projects (containing a single `.cprj` project file). An authentication is required to access the service.

- **Arm Virtual Hardware** (Identifier: `arm.virtual-hardware`): This extension allows you to manage Arm Virtual Hardware and run embedded applications on them. An authentication token is required to access the service.

Note

The **Arm Remote Build** and **Arm Virtual Hardware** extensions are experimental, and are not described in this guide.

The extensions contained in the packs can be also installed and used individually. We however 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 desktop pack Readme for more details.

3. Intended use cases for the extensions

Here are the intended use cases for the extensions:

- **Embedded and IoT software development using CMSIS-Packs and 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**: USB device management and embedded debug can be adapted to other project formats (for example CMake) and toolchains without additional overhead. This use case requires familiarity with Visual Studio Code to configure tasks. See the individual extensions for more details.

4. Get started with an example project

Quickly set up your environment and start working with an example.

We recommend installing the **Keil Studio Pack** in Visual Studio Code desktop as explained in the Readme. The pack installs all the Keil Studio extensions, including the **Arm Environment Manager**, as well as the **Red Hat YAML**, **Microsoft C/C++**, and **Microsoft C/C++ Themes** extensions.

Then:

- Do the setup using the Blinky_FRDM-K32L3A6 csolution project available from keil.arm.com (recommended).
- Download a Keil µVision `*.uvprojx` project from the website and convert it to a csolution (alternative).

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

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

The tools installed by default are:

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

Finalize the set up of your development environment. If you do not want to use **Microsoft C/C++** and **Microsoft C/C++ Themes**, you can install and set up the clangd extension instead to add smart features to your editor.

When you are ready:

- Build the Blinky_FRDM-K32L3A6 example project.
- Explore what you can do with the CMSIS csolution extension: set a context, look at the Solution outline, manage the software components of the solution.
- Connect your board and run the example on the board.
- Start a debug session.
- Check the serial output.

4.1 Import the Blinky_FRDM-K32L3A6 example

Import the recommended csolution example in Visual Studio Code. Alternatively, you can download a zip file that contains the csolution.

Procedure

1. Go to keil.arm.com.
2. Click the **Hardware** menu and select **Boards**.
3. Search for the FRDM-K32L3A6 board and click the **Results** box.
4. Find the Blinky project that is available in the **Projects** tab.
   The `Keil Studio` compatibility label indicates that the example is compatible with Keil Studio Cloud and the Keil Studio Visual Studio Code extensions.
5. Hover over the **Get Project** button, then click **Open in Keil Studio for VS Code** to import the csolution example.
6. Click the **Open Visual Studio Code** button in the "Open Visual Studio Code?" pop-up that opens at the top of your browser window.
7. Click the **Open** button in the "Allow an extension to open this URI" pop-up that opens in Visual Studio Code.
8. Choose a folder to import the project and click the **Select as Unzip Destination** button.
9. Click the **Open** button in the "Would you like to open the unzipped folder?" pop-up.
   If there are missing CMSIS-Packs, a pop-up displays in the bottom right-hand corner with the following message "Solution Blinky requires some packs that are not installed".
10. Click **Install**.
    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 desktop pack, a pop-up displays in the bottom right-hand corner with the message "Activate license for Arm tools?". See Activate your license to use Arm tools for more details on licensing.
11. Click the **Explorer** icon .
    A `vcpkg-configuration.json` is available. You do not need to do anything to install the tools. Microsoft vcpkg and the **Arm Environment Manager** extension take care of the setup. See Tools installation with Microsoft vcpkg.

    A `tasks.json` and `launch.json` files 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` for debug.

4.2 Download and convert a Keil uVision example

Download a Keil µVision `*.uvprojx` project from keil.arm.com and convert it to a csolution. Note that the conversion does not work with Arm Compiler 5 projects. You can download Arm Compiler 5 projects from the website, but you cannot use them with the extensions. Only Arm Compiler 6 projects can be converted. As a workaround, you can update Arm Compiler 5 projects to Arm Compiler 6 in Keil µVision, then convert the projects to csolutions in Visual Studio Code.

Procedure

1. Go to keil.arm.com.
2. Connect your board over USB and click **Detect hardware using WebUSB** 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, then click **Connect**.
4. Click the **Board** link in the pop-up that displays in the bottom right-hand corner.
   This takes you to the page for the board. Example projects are available in the **Projects** tab.
5. Look for an example with a `µVision` compatibility label.
6. Hover over the **Get Project** button for the project you want to use and click **Download zip** to download the Keil µVision `*.uvprojx` example.
7. Unzip the example and open the folder in Visual Studio Code.
8. A pop-up displays in the bottom right-hand corner with the following message "Convert µVision project [project-name].uvprojx to csolution?".
9. Click **Convert**.
   The conversion starts immediately.

   Alternatively, you can right-click the `*.uvprojx` and select **Convert µVision project to csolution** from the **Explorer**.

   You can also run the **CMSIS: Convert µVision project to csolution** command from the Command Palette. In that case, select the `*.uvprojx` that you want to convert on your machine and click **Select**.
10. Check the **OUTPUT** tab (**View** > **Output**). Conversion messages are logged under the **µVision to Csolution Conversion** category.
    If there are missing CMSIS-Packs, a pop-up displays in the bottom right-hand corner with the following message "Solution [solution-name] requires some packs that are not installed".
11. Click **Install**.
    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 desktop pack, a pop-up displays in the bottom right-hand corner with the message "Activate license for Arm tools?". See Activate your license to use Arm tools for more details on licensing.
12. Click the **Explorer** icon .
    The `*.cproject.yml` and `*.csolution.yml` files are available next to the `*.uvprojx`.

    A `vcpkg-configuration.json` is available.

    A `tasks.json` and `launch.json` files 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` for debug.

4.3 Tools installation with Microsoft vcpkg

Arm uses Microsoft vcpkg to set up your environment. Microsoft vcpkg works in combination with the **Arm Environment Manager** extension installed with the desktop pack for the set up.

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

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

4.3.1 Check the tools installed with Microsoft vcpkg

The `vcpkg-configuration.json` manifest file instructs Microsoft vcpkg to install the artifacts required to build the csolution example project. If you open the manifest file, you can see for example:

```
"requires": {
  "arm:tools/open-cmsis-pack/cmsis-toolbox": "^2.0.0-0",
  "arm:compilers/arm/armclang": "^6.20.0",
  "microsoft:tools/kitware/cmake": "^3.25.2",
  "microsoft:tools/ninja-build/ninja": "^1.10.2"
}
```

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

Go to the **OUTPUT** tab (**View** > **Output**) and select the **vcpkg** category in the drop-down list to see what has been installed. By default, Microsoft vcpkg installs the tools in the Visual Studio Code application directory.

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

4.4 Finalize the set up of your development environment

To finalize the set up of your development environment:

- Configure an HTTP proxy. This step is only required if you are working behind an HTTP proxy.
- The desktop pack installs all the Keil Studio extensions as well as the **Arm Environment Manager**, **Red Hat YAML**, **Microsoft C/C++** and **Microsoft C/C++ Themes** extensions. If you do not want to use the **Microsoft C/C++** and **Themes** extensions, you can disable them in Visual Studio Code and install and set up the clangd extension as an alternative.

4.4.1 Configure an HTTP proxy (optional)

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

- `HTTP_PROXY`: Set to the proxy used for HTTP requests.
- `HTTPS_PROXY`: Set to the proxy used for HTTPS requests.
- `NO_PROXY`: Set to include at least `localhost,127.0.0.1` to disable the proxy for internal traffic, which is required for the extension to work correctly.

4.4.2 clangd (alternative)

Install the **clangd** extension. Similarly to the **Microsoft C/C++** and **Microsoft C/C++ Themes** extensions, **clangd** adds smart features such as code completion, compile errors, go-to-definition and more to your editor.

Note

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

There is no extra setup needed once **clangd** has been installed. The **Arm CMSIS csolution** extension generates a `compile_commands.json` file for each project in a solution whenever a csolution file changes or when you change the context of a solution (**Target** and **Build** types). A `.clangd` file is kept up to date for each project in the solution. The `.clangd` file is used by the **clangd** extension to locate the `compile_commands.json` files and enable IntelliSense. See the clangd documentation for more details.

You can turn off the automatic generation of the `.clangd` file and `compile_commands.json` file.

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

4.5 Build the example project

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

     A **Rebuild** option is also available in the right-click menu. This option cleans output directories before building the project.
- Using the **Build** button:
  a. Click the **CMSIS** icon in the Activity Bar.
  b. Click the **Build** button in the **ACTIONS** panel.

     A **Clean Build** option is also available when you click the arrow next to **Build**. **Clean Build** is the same as **Rebuild** in the right-click menu.

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

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

4.6 Set a context for your csolution

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

The Blinky_FRDM-K32L3A6 example has just one project and one target type `FRDM-K32L3A6`. You can choose between `Debug` or `Release` for the build type.

Read Set a context for your csolution for more details.

4.7 Look at the Solution outline

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

Read Use the Solution outline for more details.

4.8 Manage software components

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

Read Manage software components for more details.

4.9 Connect your board

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

Procedure

1. Click the **Device Manager** icon in the Activity Bar to open the **Arm Device Manager** extension.
2. Connect your board to your computer over USB. In our example we use the FRDM-K32L3A6 board from NXP.

   The board is detected and a pop-up message displays.
3. Click **OK** in the pop-up message to use the hardware.

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

4.10 Run the csolution on your board

Run the csolution project on your board.

Procedure

1. Click the **CMSIS** icon in the Activity Bar.
2. Click the **Run** button Run in the **ACTIONS** panel.
   You can configure a run task in a tasks.json file to customise the behaviour of the run button. A `tasks.json` is provided for all the examples available on keil.arm.com. See Run your project on your hardware for more details.
3. As we are using a FRDM-K32L3A6 board, a device with multiple cores, you must select the appropriate processor for your project in the **Select a processor** drop-down list that displays at the top of the window. Select **cm4**.
   The project is run on the board.
4. Check the **TERMINAL** tab.

4.11 Start a debug session

Start a debug session.

Procedure

1. Click the **CMSIS** icon in the Activity Bar.
2. Click the **Debug** button Debug in the **ACTIONS** panel.

You can configure a launch configuration in a launch.json file to customise the behaviour of the debug button. A `launch.json` is provided for all the examples available on keil.arm.com. See Debug your project for more details.

3. Select the appropriate processor for your project in the **Select a processor** drop-down list that displays at the top of the window. Select **cm4**.
   The **RUN AND DEBUG** view displays and the debug session starts. The debugger stops at the function “main” of your project.
4. Check the **DEBUG CONSOLE** tab to see the debugging output.

Next steps

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

4.12 Check the serial output of your board

The serial output shows the output of your board. The serial output can be used as a debugging tool or to communicate directly with your board.

Procedure

1. Click the **CMSIS** icon in the Activity Bar.
2. Click the **Open Serial** button in the **ACTIONS** panel.
3. Select a baud rate of 115200 for your FRDM-K32L3A6 board in the drop-down list that opens at the top of the window. The baud rate you select must be the same as the baud rate of the project.
   The serial output displays in the **TERMINAL** tab.

5. Arm CMSIS csolution extension

The **Arm CMSIS csolution** extension provides support for working with CMSIS solutions (csolution projects).

With the **CMSIS csolution** extension, you can:

- Set a context for your csolution.
- Use the Solution outline.
- Manage software components.

You can also:

- Install missing CMSIS-Packs.
- Configure a build task.
- Convert a Keil µVision project to a csolution project.

5.1 Set a context for your csolution

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

Procedure

1. Click the **CMSIS** icon in the Activity Bar to open the **CMSIS** view.
2. Look at the available contexts for the csolution in the **CONTEXT** panel. You can change the target type (build target) and build configuration.
   - **Active Solution**: The name of the active csolution, `Blinky` (`Blinky.csolution.yml`).
   - **Target Type**: The build target `FRDM-K32L3A6`. Note that for this example you can only select `FRDM-K32L3A6`. Some examples are compatible with Arm Virtual Hardware (AVH) targets as well, so you can have more options in the drop-down list in that case. For more details on AVH, read the product overview.
   - **Build Type**: The build configuration `Debug` or `Release`. A build configuration adds the flexibility to configure each target type towards a specific testing. Use `Debug` for a full debug build of the software for interactive debug, or `Release` for the final code deployment to the systems. Note that you can create your own build types as required by your application.
   - **Project**: The name of the cproject, `FRDM-K32L3A6` (`FRDM-K32L3A6.cproject.yml`). If you have multiple projects in your solution, you can select the active one here.
3. Click the **Explorer** icon and open the `Blinky.csolution.yml` and `FRDM-K32L3A6.cproject.yml` files. YAML syntax support helps you with editing.
4. Go the **PROBLEMS** tab and check for errors.
5. Open the `main.c` file and check the IntelliSense features available. Read the Visual Studio Code documentation on IntelliSense to find out about the different features.

Next steps

A `*.cprj` file is generated automatically for the context selected in the **CONTEXT** panel each time you update the `*.csolution.yml` file.

You can turn off the automatic generation of `cprj` files. Note that this step is optional.

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

5.2 Use the Solution outline

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

Click the **CMSIS** icon in the Activity Bar to open the **CMSIS** view. The **SOLUTION** outline displays under the **CONTEXT** panel.

The **SOLUTION** outline shows:

- **Target types**: The active state of all the target types (build targets) in the active solution. You can have physical boards (for example, `FRDM-K32L3A6` with the Blinky_FRDM-K32L3A6 example) and Arm Virtual Hardware targets (for example, `AVH_MPS3_Corstone-300`) defined in your solution. Some attributes like `Core` (for example, `Core: Cortex-M4`) are derived from the target types.
- **Build types**: The build configurations. For example, `Debug` or `Release`. Details that can be included for a given configuration are:
    - `Compiler`: The compiler used for each configuration.
    - `Misc`: Miscellaneaous compiler-specific controls.
        - `for-compiler`: If there are several compilers defined in a solution, the compiler or compilers for which the controls defined apply.
        - `C`: List of C flags.
        - `CPP`: List of CPP flags.
        - `C-CPP`: List of C and CPP flags.
        - `ASM`: List of ASM flags.
        - `Link`: List of linker flags.
        - `Link-C`: List of linker flags for pure C project.
        - `Link-CPP`: List of linker flags for project with C++ files.
        - `Library`: List of linker flags for libraries handling.
        - `Lib`: List of Library Manager or Archiver flags.
    - `Debug`: Generic control for the generation of debug information. The values are `on` or `off`.

    - `Optimize`: Generic optimize levels for code generation. The values are `size`, `speed`, `balanced`, `none`.
- **Packs**: Packs included in the solution sorted by vendors and pack names.
- **Projects**: cprojects included in the solution. Details that can be included for a given cproject are:
    - `Groups`: Groups are a way to structure code files into logical blocks.
    - `Components`: All the software components selected for the cproject. Components are sorted by component class (Cclass).
    - `Layers`: The software layers defined for the cproject. A layer is a set of source files and pre-configured software components that can be shared across multiple projects. Layers are defined in `*.clayer.yml` files. The software components used by each layer in the cproject appear in the tree view.

When you hover over the **SOLUTION** label, you can choose the following actions:

- **Collapse All**: Click the **Collapse All** icon to close all the entries in the outline.
- **Open csolution file**: Click the **Open csolution file** icon to open the corresponding `csolution.yml` file.

When you hover over a project under the **Projects** entry, you can click the **Open file** icon to open the corresponding `cproject.yml` file.

An **Open file** icon is also available for each layer.

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

5.3 Manage software components

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

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

You can also:

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

5.3.1 Open the Software Components view

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

Procedure

1. Click the **CMSIS** icon in the Activity Bar to open the **Arm CMSIS csolution** extension.
2. In the **CONTEXT** panel, click the **Manage software components** icon.

Results

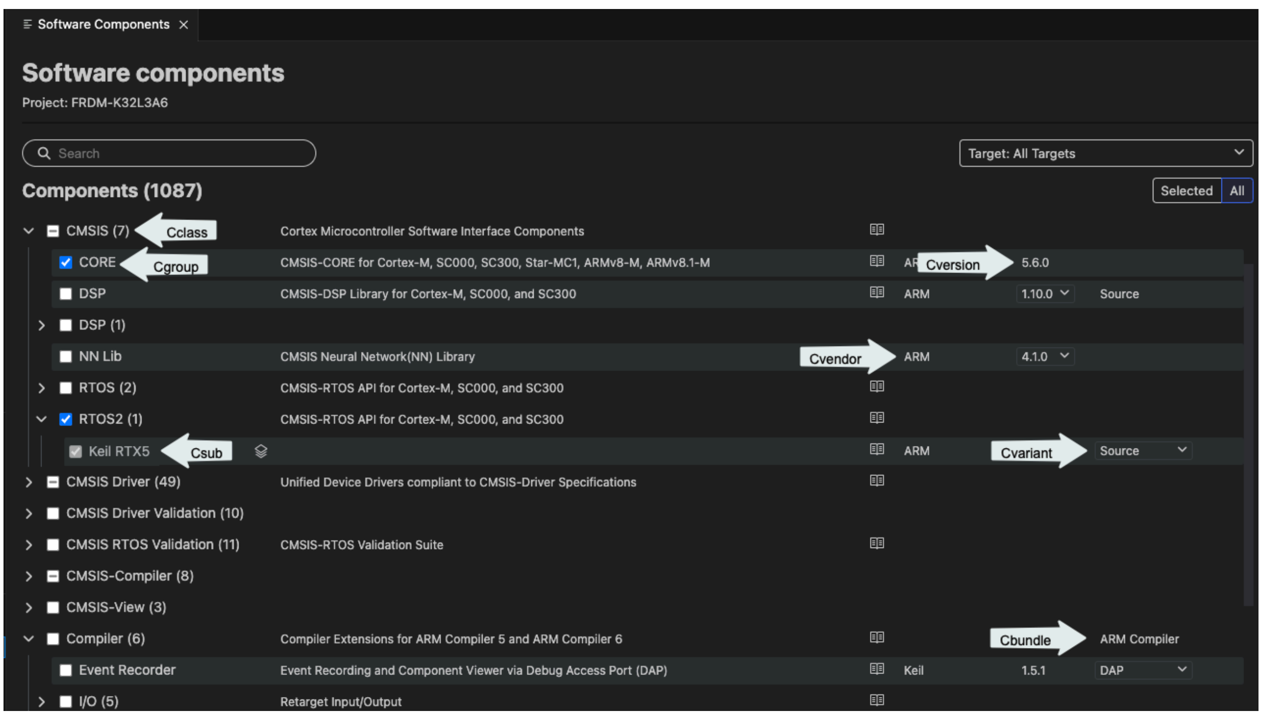
The **Software Components** view opens.

The default view displays the components included in the active project only (**Selected** toggle button). If you click the **All** toggle button, all the components available for use display.

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

With the **Target** drop-down list, you can select components for the different target types you have in your solution or for all the target types at once.

Figure 5-1: The 'Software Components' view showing all the components that are available for use



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

- Component class (Cclass): A top-level component name. For example: **CMSIS**.

- Component group (Cgroup): A component group name. For example: **CORE** for the **CMSIS** component class.
- Component version (Cversion): The version number of the software component.

Optionally, a software component might have these additional attributes:

- Component sub-group (Csub): A component sub-group 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 is typically used when the same implementation has multiple top-level configurations, like **Source** 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 files associated.

Documentation links are available for some components at the class, group, or sub-group level. Click the book icon of a component to open the related documentation.

5.3.2 Modify the software components in your project

You can add components from all the packs available. It is not limited to the packs that are already selected for a given project.

Procedure

1. Click the **All** toggle button to display all the components available.
2. Select a specific target type in the **Target** drop-down list or, if you want to modify all the target types at once, select **All Targets**. For the Blinky_FRDM-K32L3A6 example, there is just one target.
3. 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.
4. 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.
5. If there are validation issues, hover over the issues in the **Validation** panel to get more details. You can click the proposed fixes to find the components in the list. In some cases, you may

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 message “Component’s pack is not included in your solution” displays in the **Validation** panel. An error also displays in the **PROBLEMS** view. See Install missing CMSIS-Packs to know how to install CMSIS-Packs.

There can be other cases such as:

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

5.3.3 Undo changes

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

5.4 CMSIS-Packs

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

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

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

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

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

CMSIS-Packs are available for download from keil.arm.com.

5.5 Install missing CMSIS-Packs

Install the missing CMSIS-Packs for your csolution.

Procedure

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

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

5.6 Configure a build task

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

With the **Arm CMSIS csolution** extension, you can configure a build task using the `tasks.json` file to build your projects. When you run the build task, the extension executes `cbuild` with the options you defined.

Note

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

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

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.

```
{
    "tasks": [
        {
            "label": "CMSIS Build",
            "type": "cmsis-csolution.build",
            "solution": "${command:cmsis-csolution.getSolutionPath}",
            "project": "${command:cmsis-csolution.getProjectPath}",
            "buildType": "${command:cmsis-csolution.getBuildType}",
            "targetType": "${command:cmsis-csolution.getTargetType}",
```

```
                "problemMatcher": [],
                "group": {
                    "kind": "build",
                    "isDefault": true
                }
            }
        ]
}
```

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.

5.7 Convert a Keil uVision project to a csolution project

You can convert any Keil µVision project to a csolution project from the **Arm CMSIS csolution** extension. Note that the conversion does not work with Arm Compiler 5 projects. You can download Arm Compiler 5 projects from the website, but you cannot use them with the extensions. Only Arm Compiler 6 projects can be converted. As a workaround, you can update Arm Compiler 5 projects to Arm Compiler 6 in Keil µVision, then convert the projects to csolutions in Visual Studio Code.

Procedure

1. Open the project that contains the `*.uvprojx` you want to convert in Visual Studio Code.
2. A pop-up displays in the bottom right-hand corner with the following message "Convert µVision project [project-name].uvprojx to csolution?".
3. Click **Convert**.
   The conversion starts immediately.

   Alternatively, you can right-click the `*.uvprojx` and select **Convert µVision project to csolution** from the **Explorer**.

   You can also run the **CMSIS: Convert µVision project to csolution** command from the Command Palette. In that case, select the `*.uvprojx` that you want to convert on your machine and click **Select**.
4. Check the **OUTPUT** tab (**View** > **Output**). Conversion messages are logged under the **µVision to Csolution Conversion** category.
   The `*.cproject.yml` and `*.csolution.yml` files are available in the folder where the `*.uvprojx` is stored.

6. Arm Device Manager extension

Look at the hardware supported with the Keil Studio extensions.

Then, manage your hardware with the **Device Manager** extension:

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

6.1 Supported hardware

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

Here are the supported debug probes.

6.1.2.1 WebUSB-enabled CMSIS-DAP debug probes

The extensions support debug probes that implement the CMSIS-DAP protocol. See the CMSIS-DAP documentation for general information.

Such implementations are for example:

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

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

Describes how to connect your hardware for the first time.

Procedure

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

   Alternatively, you can click the **Add Device** button 

   and select your hardware in the drop-down list that displays at the top of the window.

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

Next steps

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

6.3 Edit your hardware

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

Procedure

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

6.4 Open a serial monitor

Open a serial monitor.

Procedure

1. Hover over the hardware for which you want to open a serial monitor and click the **Open Serial** icon .
   A drop-down list displays at the top of the window where you can select a baud rate (the data rate in bits per second between your computer and your hardware). To view the output of your hardware correctly, you must select an appropriate baud rate. The baud rate 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 Embedded Debugger extension

Run a project on your hardware and start a debug session with the **Embedded Debugger** extension:

- Run your project on your hardware.
- Debug your project.

Note

As mentioned in Get started with an example project, the examples provided on keil.arm.com are shipped with a `tasks.json` and a `launch.json` files that already contain some configuration to run your project and undertake debugging. You can modify the default configuration if needed.

7.1 Run your project on your hardware

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

You must first configure a task to be able to run a project on your hardware. The task transfers the binary into the appropriate memory locations on the hardware's flash memory.

There are two tasks available:

- **Flash Device**: Use this task for CMSIS-DAP (such as Nu-Link2 and ULINKplus) and ST-Link hardware. The CMSIS-Packs used in your project control the flash download.
- **Flash Device (DAPLink)**: Use this task for DAPLink hardware. The DAPLink firmware takes care of the flash download.

Procedure

1. In Visual Studio Code, go to **Terminal** > **Configure Tasks...**.
2. In the drop-down list that opens at the top of the window, select **embedded-debug.flash:Flash Device** or **embedded-debug.daplink-flash:Flash Device (DAPLink)**.

   A `tasks.json` file opens with some default configuration.

   Default configuration for **Flash Device**:

```
{
    "label": "Flash Device",
    "type": "embedded-debug.flash",
    "program": "${command:embedded-debug.getApplicationFile}",
    "serialNumber": "${command:device-manager.getSerialNumber}",
```

```
            "cmsisPack": "${command:device-manager.getDevicePack}",
            "problemMatcher": [],
            "dependsOn": "CMSIS Build"
        }
```

Default configuration for **Flash Device (DAPLink)**:

```
        {
        "type": "embedded-debug.daplink-flash",
        "serialNumber": "${command:device-manager.getSerialNumber}",
        "program": "${command:embedded-debug.getBinaryFile}",
        "problemMatcher": [],
        "label": "embedded-debug.daplink-flash: Flash Device (DAPLink)"
        }
```

3. Save the `tasks.json` file.

7.1.2 Override or extend the default tasks configuration options

You can override or extend the default configuration options. See the Flash configuration options for CMSIS-DAP and ST-Link hardware (Flash Device) and Flash configuration options for DAPLink hardware (Flash Device DAPLink).

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

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

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

Alternatively, these can be specified 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 form `<vendor>::<pack>@<version>`. Note that this triggers an automatic download of the CMSIS-Pack:

```
        {
```

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

7.1.2.1 Flash configuration options for CMSIS-DAP and ST-Link hardware (Flash Device)

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

Configuration option	Description
`"cmsisPack"`	Path (file or url) to a DFP (Device Family Pack) CMSIS-Pack for your hardware. Command available: device-manager.getDevicePack - Gets the CMSIS-Pack for the selected device.
`"connectMode"`	Connection mode. Possible values: auto (debugger decides), haltOnConnect (halts for any reset before running), underReset (holds external NRST line asserted), preReset (pre-reset using NRST), running (connects to running target without altering state). Default: auto.
`"dbgconf"`	Path (file or url) to a dbgconf file.
`"deviceName"`	CMSIS-Pack device name. Command available: device-manager.getDeviceName - Gets the device name from the DFP (Device Family Pack) of the selected device.
`"eraseMode"`	Type of flash erase to use. Possible values: sectors (erase only sectors to be programmed), full (erase full chip), none (skip flash erase). Default: sectors.
`"flm"` or `"flms"`	Path(s) (file or url) to an flm file or flm files.
`"openSerial"`	Baud rate for connected device. Opens the serial output of the device in the **TERMINAL** tab with the baud rate specified.
`"pdsc"`	Path (file or url) to a pdsc file.
`"processorName"`	CMSIS-Pack processor name for multi-core devices.
`"program"` or `"programs"`	Path(s) (file or url) to the project(s) to use. Command available: embedded-debug.getApplicationFile - Returns an Axf or Elf file used for CMSIS run and debug.
`"programFlash"`	Program code into flash. Default: true.
`"programMode"`	Mode to program an application to a target. Default: auto
`"resetAfterConnect"`	Resets the hardware after having acquired control of the CPU. Default: true.
`"resetMode"`	Type of reset to use. Possible values: auto (debugger decides), system (use ResetSystem sequence), hardware (use ResetHardware sequence), processor (use ResetProcessor sequence). Default: auto.
`"resetRun"`	Issue a hardware reset at end of flash download. Default: true.
`"sdf"`	Path (file or url) to an sdf file.
`"serialNumber"`	Serial number of the connected USB hardware to use. Command available: device-manager.getSerialNumber - Gets the serial number of the selected device.
`"targetAddress"`	Synonymous with serialNumber.
`"vendorName"`	CMSIS-Pack vendor name.
`"verifyFlash"`	Verify the contents downloaded to flash. Default: true.

7.1.2.2 Flash configuration options for DAPLink hardware (Flash Device DAPLink)

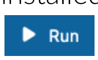
The extension provides the task options below. Other Visual Studio Code options are also available. Use the **Trigger Suggestions** command (**Ctrl+Space**) to see what is available and read the Visual Studio Code documentation on tasks, as well as the Schema for tasks.json page.

Configuration option	Description
`"openSerial"`	Baud rate for connected device. Opens the serial output of the device in the **TERMINAL** tab with the baud rate specified.
`"program"`	Path(s) (file or url) to the project(s) to use. Command available: embedded-debug.getBinaryFile - Returns a Bin or Hex file.
`"serialNumber"`	Serial number of the connected USB hardware to use. Command available: device-manager.getSerialNumber - Gets the serial number of the selected device.

7.1.3 Run your project

Run the project on your hardware.

Procedure

1. Check that your hardware is connected to your computer.
2. Select **Terminal** > **Run Task...** to run the project on your hardware.
3. In the drop-down list that opens at the top of the window, select the **embedded-debug.flash:Flash Device** task or the **embedded-debug.daplink-flash:Flash Device (DAPLink)** task.
   If you have installed the desktop pack, you can alternatively go to the **CMSIS** view and click the **Run** button 

 in the **ACTIONS** panel.
4. If you are using a device with multiple cores, you must select the appropriate processor for your project in the **Select a processor** drop-down list that displays at the top of the window.
5. Check the **Terminal** tab to verify that the project has run correctly.

7.2 Debug your project

Debug a project.

7.2.1 Add configuration

As for running a project, you must first add some launch configuration to be able to do debugging. Creating a launch configuration file allows you to configure and save debugging setup details. Visual Studio Code keeps debugging configuration information in a `launch.json` file.

Procedure

1. In Visual Studio Code, go to **Run** > **Add Configuration...**.
   A `launch.json` file opens.

2. Select **Arm: Embedded Debug** in the IntelliSense suggestions widget that opens.
   Some default configuration is added.

```
{
    "configurations": [
        {
            "name": "Embedded Debug",
            "type": "embedded-debug",
            "request": "launch",
            "serialNumber": "${command:device-manager.getSerialNumber}",
            "program": "${command:embedded-debug.getApplicationFile}",
            "cmsisPack": "${command:device-manager.getDevicePack}",
            "debugFrom": "main"
        }
    ]
}
```

3. Save the `launch.json` file.

7.2.2 Override or extend the default launch configuration options

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

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

7.2.2.1 Debug configuration options

The extension provides the task options below. Other Visual Studio Code options are also available. Use the **Trigger Suggestions** command (**Ctrl+Space**) to see what is available and read the Visual Studio Code documentation on tasks.

Configuration option	Description
`"cmsisPack"`	Path (file or url) to a DFP (Device Family Pack) CMSIS-Pack for your hardware. Command available: device-manager.getDevicePack - Gets the CMSIS-Pack for the selected device.
`"connectMode"`	Connection mode. Possible values: auto (debugger decides), haltOnConnect (halts for any reset before running), underReset (holds external NRST line asserted), preReset (pre-reset using NRST), running (connects to running target without altering state). Default: auto.
`"dbgconf"`	Path (file or url) to a dbgconf file.
`"debugFrom"`	The symbol the debugger will run to before debugging. Default: `"main"`.
`"deviceName"`	CMSIS-Pack device name. Command available: device-manager.getDeviceName - Gets the device name from the DFP (Device Family Pack) of the selected device.
`"pathMapping"`	A mapping of remote paths to local paths to resolve source files.
`"pdsc"`	Path (file or url) to a pdsc file.
`"processorName"`	CMSIS-Pack processor name for multi-core devices.
`"program"` or `"programs"`	Path(s) (file or url) to the project(s) to use. Commands available: embedded-debug.getBinaryFile: Returns a Bin or Hex file. embedded-debug.getApplicationFile: Returns an Axf or Elf file used for CMSIS run and debug.
`"programNames"`	Filename or filenames of the projects to be used. Only used for labelling.
`"resetAfterConnect"`	Resets the hardware after having acquired control of the CPU. Default: true.

Configuration option	Description
`"resetMode"`	Type of reset to use. Possible values: auto (debugger decides), system (use ResetSystem sequence), hardware (use ResetHardware sequence), processor (use ResetProcessor sequence). Default: auto.
`"sdf"`	Path (file or url) to an sdf file.
`"serialNumber"`	Serial number of the connected USB hardware to use. Command available: device-manager.getSerialNumber - Gets the serial number of the selected device.
`"svd"` or `"svdPath"`	Path (file or url) to an svd file.
`"targetAddress"`	Synonymous with serialNumber.
`"vendorName"`	CMSIS-Pack vendor name.
`"verifyApplication"`	Verify application against target memory for each application load operation in debug session. Default: true.

7.2.3 Debug

Start a debug session.

Procedure

1. Check that your device is connected to your computer.
2. Select **Run** > **Start Debugging**.
   If you have installed the desktop pack, you can alternatively go to the **CMSIS** view and click the **Debug** button 

   in the **ACTIONS** panel.

   A **Run & Debug** option is also available when you click the arrow next to **Debug**. This allows you to do both actions consecutively.
3. If you are using a device with multiple cores, you must select the appropriate processor for your project in the **Select a processor** drop-down list that displays at the top of the window.
   The **Run and Debug** view displays and the debug session starts. The debugger stops at the function “main” of your project.
4. Check the **Debug Console** tab to see the debugging output.

Next steps

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

8. Activate your license to use Arm tools

If you are using tools such as Arm Compiler, Arm Debugger, or Fixed Virtual Platforms in your toolchain, you must activate a license to be able to use those tools.

After you have installed the pack available for Visual Studio Code Desktop, **Keil Studio Pack**, a pop-up displays in the bottom right-hand corner.

Click **Activate**.

By default, this activates the Keil MDK Community Edition license. After activation, the Community license takes precedence over any existing licenses, including MDK and Flex licenses.

If you already have a commercial license, click **Don’t Ask Again** in the pop-up to ignore the Keil MDK Community Edition license activation.

To turn the licensing notifications off, you can also go to the **Keil Studio Pack** category in the settings and select the **Silence Licensing Notifications** checkbox.

9. Known issues and troubleshooting

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

9.1 Known issues

Here are the known issues.

Arm CMSIS csolution extension

The **Arm CMSIS csolution** 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.

Arm Embedded Debugger

The **Arm Embedded Debugger** extension has the following known issues:

- Support for the DWARF debugging standard is limited to version 4. Please make sure that your application is built with the appropriate settings.
- Variables and registers are read-only.
- Stack trace is limited if the debugger is halted in assembler source files.

9.2 Troubleshooting

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

9.2.1 Build fails to find toolchain

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

Solution

1. If you have installed the CMSIS csolution extension separately, not using the **Keil Studio Pack**, ensure you follow the instructions for installing and setting up CMSIS-Toolbox. In particular, ensure the `CMSIS_COMPILER_ROOT` environment variable is set correctly. Alternatively, you can install the **Keil Studio Pack** to benefit from an automated setup with Microsoft vcpkg.
2. Clean the solution. In particular, delete the `out` and `tmp` directories.
3. Run the build again.

9.2.2 Connected development board or debug probe not found

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

Solution

- Run **Device Manager** (Windows), **System Information** (Mac), or a Linux system utility tool like **hardinfo** (Linux) and check for warnings beside your hardware. Warnings can indicate that hardware drivers are not installed. If necessary, obtain and install the appropriate drivers for your hardware.
- On Windows: ST development boards and probes require extra drivers. You can download them from the ST site.
- On Windows: Check if you have an Mbed serial port driver installed on your machine. The Mbed serial port driver is required with Windows 7 only. Serial ports work out of the box with Windows 8.1 or newer. The Mbed serial port driver breaks native Windows functionality for updating drivers as it claims all the boards with a DAPLink firmware by default. It is recommended to 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 and find and delete the `mbedserial_x64.inf`, `mbedcomposite_x64.inf` drivers.

    ```
    pnputil /enum-drivers

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

    Then, connect your hardware using a USB cable and open the Windows Device Manager. In `Ports (COM & LPT)` and `Universal Serial Bus controllers`, find the `mbed` entries and uninstall both by right-clicking them. Finally, disconnect and reconnect your hardware.
  - Disable the Mbed serial port driver: Open the Windows Device Manager. In `Ports (COM & LPT)`, find the Mbed Serial Port. Right-click it and select **Properties**. Select the **Driver** tab and click the **Update Driver** button. Then click **Browse my computer for drivers** and then **Let me pick from a list of available drivers on my computer**. Select `USB Serial Device` instead of `mbed Serial Port`.
- On Linux: udev rules grant permission to access USB boards and devices. You must install udev rules to be able to build a project and run it on your hardware or debug a project.

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

  After installing the udev rules, your connected hardware is detectable in the **Device Manager** extension. You may still encounter a permission issue when accessing the serial output. If this is the case, run `sudo adduser "$USER" dialout` 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 may be claimed by other processes or tools. For example, if you are trying to access a board or device with several instances of Visual Studio Code, or with Visual Studio Code and another IDE.
- Activate the **Manage All Devices** setting. This allows you to select any USB hardware connected to your computer. By default, the **Device Manager** extension only gives you access 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.

9.2.3 Out-of-date firmware

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

Solution

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

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

Note

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

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

10. Submit feedback

Help us improve the Keil Studio extensions. To submit feedback, please open a bug report or a feature request:

- **Arm Keil Studio Packs** (Desktop or Web).
- **Arm CMSIS csolution**.
- **Arm Device Manager**.
- **Arm Embedded Debugger**.