



Arm[®] Keil[®] Microcontroller Development Kit (MDK)

v6

Getting Started Guide

Non-Confidential

Copyright © 2024 Arm Limited (or its affiliates).
All rights reserved.

Issue 03

109350_v6_03_en



Arm® Keil® Microcontroller Development Kit (MDK)

Getting Started Guide

Copyright © 2024 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0000-03	23 April 2024	Non-Confidential	Updates
0000-02	8 April 2024	Non-Confidential	Updates
0000-01	21 March 2024	Non-Confidential	First release

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. 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.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. Please follow Arm's trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

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

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

Inclusive language commitment

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

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

Contents

1. Introduction.....	7
1.1 Conventions.....	7
1.2 Other information.....	8
2. What is MDK?.....	9
2.1 A family of tools.....	9
2.2 CMSIS-Packs.....	10
2.3 Functional safety (FuSa).....	10
2.4 Debug adapters.....	10
2.5 MDK editions.....	10
2.6 License types.....	11
2.7 Download options.....	12
2.8 Access the MDK documentation.....	13
3. Tools.....	15
3.1 Keil Studio.....	15
3.1.1 Keil Studio Pack for Visual Studio Code.....	15
3.2 Keil µVision.....	16
3.3 Arm Compiler for Embedded.....	16
3.4 Arm Virtual Hardware.....	18
4. CMSIS components.....	21
4.1 CMSIS basic concepts.....	22
4.1.1 CMSIS-Pack.....	22
4.1.2 Software pack.....	23
4.1.3 Software component.....	23
4.1.4 CMSIS solutions.....	23
4.1.5 CMSIS projects.....	24
4.2 Overview of CMSIS software components.....	24
4.3 Overview of CMSIS base software components.....	25
4.3.1 CMSIS-Core.....	25
4.3.2 CMSIS-RTOS2.....	26
4.3.3 CMSIS-Driver.....	27

4.4 Overview of CMSIS extended software components.....	27
4.4.1 CMSIS-Compiler.....	28
4.4.2 CMSIS-View.....	28
4.4.3 CMSIS-DSP.....	29
4.4.4 CMSIS-NN.....	29
4.5 Overview of CMSIS tools.....	29
4.5.1 CMSIS-Stream.....	29
4.5.2 CMSIS-Toolbox.....	30
4.5.3 CMSIS-Zone.....	32
4.5.4 CMSIS-DAP.....	32
5. Other software components and packs.....	34
5.1 Product lifecycle management with software packs.....	34
5.2 Overview of additional software components.....	35
5.2.1 CMSIS-FreeRTOS.....	35
5.2.2 CMSIS-mbedTLS.....	36
5.2.3 Synchronous Data Stream (SDS) framework.....	36
5.2.4 Network component.....	37
5.2.5 File System component.....	39
5.2.6 USB component.....	41
5.2.7 IoT clients.....	42
5.2.8 Overview of open-source components.....	43
6. Create new applications.....	45
6.1 Create a new solution using the Keil Studio VS Code extensions.....	45
6.1.1 Create a solution.....	45
6.1.2 Add software components to your solution.....	46
6.1.3 Add the source code files to your solution.....	46
6.1.4 Configure virtual hardware.....	48
6.1.5 Build the solution.....	48
6.1.6 Run the solution.....	48
6.1.7 Debug the solution.....	49
7. Terminology.....	50

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
<i>italic</i>	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: <pre>MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2></pre>
SMALL CAPITALS	Terms that have specific technical meanings as defined in the <i>Arm® Glossary</i> . For example, IMPLEMENTATION DEFINED , IMPLEMENTATION SPECIFIC , UNKNOWN , and UNPREDICTABLE .



Caution

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



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 harming yourself.



This information is important and needs your attention.



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



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

1.2 Other information

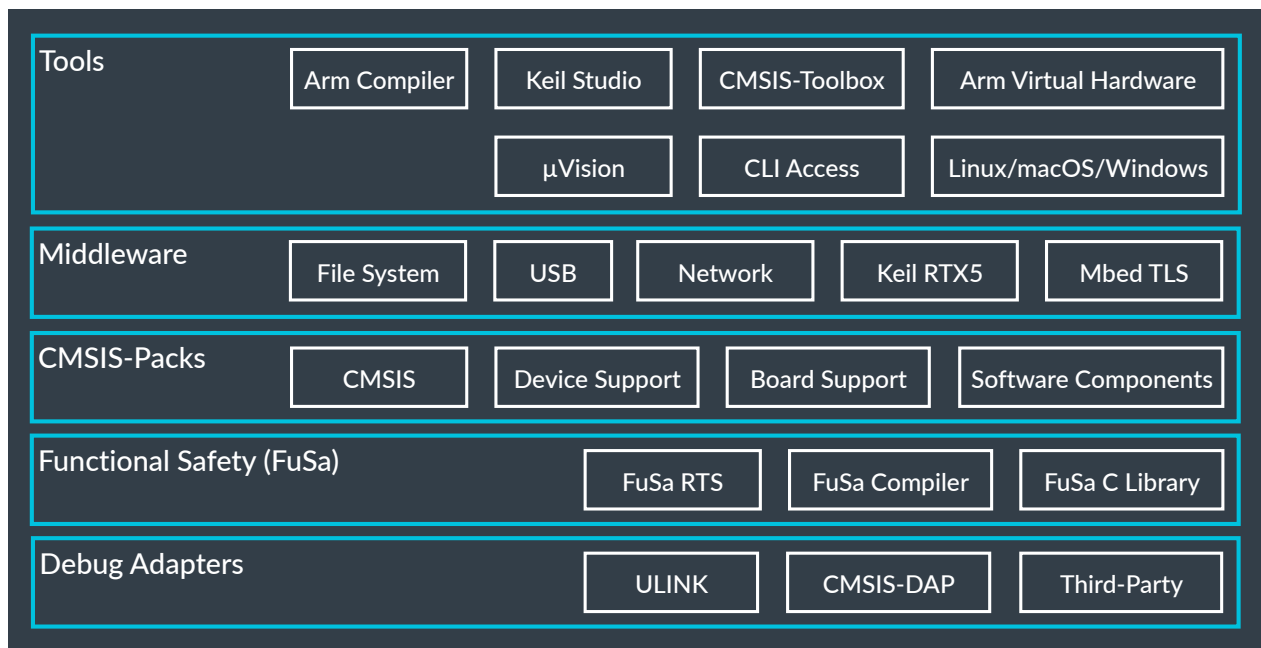
See the Arm website for other relevant information.

- [Arm® Developer.](#)
- [Arm® Documentation.](#)
- [Technical Support.](#)
- [Arm® Glossary.](#)

2. What is MDK?

Arm® Keil® Microcontroller Development Kit (MDK) is a collection of software tools for developing embedded applications based on Arm Cortex®-M and Ethos™-U processors. MDK makes software engineering easy and productive by offering you the flexibility to work with a CLI or an IDE (desktop-based or browser-based), or by deploying the tools into a continuous integration workflow.

Figure 2-1: MDK overview diagram



2.1 A family of tools

MDK includes:

- [Keil Studio](#)
- [Keil µVision®](#)
- [Arm Compiler for Embedded](#). Version 6 is based on the innovative LLVM and Clang technologies and supports the latest language standards, including C++17.
- [Arm Virtual Hardware \(AVH\)](#)

MDK uses development flows based on the [Common Microcontroller Software Interface Standard \(CMSIS\)](#). Embedded systems frequently require several years of product development, so MDK supports the entire product lifecycle from initiation to completion and maintenance.

MDK offers host support for Linux, macOS, and Windows.



Arm Virtual Hardware simulation models (Fixed Virtual Platform models, or FVPs) are currently not available on macOS, and μ Vision runs on Windows only.

2.2 CMSIS-Packs

[CMSIS-Packs](#) contain device and board support, software components, middleware, code templates, and example projects. You can add them to the tools at any time, which means that support for new devices and middleware updates are independent from the toolchain. The IDEs and CLI tools manage the software components that you can use as building blocks for the application.

2.3 Functional safety (FuSa)

The MDK-Professional edition includes components that you need for functional safety applications:

- [Arm Compiler for Embedded FuSa](#)
- A certified C library
- [FuSa Run-Time System \(RTS\)](#)

2.4 Debug adapters

MDK works with Arm's ULINK™ family of debug and trace adapters:

- [ULINKpro](#), a debug and trace unit that allows you to program, debug, and analyze your applications using its unique streaming trace technology.
- [ULINKplus](#), which combines isolated debug connection, power measurement, and I/O for test automation.
- [ULINK2](#).

You can also expand MDK with various third-party tools, starter kits, and debug adapters (such as ST-Link, JLink, and others).

2.5 MDK editions

MDK is available in the following editions:

- **MDK-Community**. For non-commercial use by evaluators, hobbyists, makers, academics, and students.

- **MDK-Essential.** For commercial development of Arm Cortex-M-based microcontroller projects.
- **MDK-Professional.** For professionals with functional safety (FuSa) requirements and the need for DevOps using simulation models. This all-in-one solution includes Arm Compiler for Embedded FuSa, and grants access to all Arm Virtual Hardware Fixed Virtual Platforms (FVPs).

Figure 2-2: MDK editions



The [product selector](#) gives an overview of the features enabled in each edition.

2.6 License types

Apart from the **MDK-Community** edition, all MDK editions require activation using a license code.

MDK supports [user-based licensing \(UBL\)](#), which binds the entitlement to use an Arm® product to the user. A user is entitled to use an Arm product license with no limits on concurrent usage, including using the same product on multiple devices. For example, you could use a single license with a service account to automatically build and test your products with Arm development tools on any number of devices.

There are different ways to get a license:

- Using an activation code provided by Arm or your license administrator
- Accessing a local license server managed by your license administrator

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

If you are using Keil Studio for Visual Studio Code, see [Activate your license to use Arm tools](#) in the Arm Keil Studio Visual Studio Code Extensions User Guide to open the **Arm License Management Utility** user interface and provide an activation code or use a license server.

If you are an administrator and you need to add products to your account on the Arm user-based licensing portal using a serial number, or you need to add licenses to existing products, watch the [Accessing the Arm License Portal video tutorial](#). More details are also available in the [User-based Licensing Administration Guide](#). To create activation codes, watch the [Cloud-based Licenses and Activation Codes video tutorial](#). See also the information available in the [User-based Licensing Administration Guide](#).

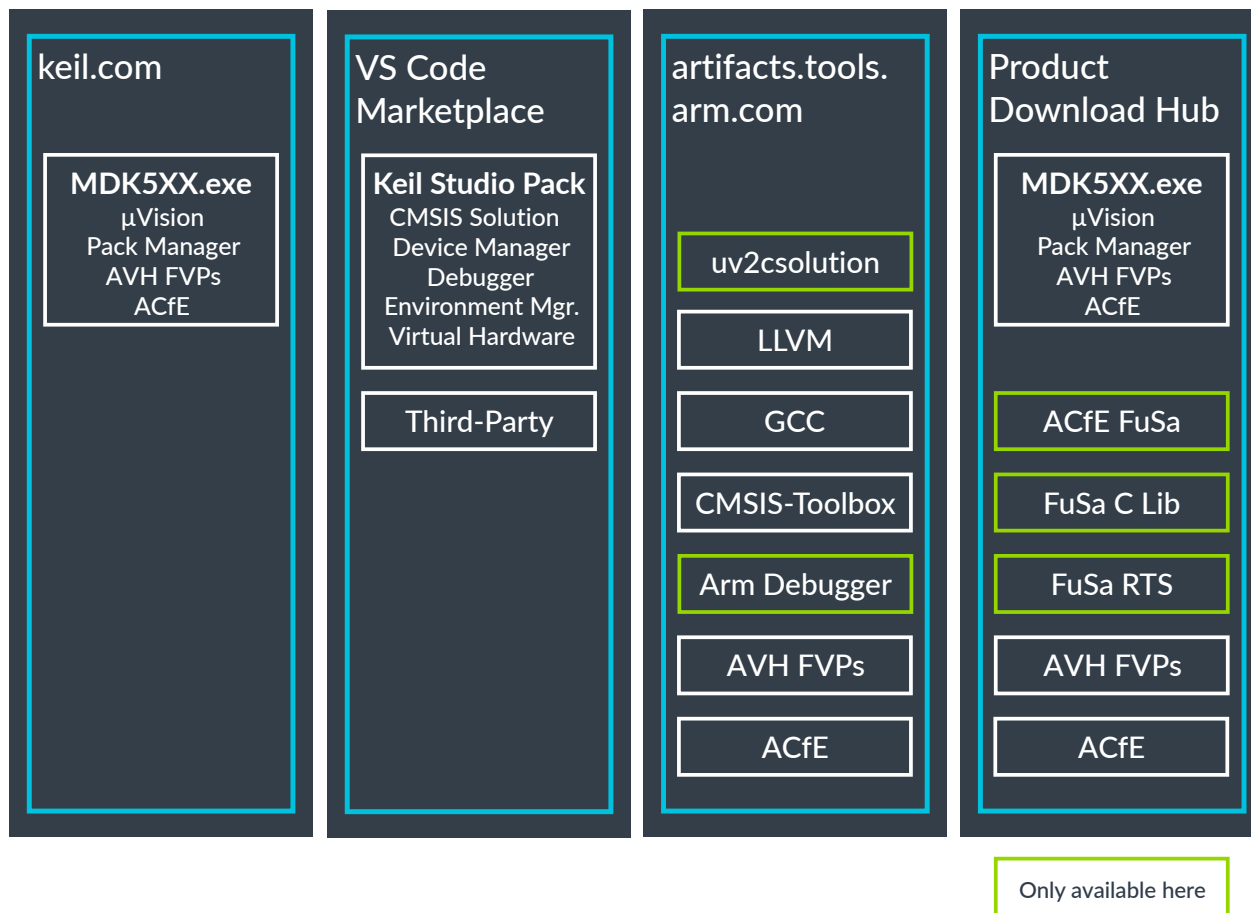
For more details on user-based licensing support and backwards compatibility, see the User-based licensing User Guide. The [Backwards compatibility](#) topic explains how you can license older

versions of MDK (MDK 5.36 and earlier), PK51, PK166, and DK251 using a product license that includes Keil MDK Professional.

2.7 Download options

MDK v6 contains various [tools](#) that you can install in different ways.

Figure 2-3: MDK v6 download options




μ Vision

The MDK5XX.exe installer contains the μ Vision® tool with Arm® Compiler for Embedded (ACfE) and the Arm Virtual Hardware-Fixed Virtual Platform (AVH-FVP) models. This installer is available on [keil.com](#) and the [Product Download Hub](#). See the [User's Guide](#) for more information on how to install the tool.

Keil Studio extensions

Keil® Studio is a [set of VS Code extensions](#) that you can download from the VS Code Marketplace.

1. In VS Code, click **Extensions**  in the Activity Bar.
2. Enter **Keil Studio Pack** in the search box.

3. In the search results, select **Arm Keil Studio Pack**.
4. Click **Install**.

When you are working with a CMSIS-based project, Keil Studio uses vcpkg to download additional tools (such as a toolchain and a debugger). These tools are served by [Artifactory](#).

Functional safety (FuSa) tools

MDK-Professional users can download the building blocks that Arm offers for functional safety applications. These tools are available only through the [Product Download Hub](#) that serves tools based on entitlement. The following tools are available:

- [Arm Compiler for Embedded FuSa](#): Arm's functional safety toolchain
- [FuSa RTS](#): the functional safety run-time system that offers software components for FuSa applications
- [FuSa C Lib](#): a C library certified for usage in FuSa applications

Other tools

The broadest set of tools is available through the new [Artifactory](#) repository manager. Artifactory serves automated downloads of tools, using either vcpkg or direct download through scripting. The following tools are available on Artifactory:

- [GCC](#): an open-source toolchain for Arm CPUs
- [Arm Compiler for Embedded](#): Arm's toolchain
- [LLVM Embedded Toolchain](#): an open-source toolchain for Arm CPUs
- [Arm Debugger](#): a command-line debug server for Arm CPUs
- [Arm Virtual Hardware](#): models for Cortex®-M based on Fast Models technology
- [uv2csolution](#): an MDK μ Vision project converter
- [Arm CMSIS-Toolbox](#): command-line tools for CMSIS projects

For a full list of all versions of these tools, see [Arm Tools Available in vcpkg](#).

2.8 Access the MDK documentation

MDK provides [documentation](#) for all of its components on [Arm Developer](#).

Get help

If you have suggestions or you have discovered an issue with any of the Keil® MDK products, please report them to us. Go to the [keil.arm.com support page](#) and use the links under **Get Help or Report Issues**.

Include your license code (if you have one) and product version when reporting a μ Vision® issue.

Online learning

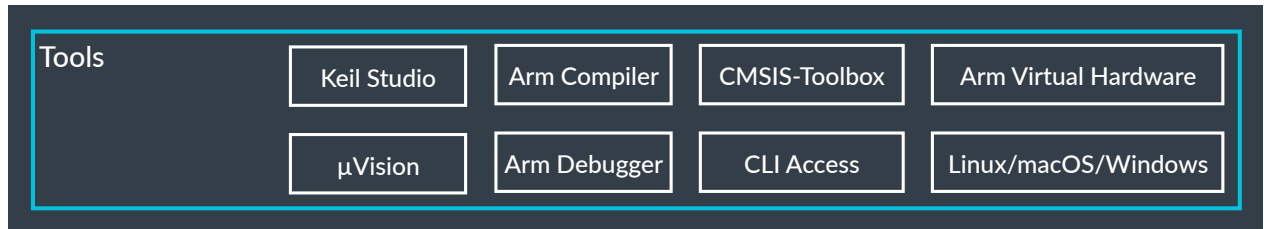
Our [Learning Paths](#) help you to learn more about the programming of Arm® Cortex®-based microcontrollers. The site contains tutorials for all levels of experience, from beginner to advanced.

Videos showing the tools and different aspects of software development are available at [Arm's Youtube channel](#).

3. Tools

Learn more about the software tools included in MDK v6.

Figure 3-1: Tools overview diagram



3.1 Keil Studio

Keil® Studio is a complete development tool for the evaluation and development of embedded, IoT, and machine learning software for Cortex®-M devices. It is available as a browser-based Integrated Development Environment (IDE) for development in the cloud, or as a set of extensions for desktop development with Visual Studio Code.

Keil Studio Cloud offers a cloud-hosted workspace for your code, comprehensive version control system integration, and a powerful C/C++ editor. You can:

- edit your projects from any computer, share them with colleagues, and export them for desktop usage
- compile projects using Arm® Compiler for Embedded
- run the projects directly on [supported development boards](#)
- debug from supported browsers (Chromium-based browsers) without having to install any software.

3.1.1 Keil Studio Pack for Visual Studio Code

The Arm Keil Studio Pack includes extensions that enable you to manage your [CMSIS](#) solutions (csolution projects), and to create, build, test, and debug embedded applications on your chosen hardware using Visual Studio Code.

For more information on available extensions, and to install the pack in Visual Studio Code, see [Arm Keil Studio Pack for Visual Studio Code](#). For information on how to set up your working environment and get started, see [Get started with an example project](#).

3.2 Keil μVision

Keil® μVision® is a Windows-based software development platform that integrates all the tools needed to develop embedded applications quickly and successfully. It combines a source code editor, a project manager for creating and maintaining your projects, and a Make tool for assembling, compiling, and linking your embedded applications.

μVision offers separate modes for building and debugging applications. You can debug applications either using Arm Virtual Hardware simulation models or directly on hardware (for example, using the Arm® Keil ULINK™ family of debug and trace adapters). You can also use third-party debug probes to analyze applications. The ULINK debug and trace adapters work with preconfigured Flash programming algorithms for downloading the application program into Flash.

μVision provides statistical data and execution analysis reports to help you to test and validate your applications thoroughly. This is particularly important if you are working on safety-critical systems.

μVision also includes:

- **System Viewer.** View information about peripheral registers and change property values manually at run time.
- **Logic Analyzer.** View changes of values on a time graph, study signal and variable changes and view their dependency or correlation.
- **Template editor.** Create common text sequences, header descriptions, and generic code blocks.
- **Source Browser.** Navigate coded procedures quickly.
- **Configuration Wizard.** Use a graphical interface to maintain device and start-up code settings.
- **Multi-Project Manager.** Combine μVision projects, which logically depend on each other, into one single Multi-Project. This increases the consistency and transparency of your embedded application design.

For more information, see the [μVision documentation](#).

3.3 Arm Compiler for Embedded

Arm® Compiler for Embedded is the most advanced embedded C and C++ compilation toolchain for the development and optimization of embedded bare-metal software, firmware, and real-time operating system (RTOS) applications, ranging from small sensors to 64-bit devices.

Because Arm Compiler for Embedded is developed alongside the Arm architecture, it provides the earliest, most complete, and most accurate support for the latest architectural features and extensions, so that you can evaluate which Arm solution best suits your requirements and verify your design.

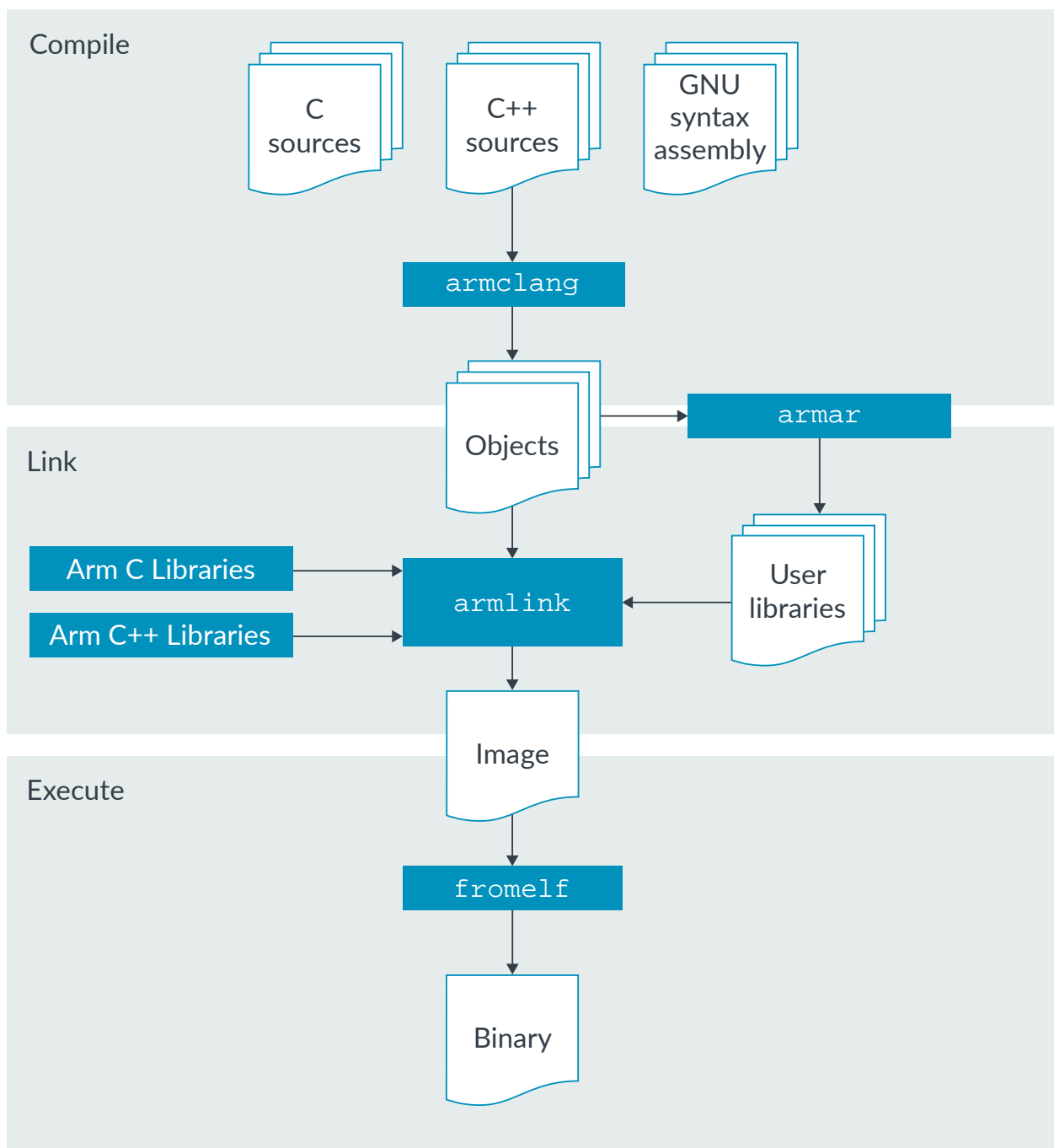
Arm Compiler for Embedded is used by leading companies in a wide variety of industries, including consumer electronics, networking, storage, telecommunications, security, and safety-critical systems.

Arm Compiler for Embedded consists of the following toolchain components:

- **armclang.** A compiler and integrated assembler based on modern LLVM and Clang technology. The armclang compiler supports GNU syntax assembly and the latest language standards, including C++17. It is highly compatible with source code originally written for GCC. It implements specifications including ANSI/ISO C and C++, ABI for the Arm architecture, ABI for the 64-bit Arm architecture, and Arm C Language Extensions (ACLE).
- **armlink.** A linker that combines objects and libraries to produce an executable.
- **Arm C libraries.** Runtime support libraries for embedded systems. These libraries include optimizations for performance and code density.
- **Arm C++ libraries.** Libraries based on the LLVM libc++ project.
- **fromelf.** An image conversion tool and disassembler.
- **armar.** An archiver that enables you to collect sets of ELF object files together.
- **Arm Compiler for Embedded FuSa.** (MDK-Professional edition only) A safety-qualified C/C++ toolchain that is suitable for developing embedded software for safety-critical markets including automotive, industrial, medical, railways, and aviation.
- **FuSa C libraries.** (MDK-Professional edition only)

The following diagram shows how the different toolchain components interact with each other in the build process for a typical embedded application:

Figure 3-2: Arm Compiler workflow diagram



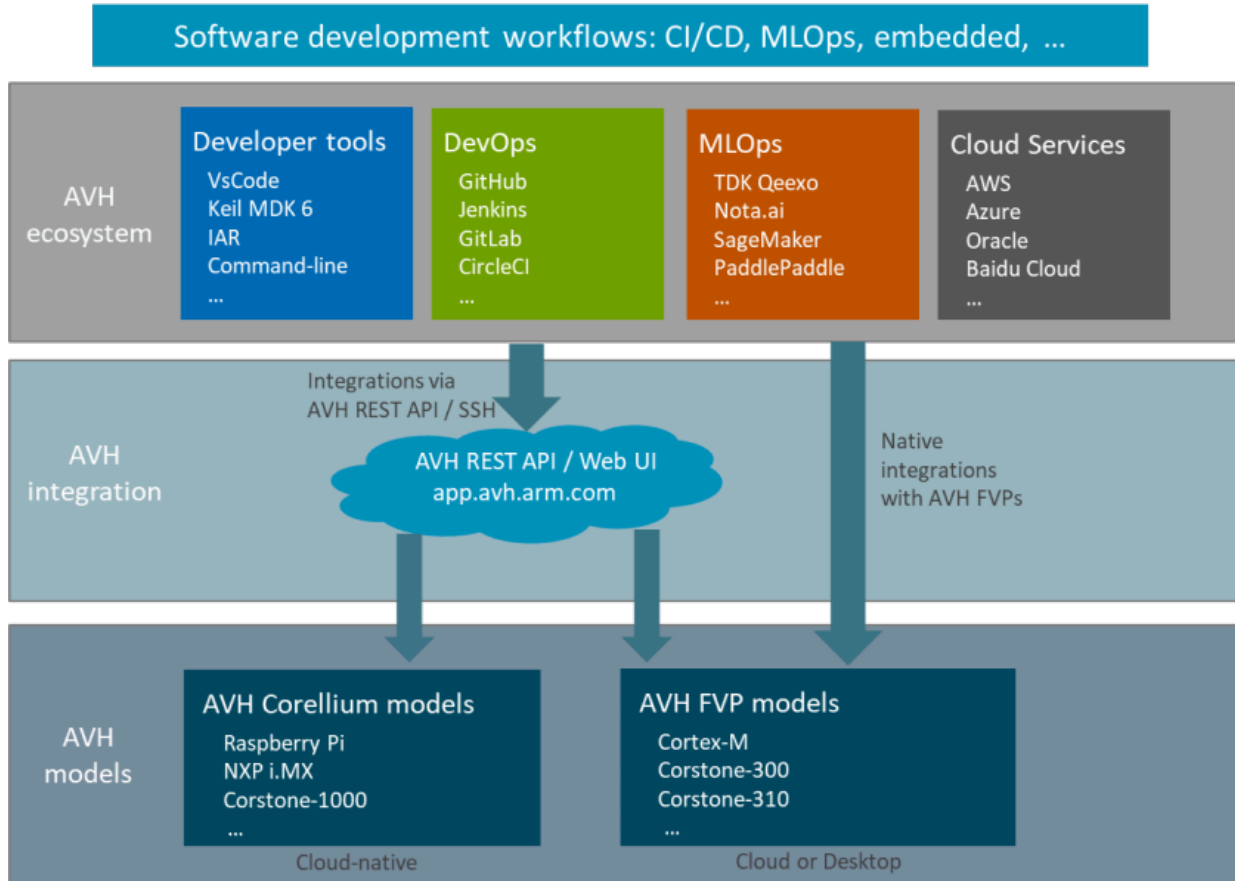
For more information, see the [Arm Compiler for Embedded documentation](#).

3.4 Arm Virtual Hardware

Arm® Virtual Hardware (AVH) enables software development on Arm-based processors using virtual targets. AVH can help simplify, automate, and accelerate the development process, and

reduce maintenance costs. This enables faster prototyping, build, and deployment cycles, and reduces time to market for embedded applications.

Figure 3-3: Arm Virtual Hardware overview diagram



You can start developing and testing your applications on AVH ahead of silicon availability, and without having to spend time and money setting up and maintaining physical board farms. AVH provides an accurate simulation of Arm-based SoCs, enabling seamless transfer from a virtual model to your target hardware.

AVH enables continuous integration and continuous delivery environments for embedded and IoT projects, and supports development processes like MLOps. Moving IoT engineers and data scientists to such development processes is key to scaling the Internet of Things to thousands or potentially millions of devices. With AVH, you can launch multiple virtual boards in seconds, and rapidly experiment with and test complex multidevice configurations.

With increased adoption of ML and edge compute in embedded applications, the ability to estimate how fast an ML model could run on a device is critical. You can use AVH to explore different network architectures and optimizations much more quickly and effectively than on physical hardware.

AVH in MDK

MDK enables you to download, install, and run AVH based on Fixed Virtual Platform models (FVPs). FVPs are precise simulation models of Arm Cortex-M based cores and reference platforms, such as Corstone™-300 or Corstone™-310.

FVP models are standalone programs that run in your target environment. They are available for cloud-native and desktop environments, and can be run from the command line or in your development tools.

For more information, including currently available board models and usage examples, see the AVH [User Guide](#) and [Solutions Overview](#).

4. CMSIS components

The [Common Microcontroller Software Interface Standard \(CMSIS\)](#) is a set of libraries, APIs, software components, and tools that enable you to write code for Arm® Cortex®-M based processors.

CMSIS is supported by many microcontroller manufacturers and provides a standardized way to write code for microcontrollers without having to know the internal details of different microcontrollers. Using CMSIS makes the process of writing and reusing code easier. It speeds up the development process, as you can port code written for one microcontroller to another microcontroller without having to modify it.

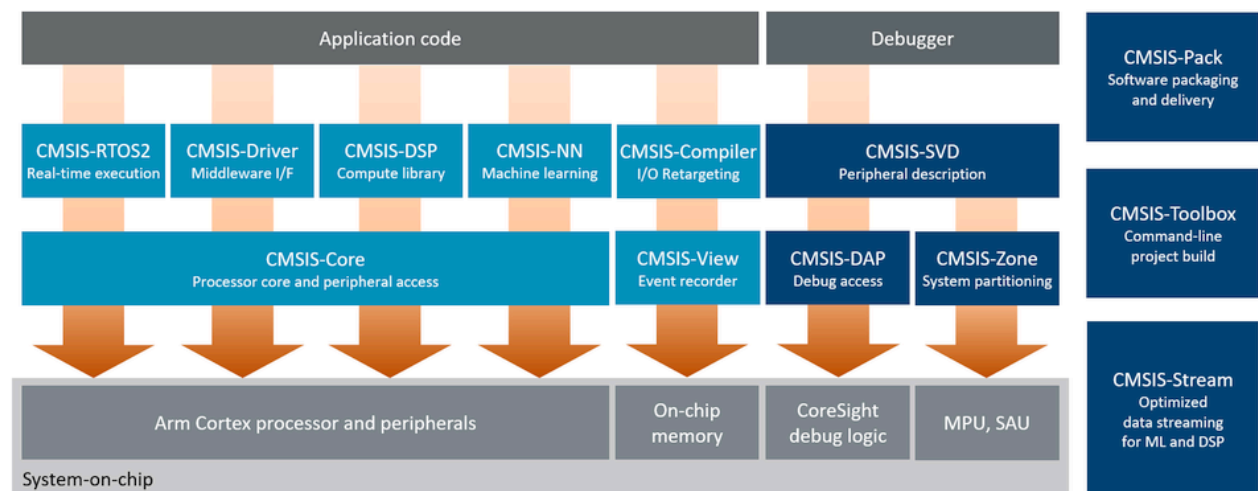
You can use the prewritten functions and libraries in CMSIS to control the hardware resources of different microcontrollers without having to learn how to manipulate those resources directly for each microcontroller. This reduces the time taken to build and debug projects, and speeds up the process of bringing new applications to market.

CMSIS also contains components that make it easier to extend the functionality of applications by adding features like digital signal processing, machine learning and neural networks, or managing multiple tasks and resources.

CMSIS is available under an Apache 2.0 license and is publicly developed on [GitHub](#).

CMSIS overview

Figure 4-1: CMSIS structure



Important developer-facing CMSIS components are:

- **CMSIS-Core**: Standardizes access to the processor core and device peripherals to make it easier to write code that runs across different Cortex-M controllers.

- **CMSIS-RTOS2**: A generic real-time operating system interface for devices based on the Arm Cortex processor. CMSIS-RTOS2 simplifies the process of managing and coordinating multiple tasks and resources. It can also help the process of migrating between different RTOS kernels.
- **CMSIS-Driver**: Provides a standardized API for configuring and controlling peripherals and devices. CMSIS-Driver is designed to be platform-independent, making it easy to reuse code across a wide range of supported microcontroller devices.
- **CMSIS-Compiler**: Provides software components for retargeting I/O operations in standard C run-time libraries, as well as a standardized API for core functions such as exceptions and interrupt handling.
- **CMSIS-View**: Provides visibility into the internal operations of microcontrollers, peripherals, hardware components, and software components during the development and debugging of embedded applications.
- **CMSIS-DSP**: A wide range of digital signal processing functions and routines. CMSIS-DSP algorithms are optimized for efficiency, helping you to maximize the performance of your applications and minimize resource usage. You can also use CMSIS-DSP as a basis for custom digital signal processing routines.
- **CMSIS-NN**: A collection of efficient neural network kernels developed to maximize the performance and minimize the memory footprint of neural networks on Arm Cortex-M processors. You can use the set of neural network operations that CMSIS-NN provides, or you can deploy your own specialized models.

CMSIS-NN enables you to perform inference at the edge rather than in the cloud. Edge computing can improve privacy and security, and reduce latency and bandwidth.

- **CMSIS-Stream**: A Python package and a set of C++ headers to use on embedded devices to process streams of samples. CMSIS-Stream provides low memory usage, minimal overhead, deterministic scheduling, and a modular design. It also provides a graphical representation.
- **CMSIS-Toolbox**: Command-line tools to work with software packs in Open-CMSIS-Pack format. This format is the basis for the csolution project format that is used in Keil Studio Cloud and the Keil Studio extensions in Visual Studio Code.
- **CMSIS-Zone**: A tool that helps to simplify partitioning, memory management, and access permissions in embedded applications.
- **CMSIS-DAP**: Provides access to the Debug Access Port (DAP) and enables communication over USB between a microprocessor and a debug tool on a host computer.

4.1 CMSIS basic concepts

This section summarizes some useful concepts to be aware of before you start working with CMSIS, and provides links to more detailed information.

4.1.1 CMSIS-Pack

CMSIS-Pack is a standardized packaging format for distributing software components for Arm® Cortex®-based microcontrollers. It simplifies the integration of components into projects, supports

versioning, and ensures compatibility across various devices, toolchains, and development environments.

CMSIS-Packs can include device-specific information, middleware components that provide common functionality, or application-level components such as code libraries for specific use cases.

CMSIS-Packs are a specific type of [software pack](#).

4.1.2 Software pack

A set of ready-to-use components and tools tailored for a specific hardware platform or for a specific purpose. This set is bundled together with a **Pack Description (PDSC) file** that describes the content that is included in the pack, and provides information on version history and any dependencies.

MDK provides tools that facilitate [product lifecycle management](#) with software packs. Additionally, you can use [CMSIS-Toolbox](#) to work with software packs in the [Open-CMSIS-Pack](#) format. This format is the basis for the [csolution](#) project format that is used in Keil Studio Cloud and the Keil Studio extensions in Visual Studio Code.

Software packs are designed to provide general-purpose resources for embedded development. There are also more specialized types of software packs called **Device Family Packs (DSP)** and **Board Support Packs (BSP)**, which are fine-tuned to provide support for specific microcontroller families or hardware boards.

For information on how working with basic software packs, DSPs, and BSPs, see the [Pack Tutorials](#) section of the Open-CMSIS-Pack documentation.

4.1.3 Software component

In embedded system development, a **software component** is a modular and reusable piece of software that fulfills a specific function within the wider system. Multiple components can be bundled together into a [software pack](#) or a [CMSIS-Pack](#). For more information, see the [Overview of additional software components](#) section of this guide.

4.1.4 CMSIS solutions

CMSIS solutions (also known as csolutions) are groups of related projects that are part of a larger application and that can be built separately. You can define a solution by editing a `*.csolution.yaml` file.

CMSIS-Toolbox takes the `*.csolution.yaml` and the `*.cproject.yaml` files as user input during the application build process.

For more information and example projects, see the [CMSIS-Toolbox documentation](#) and the [Work with CMSIS solutions](#) section of the Keil Studio Cloud User Guide.

The CMSIS Solution extension in the Keil® Studio Visual Studio Code extension pack, Arm Keil Studio Pack, also provides support for working with solutions. For more information, see the [Arm Keil Studio Visual Studio Code Extensions User Guide](#).



If you are using μ Vision®, you can use the [csolution build tool](#) in CMSIS-Toolbox to convert a `csolution` project into the `CPRJ` file format.

4.1.5 CMSIS projects

A **CMSIS project** is an individual project that can be built independently. Projects are defined by editing a `*.cproject.yaml` file to specify the files and components to include.

CMSIS-Toolbox takes the `*.csolution.yaml` file and the `*.cproject.yaml` file as user input during the application build process.

For more information and example projects, see the [CMSIS-Toolbox documentation](#) and the [Work with CMSIS solutions](#) section of the Keil Studio Cloud User Guide.

The CMSIS Solution extension in the Keil® Studio Visual Studio Code extension pack, Keil Studio Pack, also provides support for working with CMSIS solutions. For more information, see the [Arm Keil Studio Visual Studio Code Extensions User Guide](#).



If you are using μ Vision®, you can use the [csolution build tool](#) in CMSIS-Toolbox to convert a `csolution` project into the `CPRJ` file format.

4.2 Overview of CMSIS software components

Important developer-facing CMSIS components are:

- **CMSIS-Core**: Standardizes access to the processor core and device peripherals to make it easier to write code that runs across different Cortex®-M controllers.
- **CMSIS-RTOS2**: A generic real-time operating system interface for devices based on the Arm® Cortex processor. CMSIS-RTOS2 simplifies the process of managing and coordinating multiple tasks and resources. It can also help the process of migrating between different RTOS kernels.
- **CMSIS-Driver**: Provides a standardized API for configuring and controlling peripherals and devices. CMSIS-Driver is designed to be platform-independent, making it easy to reuse code across a wide range of supported microcontroller devices.
- **CMSIS-Compiler**: Provides software components for retargeting I/O operations in standard C run-time libraries, as well as a standardized API for core functions such as exceptions and interrupt handling.

- **CMSIS-View**: Provides visibility into the internal operations of microcontrollers, peripherals, hardware components, and software components during the development and debugging of embedded applications.
- **CMSIS-DSP**: A wide range of digital signal processing functions and routines. CMSIS-DSP algorithms are optimized for efficiency, helping you to maximize the performance of your applications and minimize resource usage. You can also use CMSIS-DSP as a basis for custom digital signal processing routines.
- **CMSIS-NN**: A collection of efficient neural network kernels developed to maximize the performance and minimize the memory footprint of neural networks on Arm Cortex-M processors.
- **CMSIS-Stream**: A Python package and a set of C++ headers to use on embedded devices to process streams of samples. CMSIS-Stream provides low memory usage, minimal overhead, deterministic scheduling, and a modular design. It also provides a graphical representation.
- **CMSIS-Toolbox**: Command-line tools to work with software packs in Open-CMSIS-Pack format. This format is the basis for the csolution project format that is used in Keil Studio Cloud and the Keil Studio extensions in Visual Studio Code.
- **CMSIS-Zone**: CMSIS-Zone helps to reduce the complexity of specifying partitioning, memory management, and access permissions in embedded applications using Arm Cortex-M processors.
- **CMSIS-DAP**: CMSIS-DAP is a protocol specification and an open-source firmware implementation that provides standardized access to the CoreSight™ Debug Access Port (DAP) available on many Arm Cortex processors as part of the CoreSight debug and trace functionality.

4.3 Overview of CMSIS base software components

The CMSIS base software components provide software abstractions for the basic functionality of microcontroller devices. These components are delivered in the Arm::CMSIS software pack.

4.3.1 CMSIS-Core

CMSIS-Core (Cortex®-M) implements the basic run-time system for a Cortex-M device and gives you access to the processor core and the device peripherals. It defines the following features:

- A hardware abstraction layer (HAL) for Cortex-M processor registers
- Standard system exception names to use when interfacing with system exceptions, making it easier to ensure compatibility across different platforms
- Methods to use to organize header files, and naming conventions for device-specific interrupts. Standardizing in this way makes it easy to learn new Cortex-M microcontroller products and improves software portability.
- Methods for system initialization to be used by each microcontroller vendor. For example, the standardized `SystemInit()` function is essential for configuring the clock system of the device.

- Intrinsic functions to use to generate specific CPU instructions that are not available through standard C functions.
- A standardized variable to determine the system clock frequency, which simplifies the setup of the `sysTick` timer.

Learn how to [use CMSIS-Core in embedded applications](#).

4.3.2 CMSIS-RTOS2

CMSIS-RTOS2 provides generic RTOS interfaces for devices based on the Arm® Cortex® microprocessor, and provides a standardized API for software components that require RTOS functionality. Using a standardized API moves the decision about which RTOS to use to a later stage in the design process and offers more flexibility. For more information, see the [CMSIS-RTOS2 documentation](#).

CMSIS-RTOS2 provides a set of basic features that are required in many applications, which reduces learning efforts and simplifies the sharing of software components. Middleware components that use CMSIS-RTOS2 are RTOS-agnostic and are easier to adapt.

CMSIS also provides project templates for CMSIS-RTOS2 which can be included in open-source CMSIS-RTOS2 implementations to provide a starting point for further development.

Benefits of an RTOS design

There are two basic design concepts for embedded applications, an **infinite loop** design (suitable for simple applications) and an **RTOS** design. An RTOS-based design has various benefits:

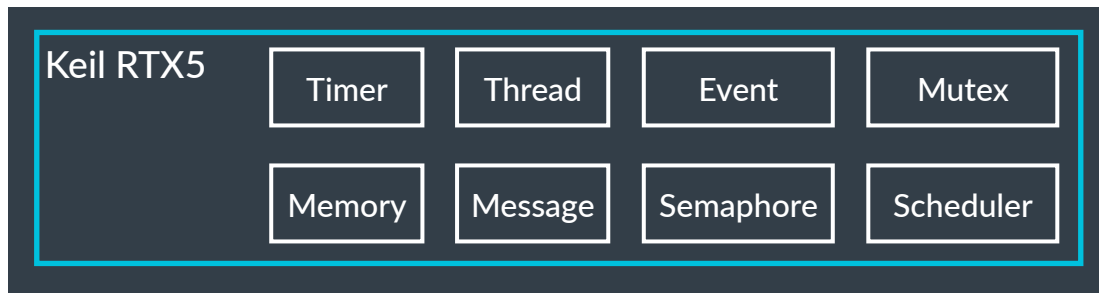
- The RTOS handles thread priority and run-time scheduling reliably.
- The RTOS provides a well-defined interface for communication between threads.
- A pre-emptive RTOS reduces the complexity of interrupt functions, because high-priority threads can perform time-critical data processing.
- Pre-emptive multitasking simplifies the ongoing enhancement of an application even across a larger development team, because you can add new functionality without risking the response time of more critical threads.
- Applications based on an infinite loop often poll for occurred interrupts. By contrast, RTOS kernels themselves are interrupt-driven and can largely eliminate polling. This enables the CPU to sleep or process threads more often.
- In the real world, your application must fulfill multiple different tasks. An RTOS-based application recreates this model in your software.

Modern RTOS kernels are designed to work closely with the interrupt system. This is essential for embedded systems and systems with real-time requirements, which must respond to interrupts (signals from hardware components such as buttons, sensors, timers, or peripherals) efficiently and promptly.

Keil RTX5

[Keil RTX version 5 \(RTX5\)](#) is a real-time operating system (RTOS) for Arm Cortex-M and Cortex-A processor-based devices that implements the [CMSIS-RTOS2 API](#) as its native interface.

Figure 4-2: RTX5 overview diagram



For more information, review the [Theory of Operation](#) and [get started with this tutorial](#).

4.3.3 CMSIS-Driver

The CMSIS-Driver API describes peripheral driver interfaces for middleware stacks and user applications. The API is designed to be generic and independent of a specific RTOS, making it reusable across a wide range of supported microcontroller devices. It covers a wide range of use cases for the supported peripheral types, but cannot take every potential use case into account. For more information, see the [CMSIS-Driver documentation](#).

The CMSIS software pack publishes the API under the **CMSIS-Driver** component class, with header files and documentation. These header files are the reference for the implementation of the standardized peripheral driver interfaces.

These implementations are typically published in the Device Family Pack (DFP) of a family or series of related microcontrollers under the **CMSIS-Driver** component class. A DFP might contain further device-specific interfaces in the **Device** component class, such as memory bus, General Purpose Input/Output (GPIO), or Direct Memory Access (DMA), in addition to the set of standard peripheral drivers covered by the specification.

The standard peripheral driver interfaces connect microcontroller peripherals to middleware that implements communication stacks, file systems, or graphical user interfaces. Each interface provides multiple instances representing physical interfaces of the same type in a device. For example, two physical Serial Peripheral Interfaces (SPIs) would have separate access structs to use to connect the driver to middleware or to the user application.

For more information, review the [Theory of Operation](#).

4.4 Overview of CMSIS extended software components

The CMSIS extended software components implement specific functionality optimized to run on Arm® processors. Each component is delivered in a separate CMSIS-Pack.

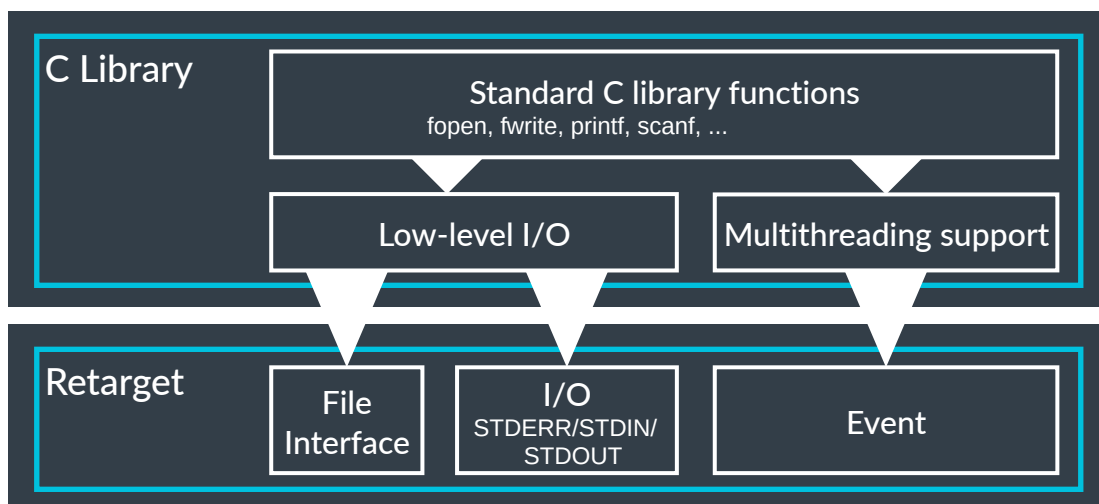
4.4.1 CMSIS-Compiler

CMSIS-Compiler provides software components that help you to adapt the input/output (I/O) operations in standard C runtime libraries to work with the specific I/O interfaces of your microcontroller or development board (a process known as retargeting).

CMSIS-Compiler supports the following interfaces for retargeting:

- A file interface for reading and writing files
- An I/O interface for standard I/O stream retargeting (stderr, stdin, stdout)
- An OS interface for multithread safety using an arbitrary RTOS

Figure 4-3: CMSIS-Compiler overview diagram



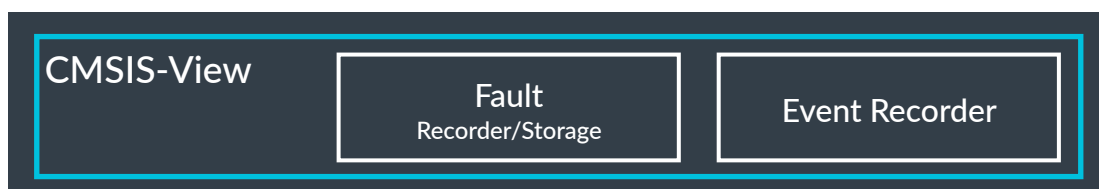
Standard C library functions are platform-independent, but multithreading support and the implementations of the low-level I/O are tailored to the target compiler toolchains.

See the [CMSIS-Compiler documentation](#) and get started using a [template](#).

4.4.2 CMSIS-View

CMSIS-View provides methodologies, software components, and utilities to help you to analyze the operation of embedded software programs on devices with Arm® Cortex®-M processors.

Figure 4-4: CMSIS-View overview diagram



CMSIS-View helps you to see how your embedded systems are operating, with minimal memory and timing overhead. The event statistics functions enable you to collect and analyze statistical data about the execution of your code.

CMSIS-View works on all Cortex-M devices, with only simple debug adapters necessary. The compiler-agnostic implementation allows simple integration with your application projects. CMSIS-View also enables RTOS-aware debugging for CMSIS-RTX and CMSIS-FreeRTOS, as well as logging capabilities for use in regression tests on Arm Virtual Hardware Fixed Virtual Platform (FVP) models (through [semihosting](#)).

See the [CMSIS-View documentation](#) and review the available [example projects](#).

4.4.3 CMSIS-DSP

CMSIS-DSP is an open-source software library that implements common digital signal processing (DSP) functions optimized for use on Arm® Cortex®-M and Cortex®-A processors.

CMSIS-DSP covers a range of compute categories, and provides kernels with several datatypes. A Python wrapper is also available, helping you to design your algorithm in Python using an API as close as possible to the C API.

See the [CMSIS-DSP documentation](#) for more information, or get started with this [learning path](#).

4.4.4 CMSIS-NN

The CMSIS-NN open-source software library maximizes performance and minimizes memory usage for neural networks running on Arm® Cortex®-M processors, through its collection of efficient neural network kernels. You can use the set of neural network operations that CMSIS-NN provides, or you can deploy your own specialized models.

CMSIS-NN enables you to perform inference at the edge rather than in the cloud. Edge computing can improve privacy and security, and reduce latency and bandwidth.

CMSIS-NN functions have several variants. CMSIS-NN automatically selects the best solution during compilation depending on the features of the target processor architecture.

For full details of available functions, see the [CMSIS-NN documentation](#), or [get started with this example](#).

4.5 Overview of CMSIS tools

MDK also includes useful tools for working with CMSIS-based components.

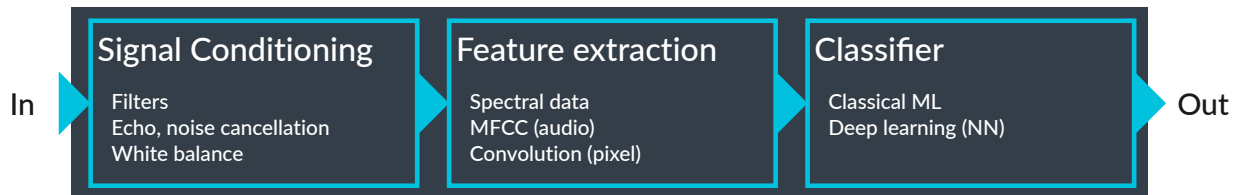
4.5.1 CMSIS-Stream

CMSIS-Stream is a Python package that optimizes data block streaming between the processing steps in DSP/ML applications. It enables modular design, which makes it easier to develop and

maintain DSP pipelines. The tools optimize scheduling of the processing nodes at build time, reducing memory usage. This process creates a clear representation of the design in the form of a compute graph.

The compute graph is a directed graph that shows the structure and sequence of data flows between processing nodes or components within the application. It uses a Python script file to describe the data formats, First In First Out (FIFO) buffers, data streams, and processing steps. The CMSIS-Stream tools convert the compute graph into optimized processing steps at build time.

Figure 4-5: CMSIS-Stream overview diagram



CMSIS-Stream provides tools to create optimized DSP pipelines, which are required to optimize ML software stacks. The visual representation that a compute graph provides can be helpful in complex DSP or ML workflows with multiple interconnected components, such as the one shown in the following diagram.

By optimizing signal conditioning and feature extraction, the complexity of the ML classifier. More DSP preprocessing helps by lowering the overall performance that is required for an ML application.

CMSIS-Stream also provides interfaces, header files, templates, and methods for data management that also work on asymmetric multiprocessing (AMP) systems, and usage examples to help you to get started.

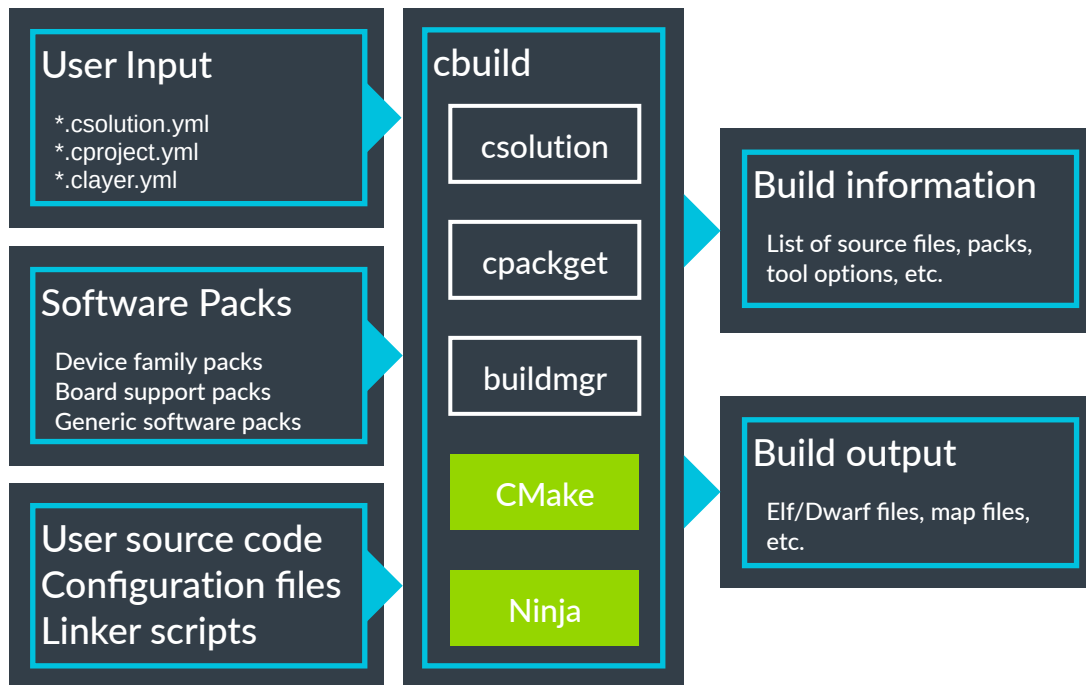
See the [CMSIS-Stream documentation](#) and review the [examples](#).

4.5.2 CMSIS-Toolbox

CMSIS-Toolbox provides command-line tools for creating and building embedded applications based on CMSIS-Packs. CMSIS-Toolbox supports multiple compilation tools, such as Arm Compiler

for Embedded, GCC, IAR, and LLVM. The tools also help you to create, maintain, and distribute CMSIS-Packs that include software components or software and hardware support.

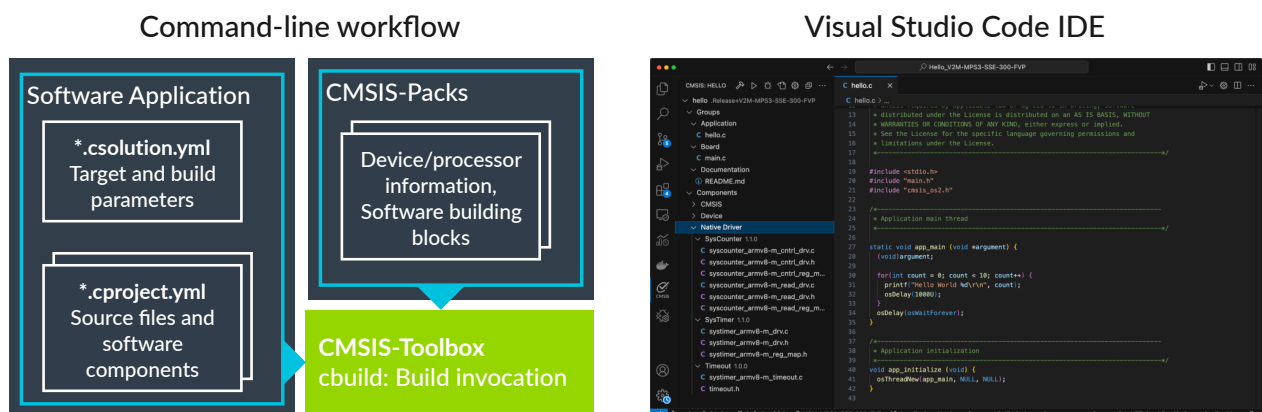
Figure 4-6: CMSIS-Toolbox overview diagram



You can use the command-line tools either standalone or integrated into the extensions for Visual Studio Code or DevOps systems for Continuous Integration (CI). Tools are available for all host platforms (Windows, Mac, and Linux) and are deployable in a flexible way.

For more information on how to use the `cbuild`, `csolution`, and `cpackget` tools from the command line, including syntax details and usage examples, see the [Build tools documentation](#).

Figure 4-7: CLI and IDE workflow



Software packs make it easier to set up tools by enabling you to select devices or boards and to create projects that provide access to reusable software components.

The ability to organize solutions into independently-managed projects simplifies many use cases, including multi-processor applications or unit testing.

CMSIS-Toolbox also makes provisions for product lifecycle management (PLM), with configuration file management and versioned software packs that are easy to update.

Software layers enable code reuse across similar applications, with a preconfigured set of source files and software components.

CMSIS-Toolbox offers support for:

- Multiple hardware targets, enabling you to deploy your application to different hardware (test board, production hardware, virtual hardware, and so on).
- Multiple build types, to support software testing and verification (debug build, test build, release build, and so on).
- Multiple toolchains (even within the same set of user input files) and command-line options that enable you to select different toolchains during verification.

CMSIS-Toolbox uses a CMake back end for the build process. Using CMake with CMSIS-Toolbox simplifies the generation of `compile_commands.json` files for solutions. These JSON files contain a list of project files and the compiler commands used in the build process, and can be used by various Visual Studio Code extensions to power IntelliSense.

See the [CMSIS-Toolbox documentation](#) for more information.

4.5.3 CMSIS-Zone

CMSIS-Zone helps to reduce the complexity of specifying partitioning, memory management, and access permissions in embedded applications using Arm® Cortex®-M processors.

You can use CMSIS-Zone to specify access and security permissions to memory regions, in both secure and non-secure modes. You can then use the XML-based zone files to generate the header files for memory management and partition generation in your application.

For more information, see the [CMSIS-Zone documentation](#).

4.5.4 CMSIS-DAP

CMSIS-DAP provides embedded software developers with standardized access to the CoreSight™ Debug Access Port (DAP) available on many Arm® Cortex® processors as part of the CoreSight on-chip debug and trace functionality.

CMSIS-DAP enables standardized communication between the microprocessor where an embedded application is run, and a debug tool running on a host computer. CMSIS-DAP provides the interface firmware for a debug unit that connects the debug port of the device to the USB port.

With it, a software debug tool that runs on a host computer can connect using USB and the debug unit.

For more information, see the [CMSIS-DAP documentation](#).

5. Other software components and packs

Designing and implementing software for embedded systems requires a modular architecture using multiple components. Software packs are collections of components bundled together for a specific purpose (for example, middleware, source code, libraries, or example projects).

Packs are used to provide ready-to-use components for specific microcontroller families or development platforms. They can simplify the process of setting up a development environment and writing code for a particular embedded system.

CMSIS-Packs are a specific type of software pack. They adhere to the CMSIS standard, which defines a consistent interface for accessing and configuring core features of Arm® Cortex®-M processors, and enable you to easily integrate and maintain software components in your projects.

A CMSIS-Pack includes metadata about the files that belong to a software component, preserving the information from the original vendor of the component. Metadata can include dependency information for toolchains, devices, and processors, which simplifies integration into applications.

Another benefit of the CMSIS-Pack system is that it enables a consistent software component upgrade process, and identifies incompatible configuration files that might be part of the user application. In addition, software component providers can specify the interfaces and their relationship to other software components.

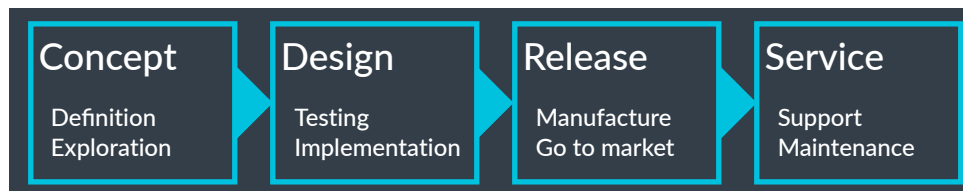
Arm maintains a [list of CMSIS-Packs](#) that are publicly available.

For more information, see the [CMSIS-Pack documentation](#).

5.1 Product lifecycle management with software packs

MDK enables you to install multiple versions of a software pack. This enables product lifecycle management (PLM), which is common for many projects.

Figure 5-1: Diagram showing the stages of PLM



There are four phases of PLM:

- **Concept**: Definition of major project requirements and exploration with a functional prototype
- **Design**: Prototype testing and implementation of the product based on the final technical features and requirements
- **Release**: The product is manufactured and brought to market

- **Service:** Maintenance of the products, including support for customers. Finally, phase-out or end-of-life.

In the concept and design phases, you normally use the latest software packs so that you can incorporate new features and bug fixes quickly. Before product release, you freeze the software components to a known tested state. In the service phase, you use the fixed versions of the software components to support customers in the field.

The strict [semantic versioning](#) of CMSIS-Packs makes it easier to manage the installed versions of software packs that you use in your projects.

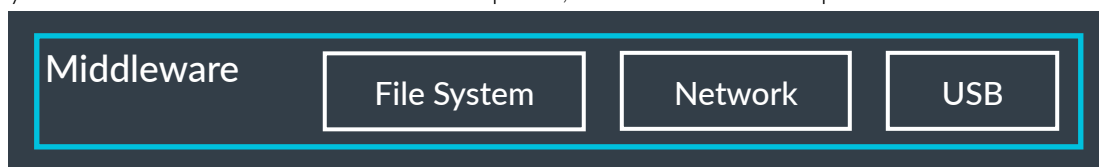
5.2 Overview of additional software components

The following table lists software components that are frequently used in embedded applications, including the components in the MDK-Middleware software pack.

Table 5-1: Frequently used software components

Software component	Description
CMSIS-FreeRTOS	A CMSIS-RTOS2 adaptation of the FreeRTOS kernel
CMSIS-mbedTLS	An MbedTLS fork delivered in a CMSIS-Pack
Synchronous Data Stream (SDS)	A data stream management framework
Network	An MDK-Middleware component for TCP/IP networking using Ethernet or serial protocols
File System	An MDK-Middleware component for file access on various storage types
USB	An MDK-Middleware component for USB host and device communication, supporting standard USB device classes
IoT Clients	Open-source clients for various cloud service providers.
Open-source components	Open-source components that you can use to extend the functionality of your applications

The following figure shows the components in the MDK-Middleware software pack; if you have installed additional software packs, more software components are available.



5.2.1 CMSIS-FreeRTOS

FreeRTOS is a market-leading real-time operating system (RTOS) for embedded microcontrollers. It is professionally developed, strictly quality controlled, robust, fully supported and documented, free to use in commercial products without a requirement to expose proprietary source code, and has no IP infringement risk.

Arm® has created an implementation of FreeRTOS that supports the [CMSIS-RTOS2](#) API for real-time operating systems (RTOS). Using this software pack, you can choose between a native

FreeRTOS implementation or one that adheres to the CMSIS-RTOS2 API and uses FreeRTOS internally. The CMSIS-RTOS2 API enables programmers to create portable application code to use with different RTOS kernels (for example, Keil RTX5).

See the [CMSIS-FreeRTOS documentation](#) and get started with an [example project](#).

5.2.2 CMSIS-mbedTLS

[Mbed TLS](#) is a C library that implements cryptographic primitives, X.509 certificate manipulation, and the SSL/TLS and DTLS protocols. It is particularly suitable for embedded systems because of its small code size.

See the [CMSIS-mbedTLS GitHub repository](#) for more information.

5.2.3 Synchronous Data Stream (SDS) framework

The Synchronous Data Stream (SDS) framework implements a data stream management system and provides methods and tools for developing and optimizing embedded applications that integrate digital signal processing (DSP) and machine learning (ML) algorithms. You can use the framework with the compute graph streaming in the [CMSIS-DSP](#) library.

SDS implements flexible data stream management for sensor and audio data interfaces. It supports data streams from multiple interfaces, including provisions for time drifts. You can also use it to record real-world data for analysis and development, or to play back real-world data for algorithm validation by using Arm Virtual Hardware. SDS data files have several use cases, such as:

- To provide input to DSP development tools such as filter designers
- To provide input to ML model classification, training, and performance optimization
- To verify that a DSP algorithm runs on Cortex®-M targets with offline tools

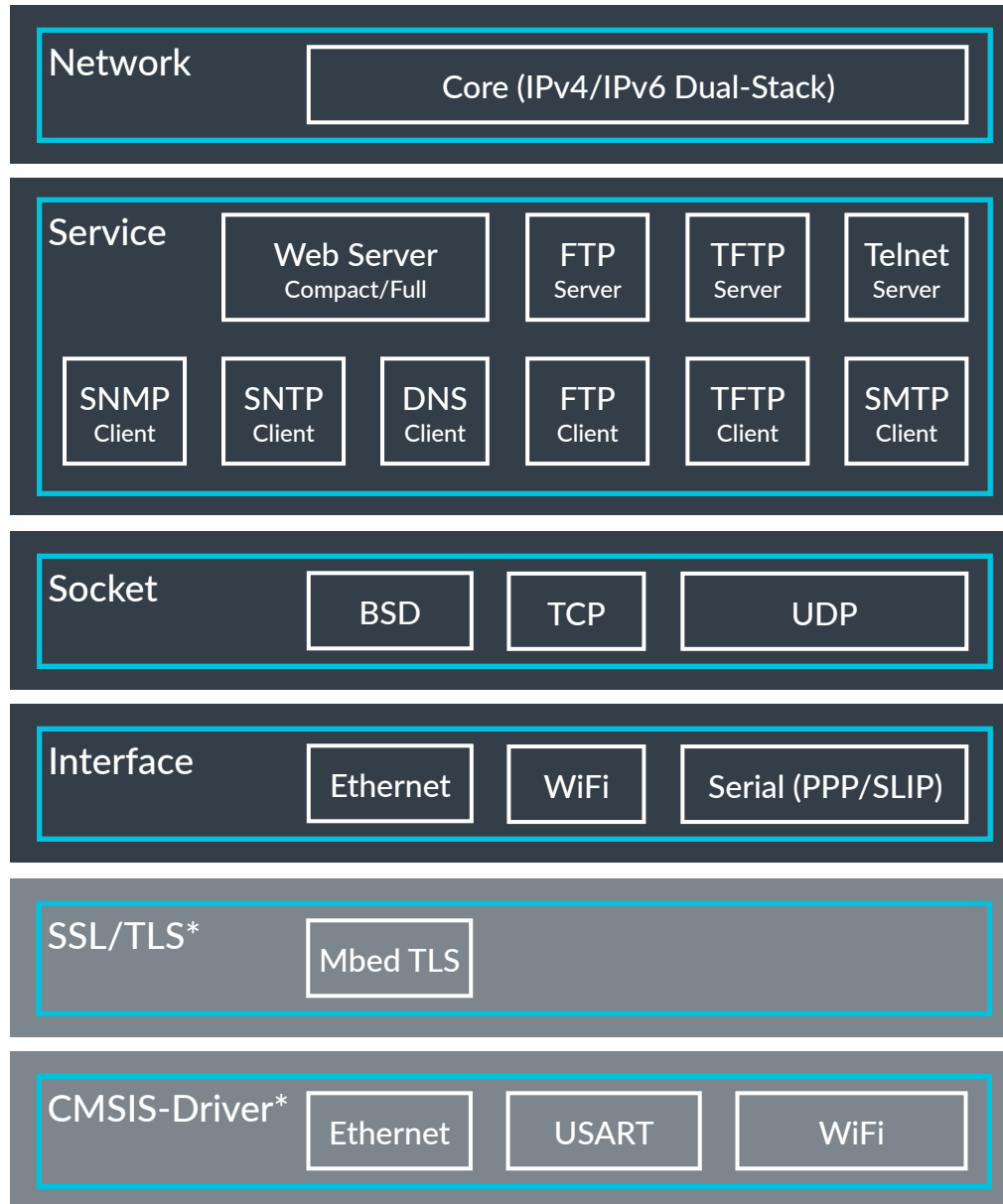
SDS defines a binary data format with a YAML-based metadata file. It also includes Python-based tools for recording, playback, visualization, and data conversion.

See the [SDS-Framework documentation](#) and get started by using an [example](#).

5.2.4 Network component

The Network component in the MDK-Middleware pack contains services, protocol sockets, and physical communication interfaces for creating IPv4 and IPv6 networking applications.

Figure 5-2: MDK-Middleware Network component overview diagram



*These components are not part of the Network component



The **Mbed TLS**, **Ethernet**, **USART**, and **Wi-Fi** components work with the Network component, but are part of separate packs.

The services provide program templates for common networking tasks.

All services rely on a network socket for communication. The Network component supports Tenable Security Center (TSC), User Datagram Protocol (UDP), and Berkeley Software Distribution (BSD) sockets.

The physical interface can be either Ethernet, WiFi, or a serial connection using Serial Line Internet Protocol (SLIP) or Point-to-Point Protocol (PPP).

A driver provides the interface to the microcontroller peripherals or external components:

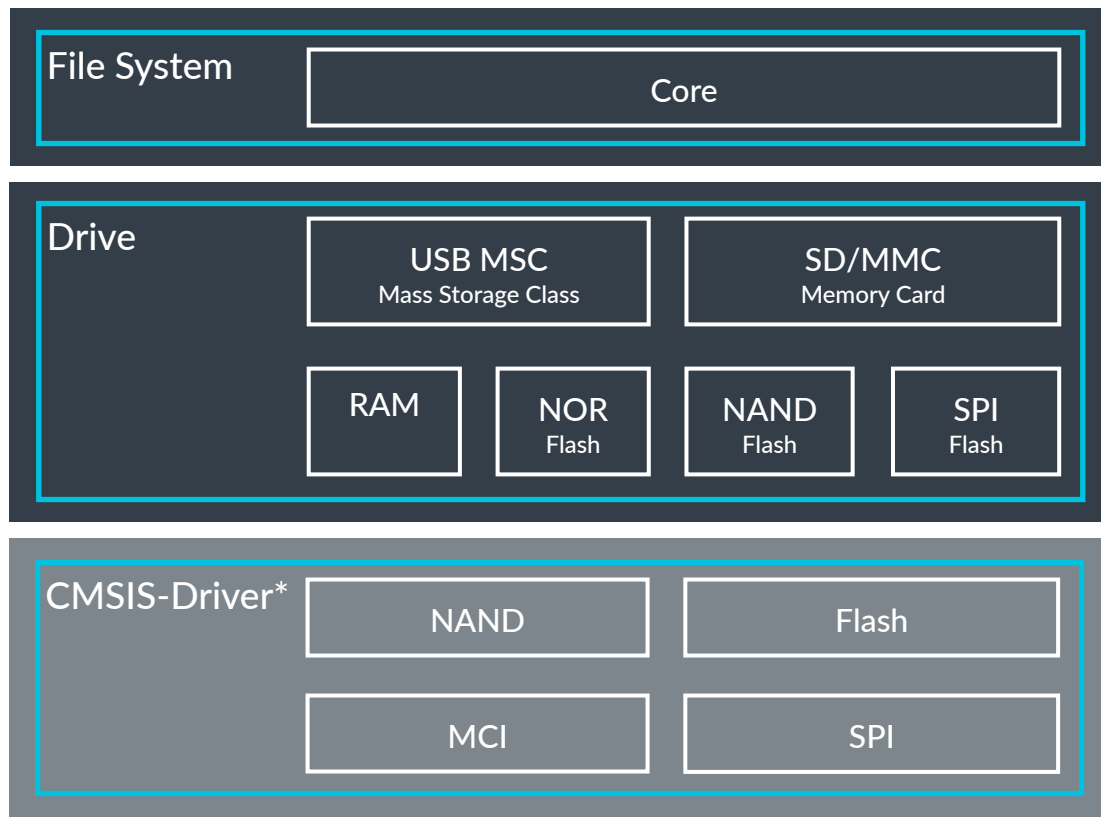
- Ethernet requires an Ethernet Media Access Control (MAC) address and an Ethernet physical layer (PHY) driver
- PPP or SLIP interfaces use a universal synchronous/asynchronous receiver/transmitter (USART) and a modem
- WiFi interfaces require a WiFi module driver

See the [Network component documentation](#) and get started by using an [example](#).

5.2.5 File System component

The File System component in the MDK-Middleware pack enables your embedded applications to create, save, read, and modify files in storage devices such as RAM, Flash, memory cards, or USB memory devices.

Figure 5-3: MDK-Middleware File System component overview diagram



*These components are not part of the File System component

The File System component is structured as follows:

- Storage devices are referenced as drives which you can access
- You can implement multiple instances of the same storage device (for example, you might want to have two SD cards attached to your system)
- The File System Core supports thread-safe operation and uses an Embedded File System (EFS) for NOR and Serial Peripheral Interface (SPI) Flashes, or a File Allocation Table (FAT) file system. The FAT file system is available in two variants:
 - The long file name variant supports up to 255 characters
 - The short file name variant supports only file names in 8.3 format
- The Core allows simultaneous access to multiple storage devices (for example, backing up data from internal flash to an external USB device)
- To access the drives, drivers are in place to support the following storage devices:

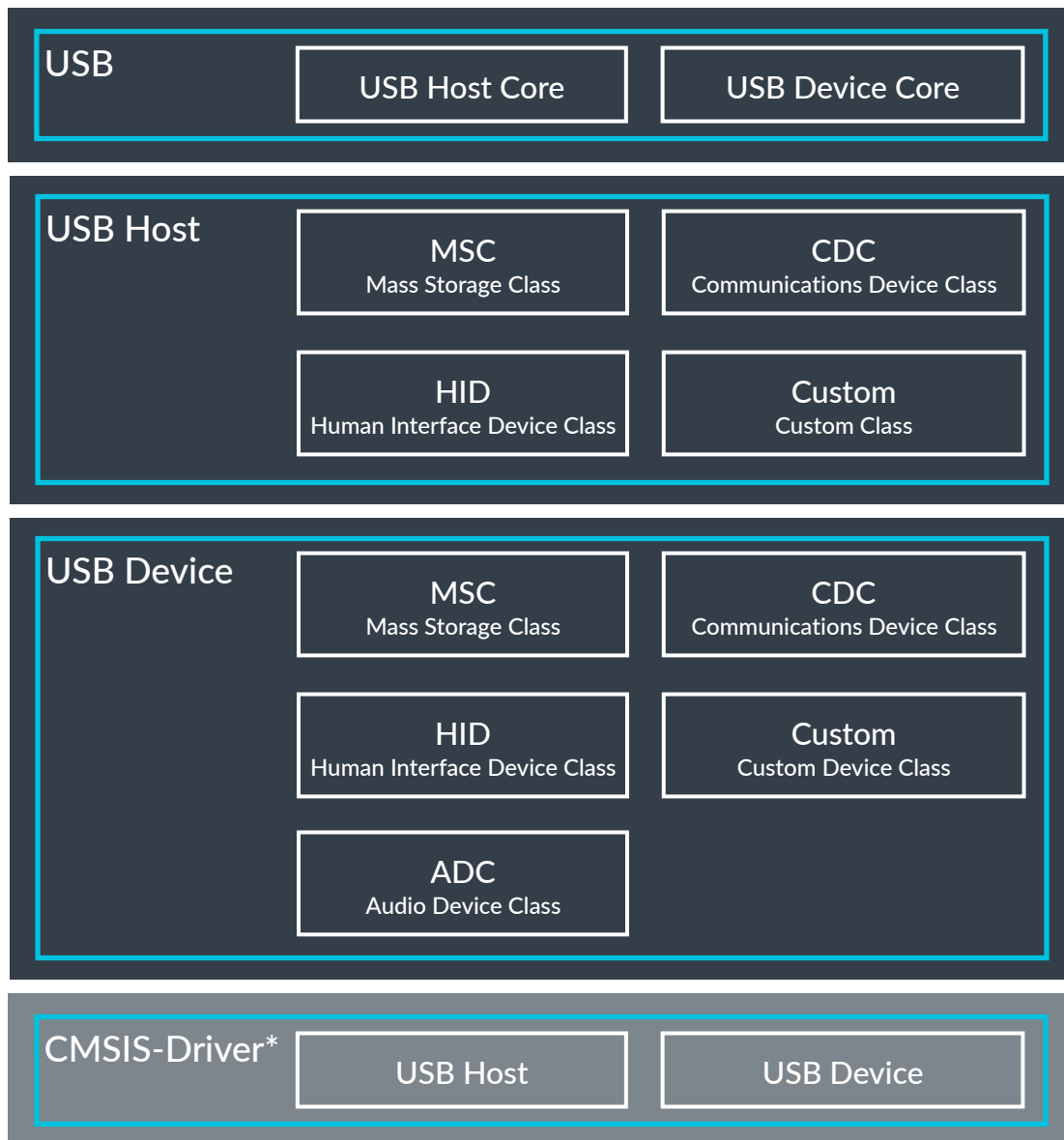
- Flash chips (NAND, NOR, and SPI)
- Memory card interfaces (SD/SDxC/MMC/eMMC)
- USB devices
- On-chip RAM, Flash, and external memory interfaces

Review the [Theory of Operation](#) and get started by using an [example](#).

5.2.6 USB component

The USB component in the MDK-Middleware pack enables you to create USB device and USB host applications. The USB component handles the USB protocol so that you can focus on your application needs.

Figure 5-4: MDK-Middleware USB component overview diagram



*These components are not part of the USB component

The USB component is structured as follows:

- USB Host ([MDK-Professional](#) only) is used to communicate to other USB device peripherals over the USB bus

- USB Device implements a device peripheral that you can connect to a USB Host
- The USB API for USB Host and USB Device provides the interface to the microcontroller peripherals

The following USB classes are supported:

- Human Interface Device (HID)
- Mass Storage Class (MSC)
- Communication Device Class (CDC)
- Audio Device Class (ADC) (USB Device only)
- Custom Class (for implementing new or unsupported USB Classes)
- Composite USB Devices that support multiple device classes.

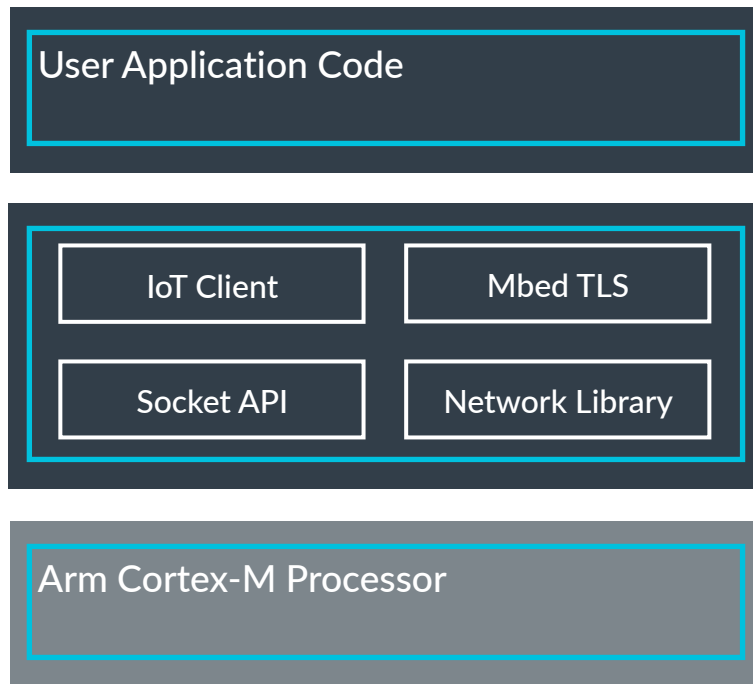
See the [USB component documentation](#) and get started by using a [USB Device example](#) or a [USB Host example](#).

5.2.7 IoT clients

The Internet of Things (IoT) describes connected end-node devices that collect, process, and exchange data. These devices are connected over the Internet to a cloud service that provides processing power, data analytics, and storage capabilities. An IoT client is a software interface which runs on the end-node device and establishes the connection to a cloud service.

Many cloud service providers offer open-source software that implements an [IoT client](#) for an embedded system. Arm® adapted these clients to use the reliable [MDK-Middleware Network component](#) for communication with the cloud service. Alternatively, you can use WiFi devices that are supported by a CMSIS-WiFi driver.

Figure 5-5: IoT application software stack



Most IoT clients use the Message Queuing Telemetry Transport (MQTT) protocol, which is a lightweight messaging protocol for IoT applications. It communicates over TCP/IP using a TCP socket (for a non-secure connection) or a TLS socket (for a secure connection with encryption).

MDK provides a CMSIS-Pack to give you the basic building blocks that are required to connect to Amazon Web Services. The [Amazon AWS IoT](#) pack provides an SDK for connecting to AWS IoT from a device using embedded C.

Software packs are generic (device-independent) and are listed in the [pack index](#).

5.2.8 Overview of open-source components

There are many open-source components that you can use in MDK v6 to extend the functionality of your embedded applications. This section outlines a small selection of open-source components that are available on the market.

LVGL

LVGL (Light and Versatile Graphics Library) is an embedded graphics library that you can use to create graphical user interfaces with a low memory footprint, suitable for use in embedded systems. You can use LVGL with any microcontroller, microprocessor, and display type.

[Download the pack](#), or for more information see the [LVGL repository](#) or the [documentation](#).

lwIP

lwIP is a lightweight implementation of the TCP/IP protocol suite. It supports most common TCP/IP protocols in full, but reduces resource usage, making it ideal for use in embedded applications.

[Download the pack](#), or for more information see the [lwIP repository](#) or the [documentation](#).

6. Create new applications

Learn more about how to create and build applications using CMSIS with MDK.

6.1 Create a new solution using the Keil Studio VS Code extensions

This section describes the basic workflow for creating, running, and debugging a simple “Hello world” example solution with the Keil Studio VS Code extensions, and provides links to more detailed instructions in the [Arm Keil Studio Visual Studio Code Extensions User Guide](#). The workflow involves the following steps:


- [Create a solution](#)
- [Add software components to your solution](#)
- [Add the source code files to your solution](#)
- [Configure virtual hardware](#)
- [Build the solution](#)
- [Run the solution](#)
- [Debug the solution](#)

6.1.1 Create a solution

Create a new solution with all the basic files that you need for the hardware that you select. For more detailed information, see [Create a solution](#) in the Arm Keil Studio Visual Studio Code Extensions User Guide.



This procedure describes creating a solution for the Arm V2M-MPS3-SSE-300-FVP virtual hardware. Adapt the steps for other starter kits or boards.

1. Install the Keil Studio Pack in Visual Studio Code Desktop.
2. Click **CMSIS**  in the Activity Bar to open the **CMSIS** view.
3. Click **Create a New Solution**. The **Create New Solution** view opens.
4. In the **Target Board** drop-down list, find the V2M-MPS3-SSE-300-FVP virtual hardware, and then click **Select**.
5. In the **Templates and Examples** drop-down list, select **Blank solution**.
6. Enter a name for the project to include in your solution (for example, `helloworld`).


7. Enter a solution name.
8. Specify the location where you want to store your solution files.
9. Click **Create**. A confirmation dialog box opens. Click **Open** to open the new solution and see the files in the **Explorer** view.
10. An **Arm Environment Activation** dialog box displays. Confirm that the Environment Manager extension can automatically activate the workspace and download the tools specified in the `vcpkg-configuration.json` file that was generated when you created the solution.



For this example, you must specify the virtual hardware models to use in the `vcpkg-configuration.json` file. See [Configure virtual hardware](#) for more details.

6.1.2 Add software components to your solution

Select the relevant software components that you want to use. For more detailed information, see [Manage software components](#).

1. In the **CMSIS** view, move your mouse over the Project header and click . The **Software Components** view opens.
2. In the **Software packs: Solution** drop-down list, select **All packs**.
3. Click the arrows next to a heading in the **Software Components** view to browse the list of components. Make sure that the following components are selected:
 - **CMSIS > Core, OS Tick**, and **RTOS2 > Keil RTX5** (with **Source** selected in the **Variant** drop-down list).
 - **Device > Definition** and **Startup**
 - **Device > Native driver > SysCounter, SysTimer**, and **Timeout**



If your solution requires some packs that are not installed, you are prompted to install them. Similarly, if the components that you add have dependencies that are not installed on your machine, you are prompted to add them.

6.1.3 Add the source code files to your solution

Add the `main.h` header file and the `helloWorld.c` files to your project, and add project-specific code to the files.

1. In the **CMSIS** view, in the project outline, go to **Groups > Source Files**.
2. Click **+** next to the **Source Files** heading, and then click **Add New File**.
3. In the dialog box that opens, name the file `main.h`, and then click **Save**.

4. Copy and paste the following code into the `main.h` file:

```
#ifndef MAIN_H
#define MAIN_H

/* Prototypes */
extern void app_initialize (void);

#endif
```

5. Click **+** next to the **Source Files** heading, and then click **Add New File**.
6. In the dialog box that opens, name the file `helloworld.c`, and then click **Save**.
7. Copy and paste the following code into the `helloworld.c` file:

```
#include <stdio.h>
#include "main.h"
#include "cmsis_os2.h"

/*-----
 * Application main thread
 *-----*/

static void app_main (void *argument) {
    (void)argument;

    for(int count = 0; count < 10; count++) {
        printf("Hello World %d\r\n", count);
        osDelay(1000U);
    }
    osDelay(osWaitForever);
}

/*-----
 * Application initialization
 *-----*/
void app_initialize (void) {
    osThreadNew(app_main, NULL, NULL);
}
```

8. Open the `main.c` file. Delete the existing code, and replace it with the following:

```
#include "RTE_Components.h"
#include CMSIS_device_header
#include "cmsis_os2.h"
#include "main.h"

int main() {
    osKernelInitialize();           // Initialize CMSIS-RTOS2
    app_initialize();               // Initialize application
    osKernelStart();               // Start thread execution



    for (;;) {
    }
}
```

6.1.4 Configure virtual hardware

To build and run projects on virtual hardware such as V2M-MPS3-SSE-300-FVP used in this example, you must add a configuration file in the project directory, and specify the models to use in the `vcpkg-configuration.json` file.



Arm Virtual Hardware simulation models (Fixed Virtual Platform models, or FVPs) are currently not available on macOS.

1. In the Activity bar, click **Explorer** .
2. In the project header, click **New File...** .
3. Enter `vht-config.txt` as the name of the new file.
4. Open the `vht-config.txt` file and copy and paste the following code into it:

```
# Parameters:
# instance.parameter=value          #(type, mode) default = 'def value' :
# description : [min..max]
#-----
cpu0.semihosting-enable=1           # (bool , init-time)
default = '0'                      : Enable semihosting SVC traps. Applications that do not use
semihosting must set this parameter to false.
mps3_board.visualisation.disable-visualisation=1      # (bool , init-time)
default = '0'                      : Enable/disable visualisation
#-----
```

5. Open the `vcpkg-configuration.json` file. Copy and paste the following line at the beginning of the "requires": block:

```
"arm:models/arm/avh-fvp": "^11.22.39",
```

6.1.5 Build the solution

To check that your example solution builds, see [Build the example project](#) in the Arm Keil Studio Visual Studio Code Extensions User Guide.

6.1.6 Run the solution

To run the solution on your virtual hardware, see [Run the solution on your board](#) and [Use the Run and Debug Configuration visual editor for your run configuration](#) in the Arm Keil Studio Visual Studio Code Extensions User Guide.

6.1.7 Debug the solution

To start a debug session, see [Start a debug session](#) and [Use the Run and Debug Configuration visual editor for your debug configuration](#) in the Arm Keil Studio Visual Studio Code Extensions User Guide.

7. Terminology

This section provides brief definitions of important concepts in Keil Studio and CMSIS. For more information and links to more detailed resources, see [CMSIS basic concepts](#).

CMSIS-Pack:

An open packaging standard for distributing embedded software libraries, documentation, device parameters, and evaluation board support. The CMSIS-Pack standard is now part of the [Open-CMSIS-Pack](#) project.

CMSIS-Toolbox:

Command-line tools for working with components defined in Open-CMSIS-Pack format. [CMSIS-Toolbox](#) includes tools for installing CMSIS-Packs, defining and scaling embedded software projects, and orchestrating builds.

CMSIS context:

A build configuration inside a CMSIS solution that combines a project, a build type (for example, `Debug` or `Release`), and a target (that is, hardware). A context is specified in the format `Project.BuildType+Target`. For more information, see the [CMSIS context documentation](#).

CMSIS software components:

Embedded software abstractions and libraries, packaged inside a [CMSIS-Pack](#). For more information, see the [CMSIS components](#) section of this guide.

CMSIS solution (also known as a csolution):

The YAML-based project format used by CMSIS-Toolbox. For more information, see [CMSIS Solution Project File Format](#).