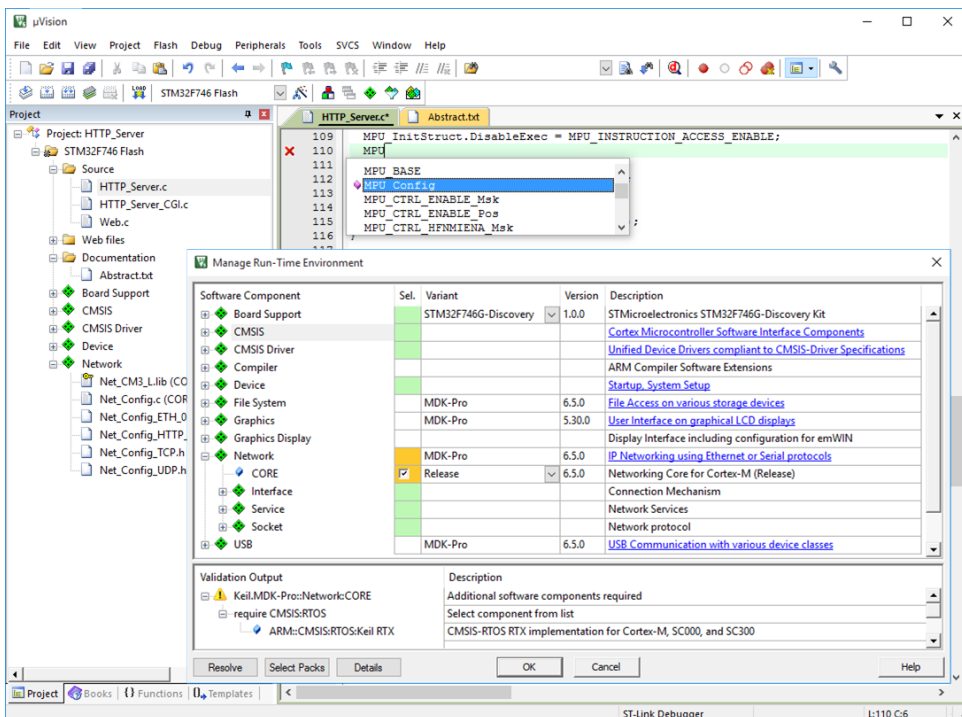# arm KEIL

Microcontroller Tools

Getting started with MDK

Create applications with μVision®
for Arm® Cortex®-M microcontrollers

Information in this document is subject to change without notice and does not represent a commitment on the part of the manufacturer. The software described in this document is furnished under license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the software on any medium except as specifically allowed in the license or nondisclosure agreement. The purchaser may make one copy of the software for backup purposes. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage and retrieval systems, for any purpose other than for the purchaser's personal use, without written permission.

Arm®, Keil®, µVision®, Cortex®, TrustZone®, CoreSight™ and ULINK™ are trademarks or registered trademarks of Arm Germany GmbH and Arm Ltd.

Microsoft® and Windows™ are trademarks or registered trademarks of Microsoft Corporation.

PC® is a registered trademark of International Business Machines Corporation.

NOTE
We assume you are familiar with Microsoft Windows, the hardware, and the instruction set of the Arm® Cortex®-M processor.

Every effort was made to ensure accuracy in this manual and to give appropriate credit to persons, companies, and trademarks referenced herein.

Preface

Thank you for using the Arm Keil® MDK Microcontroller Development Kit. To provide you with the best software tools for developing Arm Cortex-M processor based embedded applications we design our tools to make software engineering easy and productive. Arm also offers complementary products such as the ULINK™ debug and trace adapters and a range of evaluation boards. MDK is expandable with various third-party tools, starter kits, and debug adapters.

Chapter overview

The book starts with the installation of MDK and describes the software components along with complete workflow from starting a project up to debugging on hardware. It contains the following chapters:

MDK Introduction provides an overview about the MDK Tools, the software packs, and describes the product installation along with the use of example projects.

CMSIS is a software framework for embedded applications that run on Cortex-M based microcontrollers. It provides consistent software interfaces and hardware abstraction layers that simplify software reuse.

Software Components enable retargeting of I/O functions for various standard I/O channels and add board support for a wide range of evaluation boards.

Create Applications guides you towards creating and modifying projects using CMSIS and device-related software components. A hands-on tutorial shows the main configuration dialogs for setting tool options.

Debug Applications describes the process of debugging applications on real hardware and explains how to connect to development boards using a wide range of debug adapters.

MDK-Middleware gives an overview of the middleware components available for users of the MDK-Professional and MDK-Plus editions. It also explains how to create applications that use the MDK-Middleware and contains essential tips and tricks to get you started quickly.

Contents

NOTE
This user's guide describes how to create projects for Arm Cortex-M
microcontrollers using the µVision IDE/Debugger.

MDK Introduction

MDK helps you to create embedded applications for more than 7,500 Arm Cortex-M processor-based devices. MDK is a powerful, yet easy to learn and use development system. It consists of MDK-Core and software packs, which can be downloaded and installed based on the requirements of your application.

MDK Tools	MDK-Core		Arm C/C++ Compiler
	µVision IDE & debugger with pack management		With safety qualification

Software Packs	Device	CMSIS	MDK-Middleware	
	Startup	CMSIS-Core	Network IPv4/IPv6	Graphics
	Device HAL	CMSIS-DSP	USB Host/Device	Mbed TLS SSL/TLS security
	CMSIS Drivers	CMSIS-RTOS	File System	IoT connectors

Probes	ULINK*pro* for code coverage analysis	ULINK*plus* for power optimization	ULINK2 for run/stop debugging

MDK Tools

The MDK Tools include all the components that you need to create, build, and debug an embedded application for Arm based microcontroller devices. **MDK-Core** consists of the Keil µVision IDE and debugger with leading support for Cortex-M processor-based microcontroller devices.

MDK includes the **Arm C/C++ Compiler** with assembler, linker, and highly optimize run-time libraries tailored for optimum code size and performance. Arm Compiler version 6 is based on the innovative LLVM technology and supports the latest C language standards including C++11 and C++14. It is also available with a TÜV certified qualification kit for safety applications, as well as long-term support and maintenance.

Software Packs

Software packs contain device support, CMSIS components, middleware, board support, code templates, and example projects. They may be added any time to MDK-Core, making new device support and middleware updates independent from the toolchain. µVision IDE manages the provided software components that are available for the application as building blocks.

MDK Editions

The product selector, available at **keil.com/editions**, gives an overview of the features enabled in each edition:

- **MDK-Lite** is code size restricted to 32 KByte and intended for product evaluation, small projects, and the educational market.

- **MDK-Essential** supports all Cortex-M processor-based microcontrollers up to Cortex-M55.

- **MDK-Plus** adds middleware libraries for IPv4 networking, USB Device, File System, and Graphics. It supports Arm Cortex-M, Arm Cortex-R4, ARM7, and ARM9 processor-based microcontrollers.

- **MDK-Professional** contains all features of **MDK-Plus**. In addition, it supports IPv4/IPv6 dual-stack networking and a USB Host stack. It also gives access to the safety-qualified version of the Arm Compiler with all required documents and certificates.

License Types

Apart from **MDK-Lite**, all MDK editions require activation using a license code. The following licenses types are available:

1. **Single-user license** (node-locked) grants the right to use the product by one developer on two computers at the same time.

2. **Floating-user license** or **FlexNet license** grants the right to use the product on different computers by several developers at the same time.

For further details, refer to the *Licensing User's Guide* at **keil.com/support/man/docs/license**.

Installation

Software and hardware requirements

MDK has the following minimum hardware and software requirements:

- A PC running a current Microsoft Windows desktop operating system (32-bit or 64-bit)
- 4 GB RAM and 8 GB hard-disk space
- 1280 x 800 or higher screen resolution; a mouse or other pointing device

Exact requirements can be found at **keil.com/system-requirements/**

Install MDK

Download **MDK** from **keil.com/demo/eval/arm.htm** and run the installer.

Follow the instructions to install MDK on your local computer. The installation also adds the software packs for **Arm CMSIS, Arm Compiler** and **MDK-Middleware**.

After the MDK installation is complete, the **Pack Installer** starts automatically, which allows you to add supplementary software packs. As a minimum, you need to install a software pack that supports your target microcontroller device.

NOTE
MDK version 5 can use MDK version 4 projects after installation of the legacy support from keil.com/mdk5/legacy. This adds support for Arm7, Arm9, and Cortex-R processor-based devices.
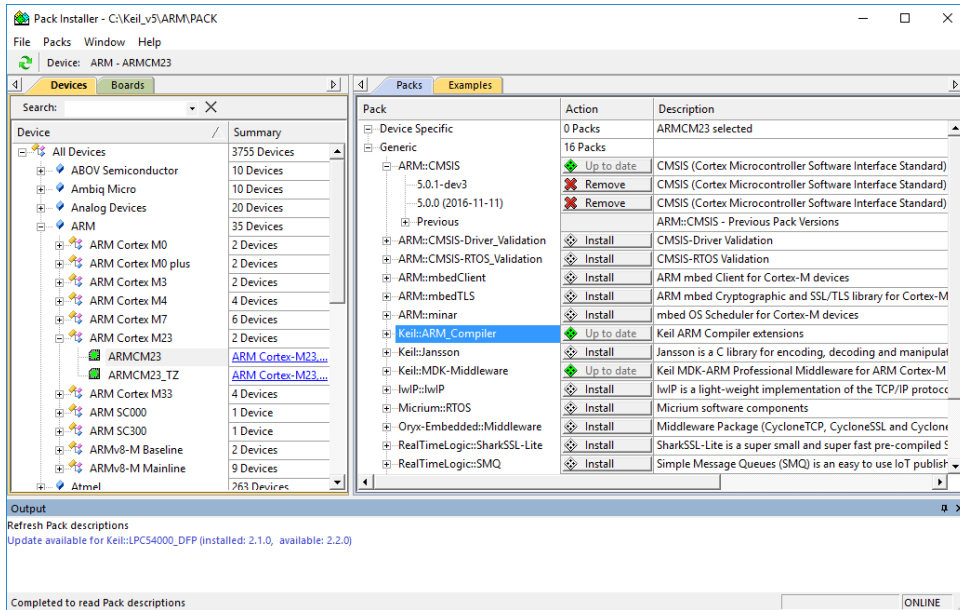
Install Software Packs

The **Pack Installer** manages software packs on the local computer. The software packs are stored in the pack root folder (default: *%localappdata%\Arm\Packs*).

The **Pack Installer** runs automatically during the installation, but also can be run from µVision using the menu item **Project – Manage – Pack Installer**. To get access to devices and example projects, install the software pack related to your target device or evaluation board.

NOTE
To obtain information of published software packs the Pack Installer connects to *keil.com/pack*.

The status bar at the bottom of the Pack Installer, shows information about the Internet connection and the installation progress.

TIP: The device database lists all supported devices and provides download access to the related software packs. It is available at **https://developer.arm.com/embedded/cmsis/cmsis-packs/devices**. If the Pack Installer does not have Internet access, you can manually install software packs using the menu command **File – Import** or by double-clicking *.PACK files.

Manage local repositories

While developing a software pack, it is useful to quickly verify how it works in a µVision project without re-building and re-installing the pack after every modification.

For this purpose, the folder with the pack's content shall be added to the list of **managed local repositories**. To do this use the Pack Installer menu **File - Manage Local Repositories...**, click **Add...**, select the PDSC file in the pack folder and press **OK**:
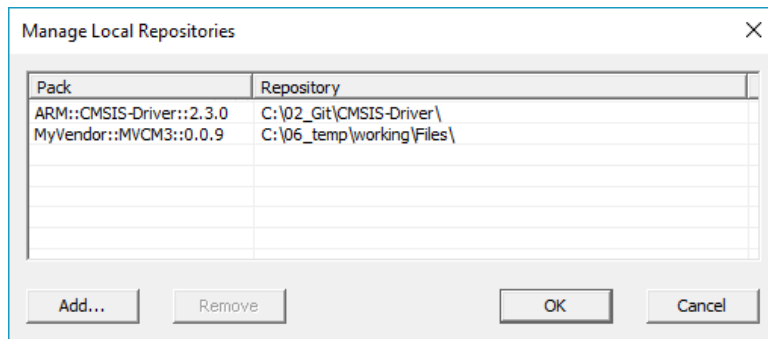
To ensure that the changes to the pack are applied in the project reload the packs using µVision menu **Project - Manage - Reload Software Packs**.

MDK-Professional Trial License

MDK has a built-in functionality to request a thirty-day trial license for MDK-Professional. This removes the code size limits and you can explore and test the comprehensive middleware.

Start µVision with administration rights.

☞ In µVision, go to **File – License Management...** and click **Evaluate MDK Professional**

A window opens that shows you the data that is submitted to the Arm Keil server to generate your personal license key:

When you click OK, your browser opens, and you are directed to a registration page. Confirm that the information is correct by clicking the **Submit** button:

Request a free 30 day trial of MDK-Professional

Please validate the information on the following form and submit.
Please make certain your e-mail address is valid. We will send you a License ID Code(LIC) vie e-mail.
Email is sent from licmgr@keil.com so make sure any spam blocker you use is configured to allow this address.

Enter Your Contact Information Below

|	
Computer ID (CID):	
First Name:	
Last Name:	
E-mail:	
Company:	

☐ I would like to get assistance during my evaluation.
NOTICE:
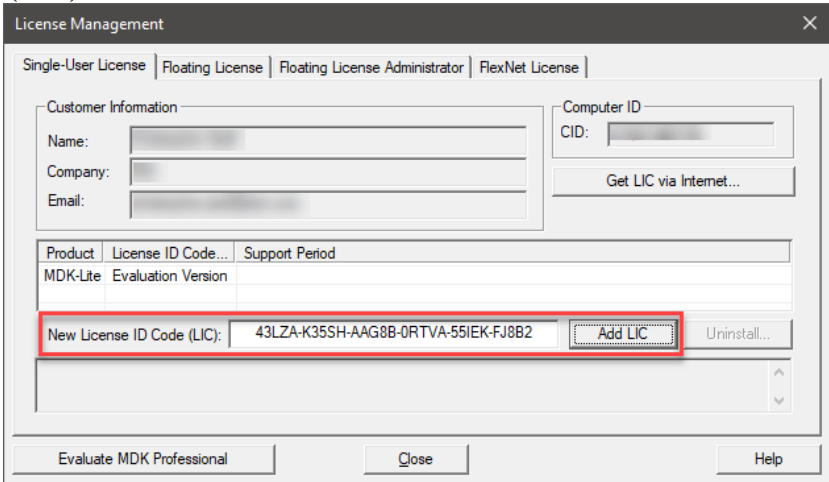If you select this check box, you **agree** that an Arm technical person may contact you via e-mail

We will process your information in accordance with the Forum section of our Privacy Policy

Please complete the Captcha check

✓ I'm not a robot
reCAPTCHA
Privacy - Terms

Submit   **Reset**

Once done, you receive an email from the Keil web server with the license number for your evaluation.

In µVision's License Management dialog, enter the value in the **New License ID Code (LIC)** field and click **Add LIC**:

License Management   ✕

Single-User License | Floating License | Floating License Administrator | FlexNet License

Customer Information
Name:
Company:
Email:

Computer ID
CID:

Get LIC via Internet...

Product	License ID Code...	Support Period
MDK-Lite	Evaluation Version	

New License ID Code (LIC):   43LZA-K35SH-AAG8B-0RTVA-55IEK-FJ8B2   Add LIC   Uninstall...

Evaluate MDK Professional   Close   Help

Now you can use MDK-Professional for thirty days.

Verify Installation using Example Projects

Once you have selected, downloaded, and installed a software pack for your device, you can verify your installation using one of the examples provided in the software pack. To verify the software pack installation, we recommend using a *Blinky* example, which typically flashes LEDs on a target board.

TIP: Review the getting started video on **keil.com/mdk5/install** that explains how to connect and work with an evaluation kit.

Copy an Example Project

In the Pack Installer, select the tab **Examples**. Use Search fielad in the toolbar to narrow the list of examples.
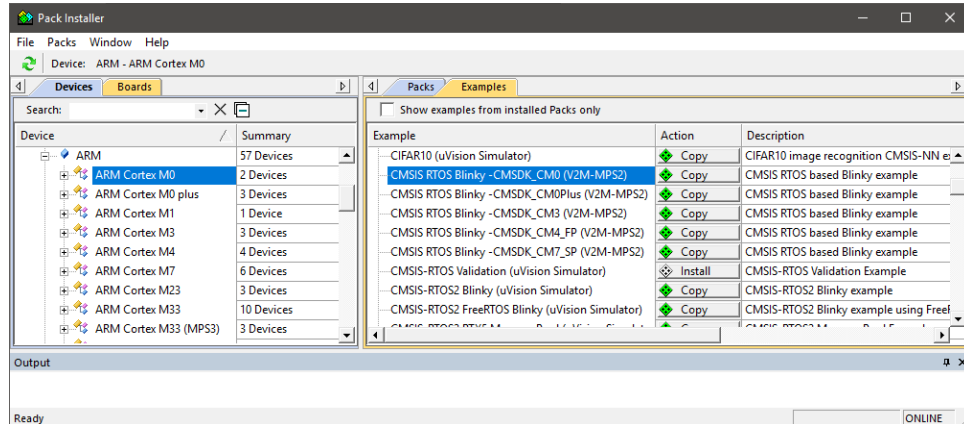
Click **Copy** and enter the **Destination Folder** name of your working directory.
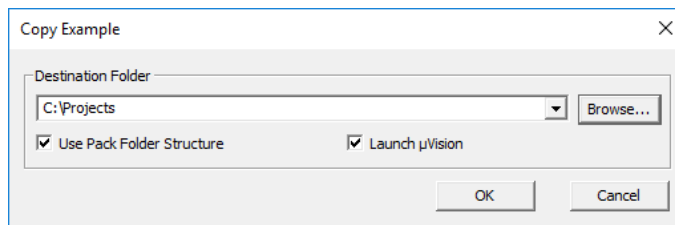
NOTE
You must copy the example projects to a working directory of your choice.

▪ Enable **Launch μVision** to open the example project directly in the IDE.

- Enable **Use Pack Folder Structure** to copy example projects into a common folder. This avoids overwriting files from other example projects. Disable **Use Pack Folder Structure** to reduce the complexity of the example path.

- Click **OK** to start the copy process.

Use an Example Application with µVision

µVision starts and loads the example project where you can:

- Build the application, which compiles and links the related source files.

- Download the application, typically to on-chip Flash ROM of a device.

- Run the application on the target hardware using a debugger.

The step-by-step instructions show you how to execute these tasks. After copying the example, µVision starts and looks like the picture below.
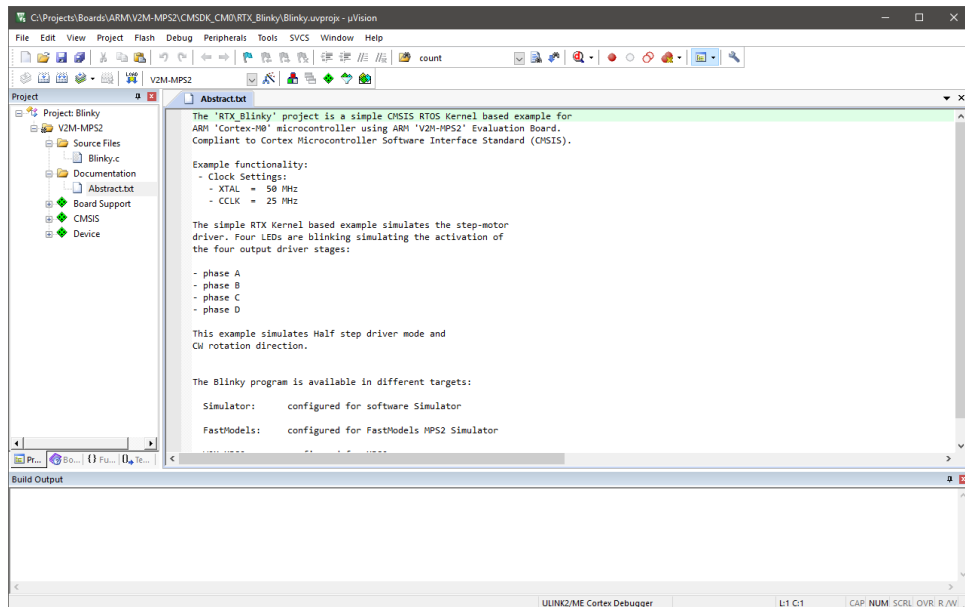
TIP: Most example projects contain an *Abstract.txt* file with essential information about the operation and hardware configuration.

Build the Application

Build the application using the toolbar button **Rebuild.**

The **Build Output** window shows information about the build process. An error-free build shows information about the program size.

```
Build Output                                                              ⌐ ☒
*** Using Compiler 'V5.06 update 6 (build 750)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
Rebuild target 'V2M-MPS2'
compiling GLCD_Fonts.c...
compiling Blinky.c...
compiling GLCD_V2M-MPS2.c...
compiling LED_V2M-MPS2.c...
assembling startup_CMSDK_CM0.s...
compiling RTX_Conf_CM.c...
compiling system_CMSDK_CM0.c...
linking...
Program Size: Code=8976 RO-data=6732 RW-data=92 ZI-data=3620
".\Out\Blinky.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:01
```

Download the Application

Connect the target hardware to your computer using a *debug adapter* that typically connects via USB. Several evaluation boards provide an on-board debug adapter.

Now, review the settings for the debug adapter. Typically, example projects are pre-configured for evaluation kits; thus, you do not need to modify these settings.

Click **Options for Target** on the toolbar and select the **Debug** tab. Verify that the correct debug adapter of the evaluation board you are using is selected and enabled. For example, **CMSIS-DAP Debugger** is a common on-board debug adapter for various starter kits.
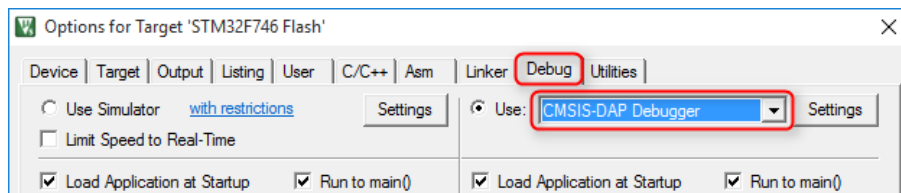
☞ Enable **Load Application at Startup** for loading the application into the µVision debugger whenever a debugging session is started.

Enable **Run to main()** for executing the instructions up to the first executable statement of the main() function. The instructions are executed upon each reset.

TIP: Click the button **Settings** to verify communication settings and diagnose problems with your target hardware. For further details, click the button **Help** in the dialogs. If you have any problems, refer to the user guide of the starter kit.

Click **Download** on the toolbar to load the application to your target hardware.

```
Build Output                                                                    x
Load "C:\\Workspaces\\MDK\\STM32\\MDK\\Boards\\ST\\STM32F746G_Discovery\\Blinky\\Flash\\Blinky.axf"
Erase Done.
Programming Done.
Verify OK.
Application running ...
Flash Load finished at 14:38:29
```

The **Build Output** window shows information about the download progress.

Run the Application

Click **Start/Stop Debug Session** on the toolbar to start debugging the application on hardware.

Click **Run** on the debug toolbar to start executing the application. LEDs should flash on the target hardware.

Access Documentation

MDK provides online manuals and context-sensitive help. The µVision **Help** menu opens the main help system that includes the *µVision User's Guide*, getting started manuals, compiler, linker and assembler reference guides.

Many dialogs have context-sensitive **Help** buttons that access the documentation and explain dialog options and settings.

You can press **F1** in the editor to access help on language elements like RTOS functions, compiler directives, or library routines. Use **F1** in the command line of the **Output** window for help on debug commands, and some error and warning messages.

The **Books** window may include device reference guides, data sheets, or board manuals. You can even add your own documentation and enable it in the **Books** window using the menu **Project – Manage – Components, Environment, Books – Books**.

The **Manage Run-Time Environment** dialog offers access to documentation via links in the *Description* column.

In the **Project** window, you can right-click a software component group and open the documentation of the corresponding element.

Access the **µVision User's Guide** on-line: **keil.com/support/man/docs/uv4**.

Request Assistance

If you have suggestions or you have discovered an issue with the software, please report them to us. Support information can be found at **keil.com/support**.

When reporting an issue, include your license code (if you have one) and product version, available from the µVision menu **Help – About**.

On-line Learning

Our **keil.com/learn** website helps you to learn more about the programming of Arm Cortex-based microcontrollers. It contains tutorials, further documentation, as well as useful links to other websites.

Selected videos showing the tools and different aspects of software development are available at **keil.com/video**.

CMSIS

The **Cortex Microcontroller Software Interface Standard** (CMSIS) provides a standardized software framework for embedded applications that run on Cortex based microcontrollers. CMSIS enables consistent and simple software interfaces to the processor and the peripherals, simplifying software reuse, reducing the learning curve for microcontroller developers.

CMSIS is available under an Apache 2.0 license and is publicly developed on GitHub: **https://github.com/ARM-software/CMSIS_5**.

NOTE
This chapter is a reference section. The chapter Create Applications on page 44 shows you how to use CMSIS for creating application code.

CMSIS provides a common approach to interface peripherals, real-time operating systems, and middleware components. The CMSIS application software components are:

- **CMSIS-CORE**: Defines the API for the Cortex-M processor core and peripherals and includes a consistent system startup code. The software components **::CMSIS:CORE** and **::Device:Startup** are all you need to create and run applications on the native processor that uses exceptions, interrupts, and device peripherals.

- **CMSIS-RTOS2:** Provides a standardized real-time operating system API and enables software templates, middleware libraries, and other components that can work across supported RTOS systems. This manual explains the usage of the Keil RTX5 implementation.

- **CMSIS-DSP**: Is a library collection for digital signal processing (DSP) with over 60 Functions for various data types: fix-point (fractional q7, q15, q31) and single precision floating-point (32-bit).

- **CMSIS-Driver**: Is a software API that describes peripheral driver interfaces for middleware components and user applications. The CMSIS-Driver API is designed to be generic and independent of a specific RTOS making it reusable across a wide range of supported microcontroller devices.

- **CMSIS-Zone**: Defines methods to describe and partition system resources into multiple projects and execution areas. The system resources may include multiple processors, memory areas, peripherals and related interrupts.

CMSIS-CORE

This section explains the usage of CMSIS-CORE in applications that run natively on a Cortex-M processor. This type of operation is known as *bare-metal*, because it does not use a real-time operating system.

Using CMSIS-CORE

A native Cortex-M application with CMSIS uses the software component **::CMSIS:CORE**, which should be used together with the software component **::Device:Startup**. These components provide the following key files:

The *startup_<device>.s* file with reset handler and exception vectors.

The *system_<device>.c* configuration file for basic device setup.

The *<device>.h* header file for user code access to the microcontroller device. This file is included in C source files and defines:

- Peripheral access with standardized register layout.

- Access to interrupts and exceptions, and the Nested Interrupt Vector Controller (NVIC).

- Intrinsic functions to generate special instructions, for example to activate sleep mode.

- Systick timer (SYSTICK) functions to configure and start a periodic timer interrupt.

- Debug access for *printf*-style I/O and ITM communication via on-chip CoreSight.

The *partition_<device>.h* header file contains the initial setup of the TrustZone hardware in an Armv8-M system (refer to section **Secure/non-secure programming**).
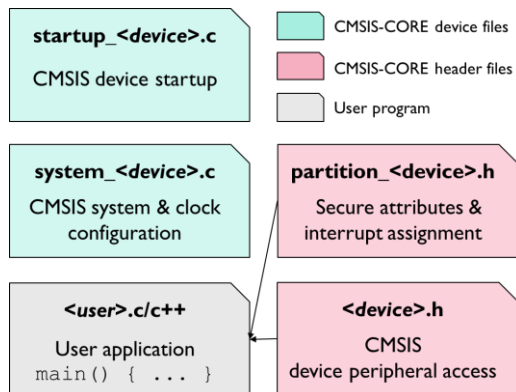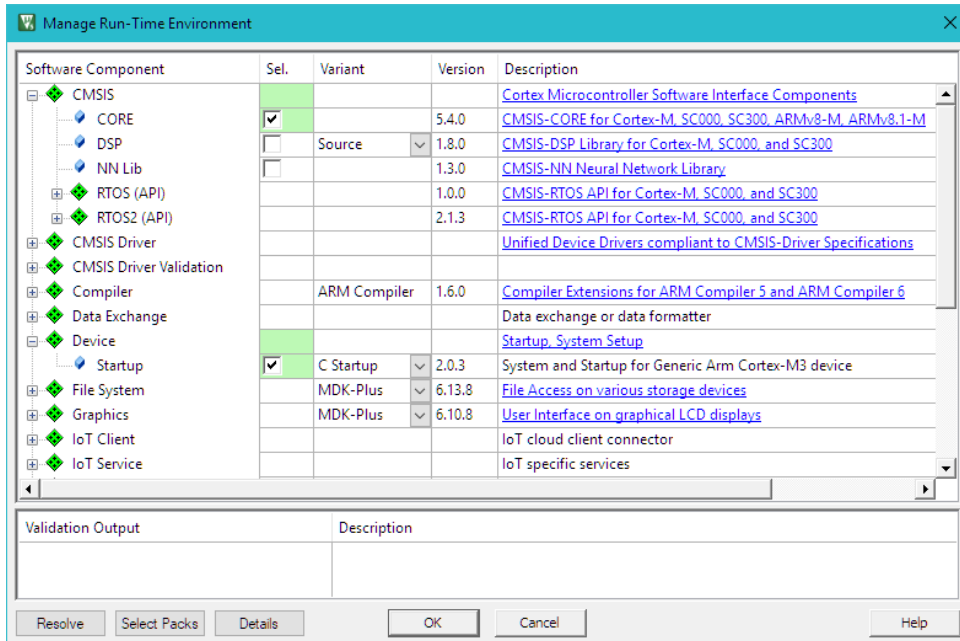

NOTE
In actual file names, <device> is the name of the microcontroller device.

Adding CMSIS-CORE Components to the Project

The files for the components **::CMSIS:CORE** and **::Device:Startup** are added
to a project using the µVision dialog **Manage Run-Time Environment**. Just
select the software components as shown below:

The µVision environment adds the related files.

Source Code Example

The following source code lines show the usage of the CMSIS-CORE layer.

Example of using the CMSIS-CORE layer

```c
#include "stm32f4xx.h"                 // File name depends on device used

uint32_t volatile msTicks;            // Counter for millisecond Interval
uint32_t volatile frequency;          // Frequency for timer


void SysTick_Handler (void) {          // SysTick Interrupt Handler
  msTicks++;                           // Increment Counter
}

void WaitForTick (void) {
  uint32_t curTicks;
  curTicks = msTicks;                  // Save Current SysTick Value
  while (msTicks == curTicks) {        // Wait for next SysTick Interrupt
    __WFE ();                          // Power-Down until next Event
  }
```

```
}

void TIM1_UP_IRQHandler (void) {      // Timer Interrupt Handler
  ; // Add user code here
}

void timer1_init(int frequency) {     // Set up Timer (device specific)
  NVIC_SetPriority (TIM1_UP_IRQn, 1); // Set Timer priority
  NVIC_EnableIRQ (TIM1_UP_IRQn);      // Enable Timer Interrupt
}

// Configure & Initialize the MCU
void Device_Initialization (void) {
  if (SysTick_Config (SystemCoreClock / 1000)) {     // SysTick 1ms
    : // Handle Error
  }
  timer1_init (frequency);          // Setup device-specific timer
}

// The processor clock is initialized by CMSIS startup + system file
int main (void) {                   // User application starts here
  Device_Initialization ();         // Configure & Initialize MCU

  while (1) {                       // Endless Loop (the Super-Loop)
    __disable_irq ();               // Disable all interrupts
    // Get_InputValues ();
    __enable_irq ();                // Enable all interrupts
    // Process_Values ();
    WaitForTick ();                 // Synchronize to SysTick Timer
  }
}
```

For more information, right-click the group CMSIS in the Project window, and choose **Open Documentation**, or refer to the CMSIS-CORE documentation **arm-software.github.io/CMSIS_5/Core/html/index.html**.

CMSIS-RTOS2

This section introduces the CMSIS-RTOS2 API and the Keil RTX5 real-time operating system, describes their features and advantages, and explains configuration settings of Keil RTX5.

NOTE
MDK is compatible with many third-party RTOS solutions. However, CMSIS-RTOS Keil RTX5 is feature-rich and tailored towards the requirements of deeply embedded systems. Also, it is well integrated into MDK.
While CMSIS-RTOS Keil RTX5 is open source, a variant certified for functional safety applications is available as well. See keil.com/fusa-rts for details.

Software Concepts

There are two basic design concepts for embedded applications:

- **Infinite Loop Design:** involves running the program as an endless loop. Program functions (threads) are called from within the loop, while interrupt service routines (ISRs) perform time-critical jobs including some data processing.

- **RTOS Design:** involves running several threads with a **real-time operating system (RTOS)**. The RTOS provides inter-thread communication and time management functions. A pre-emptive RTOS reduces the complexity of interrupt functions, because high-priority threads can perform time-critical data processing.

Infinite Loop Design

Running an embedded program in an endless loop is an adequate solution for simple embedded applications. Time-critical functions, typically triggered by hardware interrupts, execute in an ISR that also performs any required data processing. The main loop contains only basic operations that are not time-critical and run in the background.

Advantages of an RTOS Kernel

RTOS kernels, like the Keil RTX5, are based on the idea of parallel execution threads (tasks). As in the real world, your application will have to fulfill multiple different tasks. An RTOS-based application recreates this model in your software with various benefits:

- Thread priority and run-time scheduling is reliably handled by the RTOS.

- The RTOS provides a well-defined interface for communication between threads.

- A pre-emptive multi-tasking concept simplifies the progressive enhancement of an application even across a larger development team. New functionality can be added without risking the response time of more critical threads.

- Infinite loop software concepts often poll for occurred interrupts. In contrast, RTOS kernels themselves are interrupt driven and can largely eliminate polling. This allows the CPU to sleep or process threads more often.

Modern RTOS kernels are transparent to the interrupt system, which is mandatory for systems with hard real-time requirements. Communication facilities can be used for IRQ-to-task communication.

Using Keil RTX5

The Keil RTX 5 implements the CMSIS-RTOS API v2 as a native RTOS interface for Cortex-M processor-based devices.

Once the execution reaches *main()*, there is a recommended order to initialize the hardware and start the kernel. The *main()* of your application should implement at least the following in the given order:

1. Initialization and configuration of hardware including peripheral, memory, pin, clock, and interrupt system.

2. Update **SystemCoreClock** using the respective CMSIS-CORE function.

3. Initialize CMSIS-RTOS kernel using **osKernelInitialize**.

4. Optionally, create a new thread *app_main*, which is used as a main thread using **osThreadNew**. Alternatively, threads can be created in *main()* directly.

5. Start RTOS scheduler using **osKernelStart**. osKernelStart does not return in case of successful execution. Any application code after *osKernelStart* will not be executed unless *osKernelStart* fails.

The software component **::CMSIS:RTOS2 (API):Keil RTX5** must be used together with the components **::CMSIS:CORE** and **::Device:Startup** explained in **Using CMSIS-CORE** section.

Central Keil RTX5 files are:

The header file *cmsis_os2.h* exposes the RTX functionality to the user application via CMSIS-RTOS2 API.

The configuration files *RTX_Config.c/.h* define thread options, timer configurations, and RTX kernel settings.

The file *RTX_<core>.lib* contains the library with RTOS functions and gets included when RTX5 is used in a library variant. In this case *rtx_lib.c* file contains the RTX5 library configuration.

Section **Project with CMSIS-RTOS2** gives an example how to setup a project based on Keil RTX5.

Adding Keil RTX5 Components to the Project

The files for the components **::CMSIS:RTOS2 (API):Keil RTX5**, **::CMSIS:CORE** and **::Device:Startup** are added to a project using the μVision dialog **Manage Run-Time Environment**. Just select the software components as shown below:
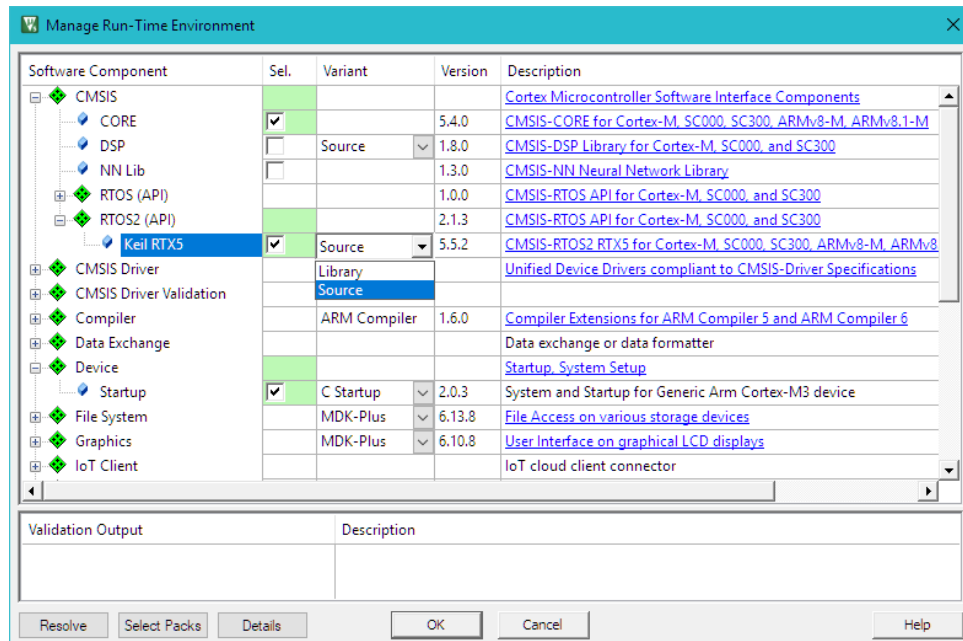
Library variant of Keil RTX5 has more compact code, while source variant allows full program debug and supports RTOS-aware debugging via **Event Recorder** support.

CMSIS-RTOS2 API Functions

The file *cmsis_os2.h* is a standard header file that defines interfaces to every CMSIS-RTOS API v2 compliant RTOS.

All definitions in the header file are prefixed with **os** to give a unique name space for the CMSIS-RTOS functions.

All definitions and functions that belong to a module are grouped and have a common prefix, for example, **osThread** for threads.

Refer to section **Reference: CMSIS-RTOS2 API** of the online documentation at **arm-software.github.io/CMSIS_5/RTOS2/html/index.html**, for more information.

Keil RTX5 Configuration

The file *RTX_Config.h* contains configuration parameters for Keil RTX5. A copy of this file is part of every project using the RTX component.
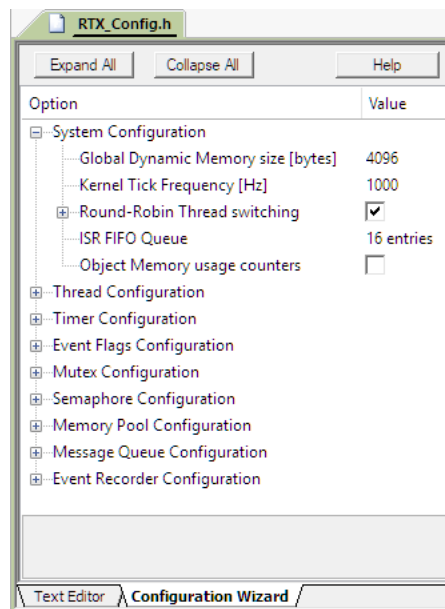
You can set various system parameters such as the Tick Timer frequency, Round-Robin time slice, specify configurations for specific RTOS objects, such as

threads, timers, event flags, mutexes, semaphores, memory pools, and message queues, as well configure Event Recorder operation.

For more information about configuration options, open the RTX documentation from the **Manage Run-Time Environment** window. The section **Configure RTX v5** describes all available settings:

arm-software.github.io/CMSIS_5/RTOS2/html/config_rtx5.html

CMSIS-RTOS User Code Templates

MDK provides user code templates you can use to create C source code for the application.

☞ In the **Project** window, right click a group, select **Add New Item to Group**, choose **User Code Template,** select any template and click **Add**.
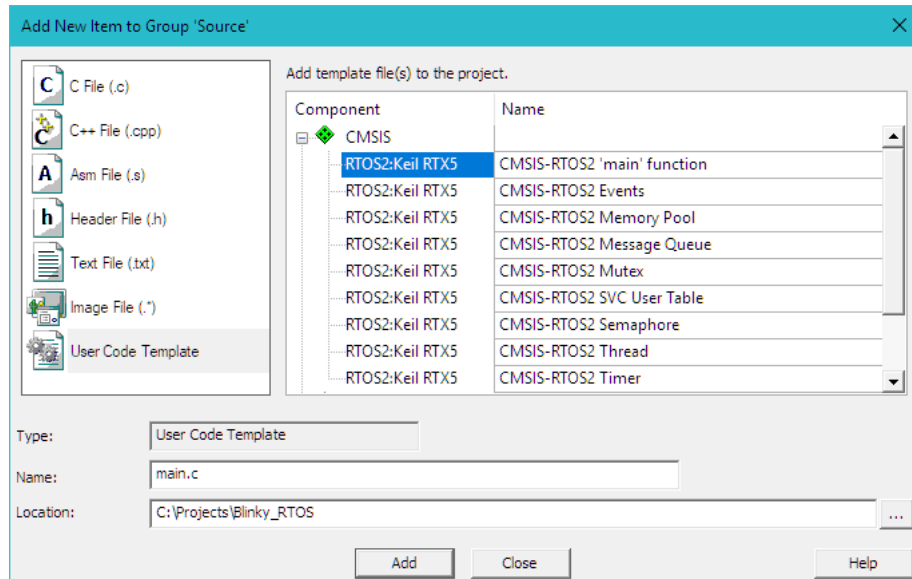
Source Code Example

Once these files are part of the project, developers can start using the CMSIS-RTOS2 RTX functions.

The code example shows the use of CMSIS-RTOS RTX functions.

```c
#include "cmsis_os2.h"                      // CMSIS RTOS2 header file

void app_main (void *argument) {

  tid_phaseA = osThreadNew(phaseA, NULL, NULL);
  osDelay(osWaitForever);
  while(1);
}

int main (void) {

  // System Initialization
  SystemCoreClockUpdate();

  osKernelInitialize();                 // Initialize CMSIS-RTOS
  osThreadNew(app_main, NULL, NULL);    // Create application main thread
  if (osKernelGetState() == osKernelReady) {
    osKernelStart();                    // Start thread execution
  }

while(1);
}
```

Section **Project with CMSIS-RTOS2** explains in details how to setup an RTOS-based application using Keil RTX5.

Component Viewer for RTX RTOS

Keil RTX5 comes with an SCVD file for the **Component Viewer** for RTOS aware debugging. In the debugger, open **View – Watch Windows – RTX RTOS**. This window shows system state information and the running threads.

The **System** property shows general information about the RTOS configuration in the application.

The **Threads** property shows details about thread execution of the application. For each thread, it shows information about priority, execution state and stack usage.

If the option **Stack usage watermark** is enabled for **Thread Configuration** in the file *RTX_Config.h,* the field **Stack** shows the stack load. This allows you to:

- ▪ Identify stack overflows during thread execution *or*

- ▪ Optimize and reduce the stack space used for threads.

RTX RTOS	
Property	Value
System	
Kernel ID	RTX V5.5.1
Kernel State	osKernelRunning
Kernel Tick Count	201385
Kernel Tick Frequency	1000
Round Robin Tick Count	0
Round Robin Timeout	5
Global Dynamic Memory	Base: 0x20000000, Size: 4096, ...
Stack Overrun Check	Enabled
Stack Usage Watermark	Enabled
Default Thread Stack Size	256
ISR FIFO Queue	Size: 16, Used: 0
Threads	
id: 0x200012D0 "osRtxIdleThread"	osThreadReady, osPriorityIdl...
id: 0x20001314 "osRtxTimerThread"	osThreadBlocked, osPriority...
id: 0x200000D8 "app_main"	osThreadRunning, osPriority...
State	osThreadRunning
Priority	osPriorityNormal
Attributes	osThreadDetached
Stack	Used: 9% [116], Max: 24% [288]
Used	116
Max	288
Top	0x20001AD8
Current	0x20001A64
Limit	0x20001628
Size	1200
Flags	0x00000000
Mutexes	
Message Queues	

Information about other RTX5 objects, such as mutexes, semaphores, message queues, is provided in corresponding properties as well.

NOTE
The µVision debugger also provides a view with detailed runtime information.
*Refer to **Event Recorder** on page 74 for more information.*

CMSIS-DSP

The CMSIS-DSP library is a suite of common digital signal processing (DSP) functions. The library is available in several variants optimized for different Arm Cortex-M processors.

When enabling the software component **::CMSIS:DSP** in the **Manage Run-Time Environment** dialog, the appropriate library for the selected device is automatically included into the project. It is also possible to select source-code variant,.
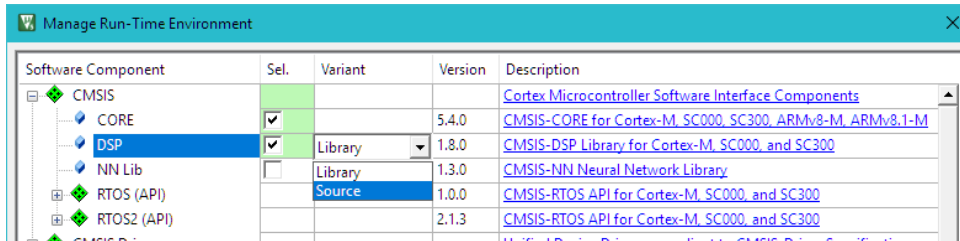
The code example below shows the use of CMSIS-DSP library functions.

Multiplication of two matrixes using DSP functions

```c
#include "arm_math.h"                    // ARM::CMSIS:DSP

const float32_t buf_A[9] = {            // Matrix A buffer and values
 1.0, 32.0, 4.0,
 1.0, 32.0, 64.0,
 1.0, 16.0, 4.0,
};

float32_t buf_AT[9];                    // Buffer for A Transpose (AT)
float32_t buf_ATmA[9] ;                 // Buffer for (AT * A)

arm_matrix_instance_f32 A;              // Matrix A
arm_matrix_instance_f32 AT;             // Matrix AT(A transpose)
arm_matrix_instance_f32 ATmA;           // Matrix ATmA( AT multiplied by A)

uint32_t rows = 3;                      // Matrix rows
uint32_t cols = 3;                      // Matrix columns

int main(void) {
  // Initialize all matrixes with rows, columns, and data array
  arm_mat_init_f32 (&A, rows, cols, (float32_t *)buf_A);  // Matrix A
  arm_mat_init_f32 (&AT, rows, cols, buf_AT);             // Matrix AT
  arm_mat_init_f32 (&ATmA, rows, cols, buf_ATmA);         // Matrix ATmA

  arm_mat_trans_f32 (&A, &AT);         // Calculate A Transpose (AT)
  arm_mat_mult_f32 (&AT, &A, &ATmA); // Multiply AT with A

  while (1);
}
```
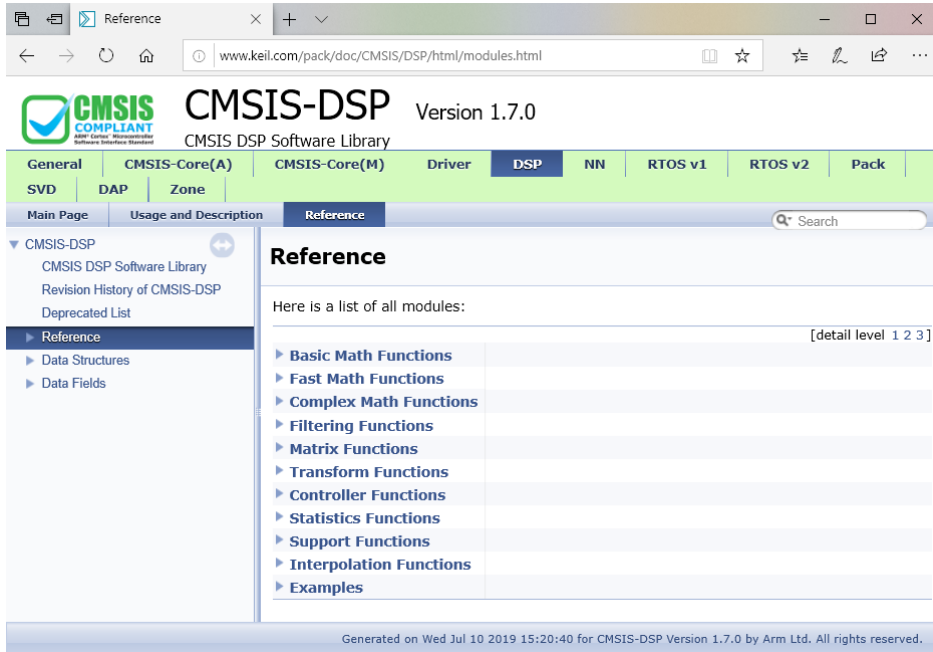
For more information, refer to the CMSIS-DSP documentation on **arm-software.github.io/CMSIS_5/DSP/html/index.html**.

CMSIS-Driver

Device-specific **CMSIS-Drivers** provide the interface between the middleware and the microcontroller peripherals. These drivers are not limited to the MDK-Middleware and are useful for various other middleware stacks to utilize those peripherals.

The device-specific drivers are usually part of the software pack that supports the microcontroller device and comply with the CMSIS-Driver standard. The device database on **https://developer.arm.com/embedded/cmsis/cmsis-packs/devices/** lists drivers included in the software pack for the device.
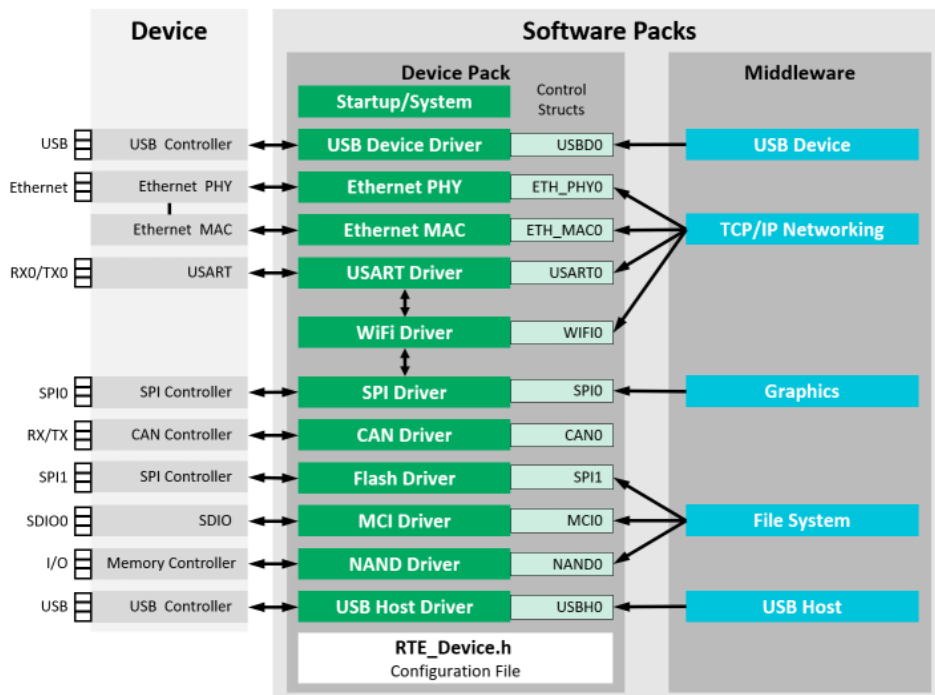
Middleware components usually have various configuration files that connect to these drivers. Depending on the device, an *RTE_Device.h* file configures the drivers to the actual pin connection of the microcontroller device. Some devices require specific third-party tools to configure the hardware correctly.

The middleware/application code connects to a driver instance via a *control struct*. The name of this *control struct* reflects the peripheral interface of the device. Drivers for most of the communication peripherals are part of the software packs that provide device support.

Use traditional C source code to implement missing drivers according the CMSIS-Driver standard.

Refer to **arm-software.github.io/CMSIS_5/Driver/html/index.html** for detailed information about the API interface of these CMSIS drivers.

ARM::CMSIS-Driver pack contains example CMSIS-Driver implementations for such interfaces as WiFi, Ethernet, Flash, I2C and SPI.

Configuration

There are multiple ways to configure a CMSIS-Driver. The classical method is using the *RTE_Device.h* file that comes with the device support.

Other devices may be configured using third party graphical configuration tools that allow the user to configure the device pin locations and the corresponding drivers. Usually, these configuration tools automatically create the required C code for import into the µVision project.

Using RTE_Device.h

For most devices, the *RTE_Device.h* file configures the drivers to the actual pin connection of the microcontroller device:
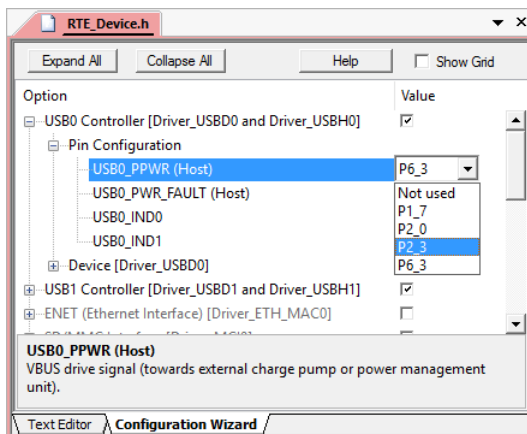
Using the **Configuration Wizard** view, you can configure the driver interfaces in a graphical mode without the need to edit manually the #defines in this header file.

Using STM32CubeMX

MDK supports CMSIS-Driver configuration for STM32 devices using STM32CubeMX. This graphical software configuration tool allows you to generate C initialization code using graphical wizards for STMicroelectronics devices.

Simply select the required CMSIS-Driver in the Manage Run-Time Environment window and choose **Device:STM32Cube Framework (API):STM32CubeMX**. This will open STM32CubeMX for device and driver configuration. Once finished, generate the configuration code and import it into µVision.

For more information, visit the online documentation at **keil.com/pack/doc/STM32Cube/General/html/index.html**.

Validation Suites for Drivers and RTOS

Software packs to validate user-written CMSIS-Drivers or a new implementation of a CMSIS-RTOS are available from **keil.com/pack**. They contain the source code and documentation of the validation suites along with required configuration files, and examples that show the usage on various target platforms.

The **CMSIS-Driver** validation suite performs the following tests:

- Generic validation of API function calls
- Validation of configuration parameters
- Validation of communication with loopback tests
- Validation of communication parameters such as baudrate
- Validation of event functions

The test results can be printed to a console, output via ITM printf, or output to a memory buffer. Refer to the **Driver Validation** section in the documentation at **arm-software.github.io/CMSIS_5/Driver/html/driverValidation.html**.

The **CMSIS-RTOS** validation suite performs generic validation of various RTOS features. The test cases verify the functional behavior, test invalid parameters and call management functions from ISR.

The validation output can be printed to a console, output via ITM printf, or output to a memory buffer. Refer to the section **RTOS Validation** in the documentation at **arm-software.github.io/CMSIS_5/RTOS2/html/rtosValidation.html**.

Software Components

The development of complex embedded applications requires a modular architecture with multiple own and third-party components used. MDK and CMSIS allow to easily integrate and maintain software components in your projects.

Use Software Packs

Software packs contain information about microcontroller devices and software components that are available for the application as building blocks.

The device information pre-configures development tools for you and shows only the options that are relevant for the selected device.

Start µVision and use the menu **Project - New µVision Project**. After you have selected a project directory and specified the project name, select a target device.
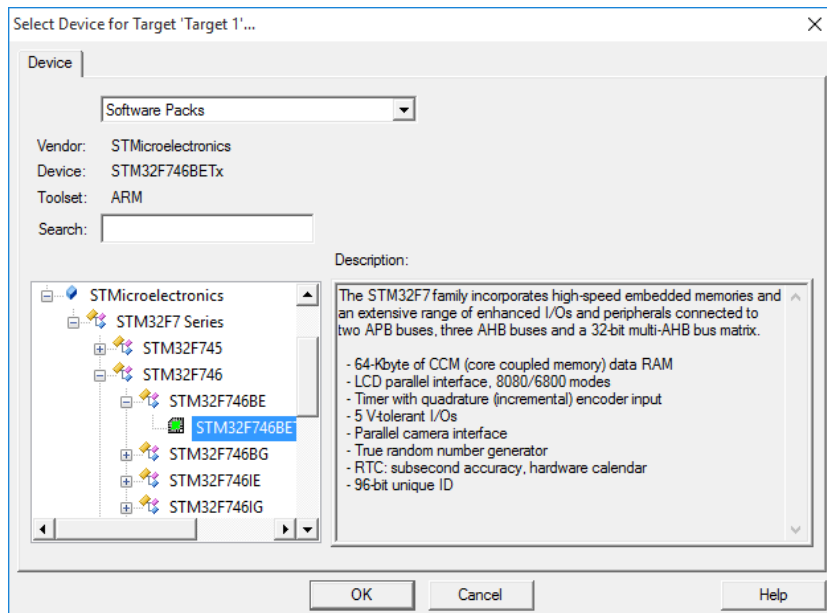
TIP: Only devices that are part of the installed software packs are shown. If you are missing a device, use the Pack Installer to add the related software pack. The search box helps you to narrow down the list of devices.

❖ After selecting the device, the **Manage Run-Time Environment** window shows the related software components for this device.
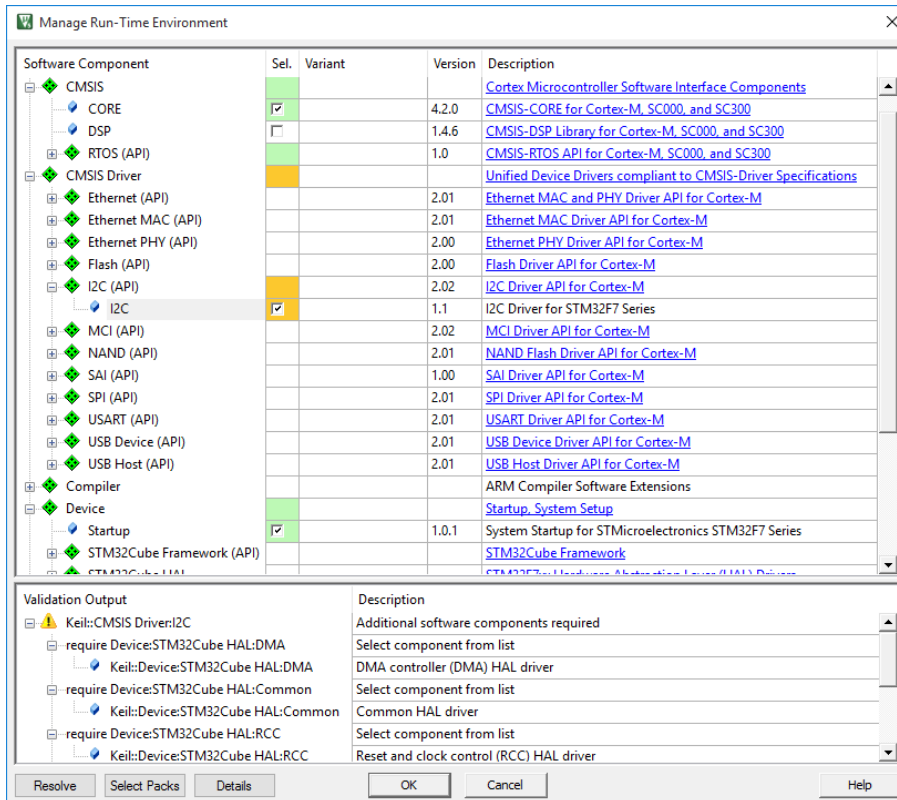
TIP: The links in the column *Description* provide access to the documentation of each software component.

Software Component Overview

The following table shows the software components included with a typical MDK installation. Depending on your MDK edition and selected device, some of these software components might not be available in the Manage Run-Time Environment window. In case you have installed additional software packs, more software components will be available.

Software Component	Description	Page
CMSIS	CMSIS interface components, such as CORE, DSP, and CMSIS-RTOS.	19
CMSIS Driver	Unified device drivers for middleware and user applications.	19
Compiler	Arm Compiler specific software components to retarget I/O operations for example for printf style debugging. Event recorder for debugging software components and user application code.	39
Board Support	Interfaces to the peripherals of evaluation boards.	42
IoT Clients	Components for communication with cloud services.	43
Device	System startup and low-level device drivers.	58
File System	Middleware component for file access on various storage device types.	93
Graphics	Middleware component for creating graphical user interfaces.	95
Network	Middleware component for TCP/IP networking using Ethernet or serial protocols.	91
USB	Middleware component for USB Host and USB Device supporting standard USB Device classes.	94
Mbed IoT Components	Mbed libraries for secure communication and cryptography	96

Product Lifecycle Management with Software Packs

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

There are four distinct 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 phase, you normally want to use the latest software packs to be able to incorporate new features and bug fixes quickly. Before product release, you will freeze the software components to a known tested state. In the product service phase, use the fixed versions of the software components to support customers in the field.

The dialog **Select Software Packs** helps you to manage the versions of each software pack in your project:
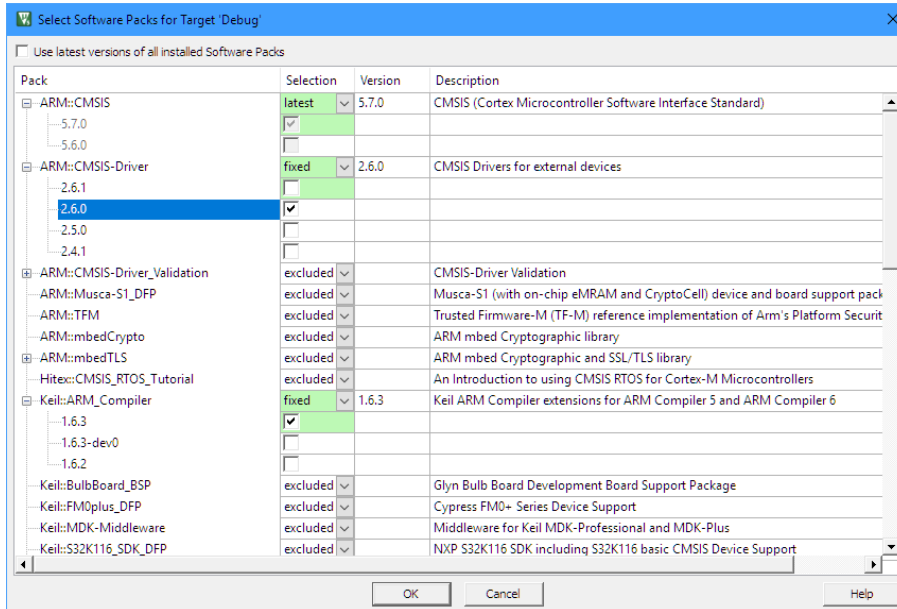
When the project is completed, disable the option **Use latest version of all installed Software Packs** and specify the software packs with the settings under **Selection**:

- **latest**: use the latest version of a software pack. Software components are updated when a newer software pack version is installed.

- **fixed**: specify an installed version of the software pack. Software components in the project target will use these versions.

- **excluded**: no software components from this software pack are used.

The colors indicate the usage of software components in the current project target:

Some software components from this pack are used.

Some software components from this pack are used, but the pack is excluded.

No software component from this pack is used.

Software Version Control Systems (SVCS)

µVision carries template files for GIT, SVN, CVS, and others to support Software Version Control Systems (SVCS).

Application note 279 "Using Git for Project Management with µVision" (**keil.com/appnotes/docs/apnt_279.asp**) describes how to establish a robust workflow for version control of projects using software packs.

Compiler:Event Recorder

Modern microcontroller applications often contain middleware components, which are normally a "black box" to the application programmer. Even when comprehensive documentation and source code is provided, analyzing of potential issues is challenging.

The software component **Compiler:Event Recorder** uses event annotations in the application code or software component libraries to provide event timing and data information while the program is executing. This event information is stored in an event buffer on the target system that is continuously read by the debug unit and displayed in the **e**vent recorder window of the µVision debugger.
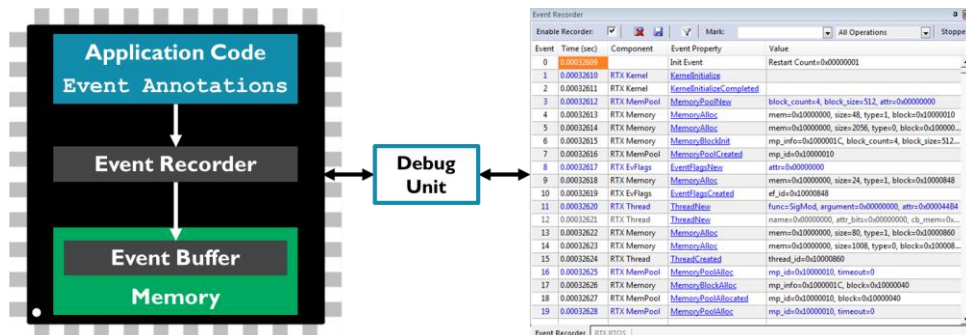
During program execution, the µVision debugger reads the content of the event buffer using a debug adapter that is connected via JTAG or SWD to the CoreSight Debug Access Port (DAP). The event recorder requires no trace hardware and can therefore be used on any Cortex-M processor-based device.

To display the data stored in the event buffer in a human readable way, you need to create a Software Component Viewer Description (SCVD) file. Refer to: **keil.com/pack/doc/compiler/EventRecorder/html/index.html**

The section **Event Recorder** on page 74 shows how to use the event recorder in a debug session.

Compiler:I/O

The software component **Compiler:I/O** allows you to retarget I/O functions of the standard C run-time library. Application code frequently uses standard I/O library functions, such as *printf()*, *scanf()*, or *fgetc()* to perform input/output operations.

The structure of these functions in the standard Arm Compiler C run-time library is:
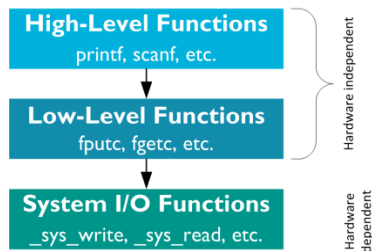
The high-level and low-level functions are not target-dependent and use the system I/O functions to interface with hardware.

The MicroLib of the Arm Compiler C run-time library interfaces with the hardware via low-level functions. The MicroLib implements a reduced set of high-level functions and therefore does not implement system I/O functions.

The software component **Compiler:I/O** retargets the I/O functions for the various standard I/O channels: File, STDERR, STDIN, STDOUT, and TTY:
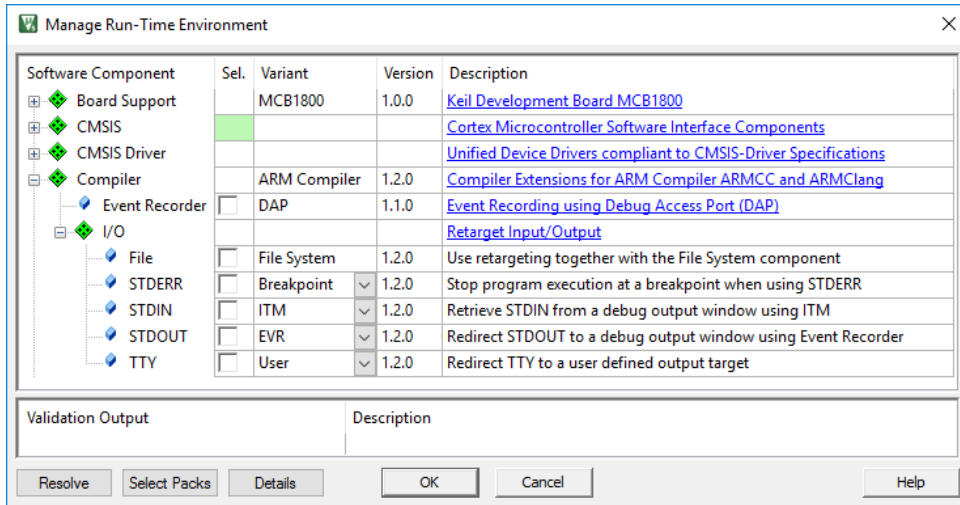
I/O Channel	Description
File	Channel for all file related operations (*fscanf, fprintf, fopen, fclose*, etc.)
STDERR	Standard error stream of the application to output diagnostic messages.
STDIN	Standard input stream going into the application (*scanf* etc.).
STDOUT	Standard output stream of the application (*printf* etc.).
TTY	Teletypewriter which is the last resort for an error output.

The variant selection allows you to change the hardware interface of the I/O channel.

Variant	Description
File System	Use the File System component as the interface for File related operations
EVR	Use the event recorder to display printf debug messages
Breakpoint	When the I/O channel is used, the application stops with BKPT instruction.
ITM	Use Instrumentation Trace Macrocell (ITM) for I/O communication via the debugger.
User	Retarget I/O functions to a user defined routine (such as USART, keyboard).

The software component **Compiler** adds the file *retarget_io.c* that will be configured acording to the variant settings. For the **User** variant, user code templates are available that help you to implement your own functionality. Refer to the documentation for more information.

ITM in the Cortex-M3/M4/M7 supports *printf* style debugging. If you choose the variant **ITM**, the I/O
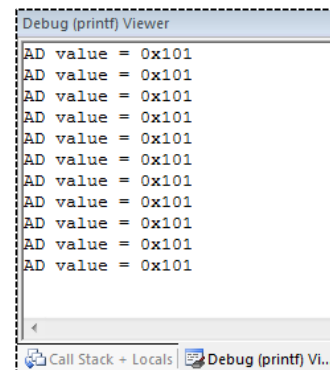
library functions perform I/O operations via the **Debug (printf) Viewer** window.

As ITM is not available in Cortex-M0/M0+ devices, you can use the event recorder to display printf debug messages. Use the **EVR** variant of the **STDOUT I/O** channel for this purpose (works with all Cortex-M based devices).

For more details refer to:
keil.com/pack/doc/compiler/RetargetIO/html/index.html

Board Support

There are a couple of interfaces that are frequently used on development boards, such as LEDs, push buttons, joysticks, A/D and D/A converters, LCDs, and touchscreens as well as external sensors such as thermometers, accelerometers, magnetometers, and gyroscopes.

The **Board Support Interface API** provides standardized access to these interfaces. This enables software developers to concentrate on their application code instead of checking device manuals for register settings to toggle a GPIO.

Many Device Family Packs (DFPs) have board support included. You can choose board support from the Manage Run-Time Environment window:

Software Component	Sel.	Variant	Version	Description
⊟ ◈ Board Support		STM32F746G-Discovery ▾	1.0.0	STMicroelectronics STM32F746G-Discovery Kit
⊟ ◈ Buttons (API)			1.00	Buttons Interface
● Buttons	☑		1.0.0	Buttons Interface for STMicroelectronics STM32F746G-Discovery Kit
⊞ ◈ Drivers				Kinetis BSP Drivers
⊞ ◈ Graphic LCD (API)			1.00	Graphic LCD Interface
⊟ ◈ LED (API)			1.00	LED Interface
● LED	☑		1.0.0	LED Interface for STMicroelectronics STM32F746G-Discovery Kit
⊞ ◈ Touchscreen (API)			1.00	Touchscreen Interface
⊞ ◈ emWin LCD (API)			1.1	emWin LCD Interface

Be sure to select the correct **Variant** to enable the correct pin configurations for your development board.

You can add board support to your custom board by creating the required support files for your board's software pack. Refer to the API documentation available at:
keil.com/pack/doc/mw/Board/html/index.html

IoT Clients

A set of MDK-Packs provides building blocks that enable secure connection from a device to a cloud provider of choice.
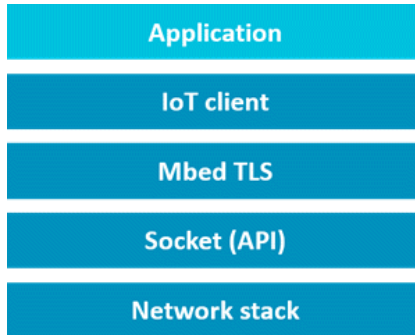
MDK-Middleware Network Component, **lwIP** and various WiFi modules (through CMSIS WiFi-Driver) are supported as underlying **network stacks**.

Reference **Socket (API)** implementations are provided in the **MDK::IoT_Socket** pack. **mbed TLS** contains required components to secure the connection. Finally, communication with a cloud service is enabled with **IoT Clients** available for the following providers:

- Amazon AWS IoT
- Google Cloud IoT
- IBM Watson IoT
- Microsoft Azure IoT Hub
- Paho MQTT (Eclipse)

The software packs are generic (device-independent) and can be found in the **Pack Installer**.

Pack	Action	Description
MDK-Packs::AWS_IoT_Device	Up to date	SDK for connecting to AWS IoT from a device using embedded C
MDK-Packs::Azure_IoT	Up to date	Microsoft Azure IoT SDKs and Libraries
MDK-Packs::cJSON	Up to date	Ultralightweight JSON parser in ANSI C
MDK-Packs::Google_IoT_Device	Up to date	Google Cloud IoT Device Connector
MDK-Packs::IoT_Socket	Up to date	Simple IP Socket (BSD like)
MDK-Packs::Paho_MQTT	Up to date	Embedded MQTT C/C++ Client Libraries
MDK-Packs::Watson_IoT_Device	Up to date	Client libraries and samples for connecting to IBM Watson IoT using Embedded C

Additional information is provided at: **keil.com/iot.**

Create Applications

This chapter guides you through the steps required to create a projects using CMSIS components described in the previous chapter.

For many popular development boards MDK already provides ready-to-use CMSIS based examples. It is always beneficial to take such an example as a starting point as explained in **Verify Installation using Example Projects** and then modify it for own application needs.

Device vendors may also provide MDK example applications in separate deliverables not indexed in the MDK Pack Installer explained in **Install Software Packs**. Development and configuration tools from device vendors may also allow export of application projects into Keil MDK format. These two options should be explored if no examples are found in MDK Pack Installer.

This chapter is structured as follows:

- Section **µVision Project from Scratch** explains how to start a new project from scratch and can be followed when there is no example applications available.

- Section **Project with CMSIS-RTOS2** shows how to easily convert an existing application with infinite loop design into Real-Time OS based system using CMSIS-RTOS2 API.

- **Device Configuration Variations** explains integrations with device vendor tools for device startup.

- Finally, section **Secure/non-secure programming** guides through the project setup for devices based on Armv8-M architecture.

NOTE
The example code in this chapter works for the MIMXRT1050-EVK evaluation board (populated with MIMXRT1052DVL6B device). Adapt the code for other starter kits or boards.

µVision Project from Scratch

This section describes the steps for setting up a new CMSIS based project from scratch:

- **Setup New µVision Project:** create a project file and select the microcontroller device along with the relevant CMSIS components.

- ▪ **Add main.c Source Code File**: Add *main.c* file to the project with initial code for *main()* function and device initialization.

- ▪ **Configure Project Options**: adjust project settings to ensure that the project can be built correctly.

- ▪ **Build the Application Project**: compile and link the application for programming it onto the target microcontroller device.

- ▪ **Using the Debugger** guides you through the steps to connect your evaluation board to the PC and to download the application to the target.

Setup New µVision Project

From the µVision menu bar, choose **Project – New µVision Project**.

☞ Select an empty folder and enter the project name, for example, *MyProject*. Click **Save**, which creates an empty project file with the specified name (*MyProject.uvprojx*).

Next, the dialog **Select Device for Target** opens.
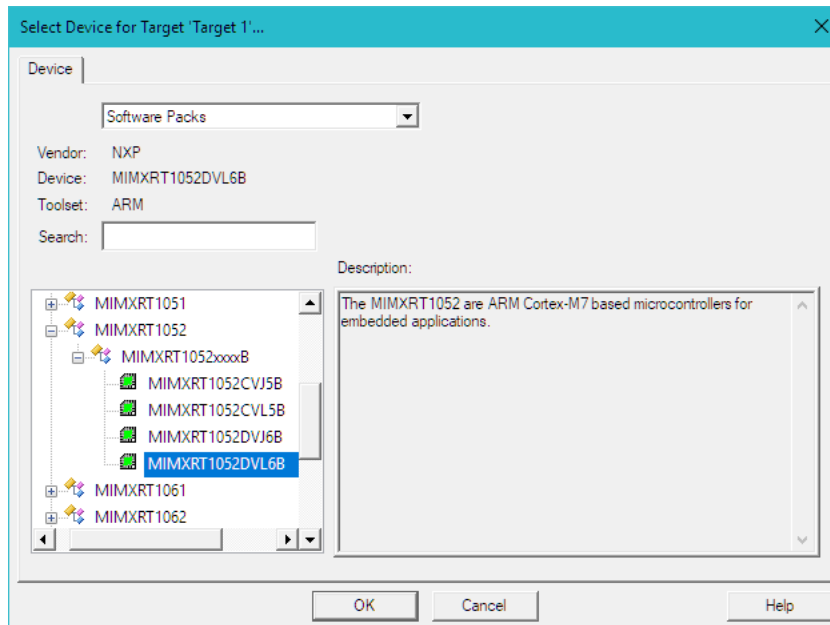
☞ Select the target device and, if necessary, the target CPU in a multi-core device. In our case this is MIMXRT1052DVL6B and click **OK**.

> TIP: If the target device is not available in the list – verify that the corresponding Device Family Pack (DFP) is installed as explained in **Install Software Packs.**

The device selection defines essential tool settings such as compiler controls, the memory layout for the linker, and the Flash programming algorithms. However, in some cases (especially for more complex devices) additional configurations are required to achieve correct project build and debug. This is explained in step **Configure Project Options**.

Then the **Manage Run-Time Environment** dialog opens and shows the software components that are installed and available for the selected device.

Following components need to be added for CMSIS-based project:

☞ Expand **::CMSIS** and enable **CORE**.

Expand **::Device::Startup** and enable one of the offered variants. In our case it is just one: **MIMXRT1052_startup**.
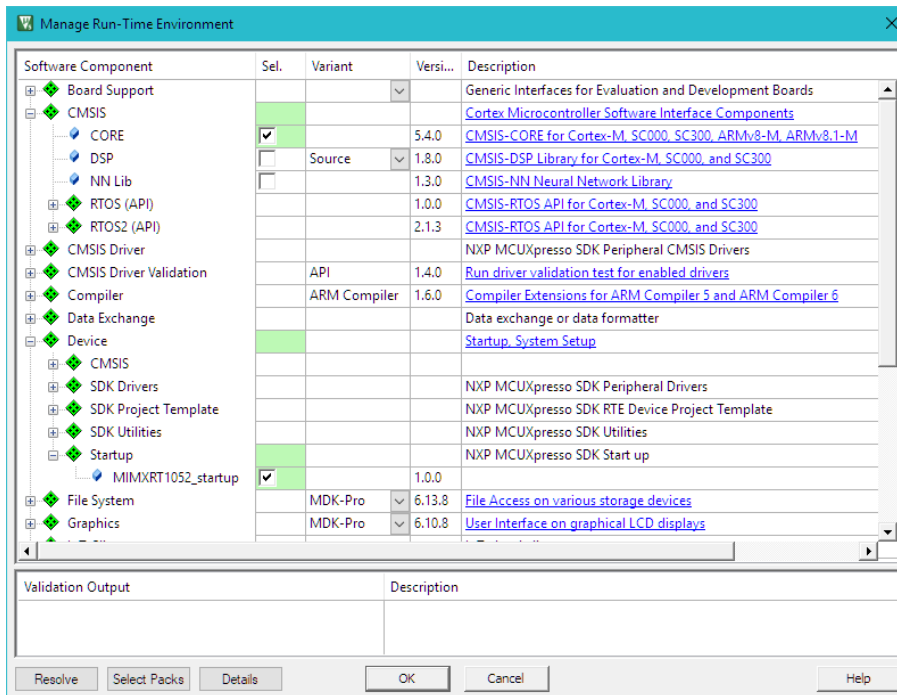
Other components can be added depending on the application needs. In our case we limit to the bare minimum.

The **Validation Output** field may show dependencies to other software components that are required based on the current selection. In such case click **Resolve** button to automatically resolve all dependencies and enable other required

TIP: A click on a message highlights the related software component.

In our example shown above there is no extra dependencies to resolve.

☞ Click **OK**.

The selected software components are included into the project together with the device startup file and CMSIS system files. The **Project** window displays the selected software components along with the related files. Double-click on a file to open it in the editor.
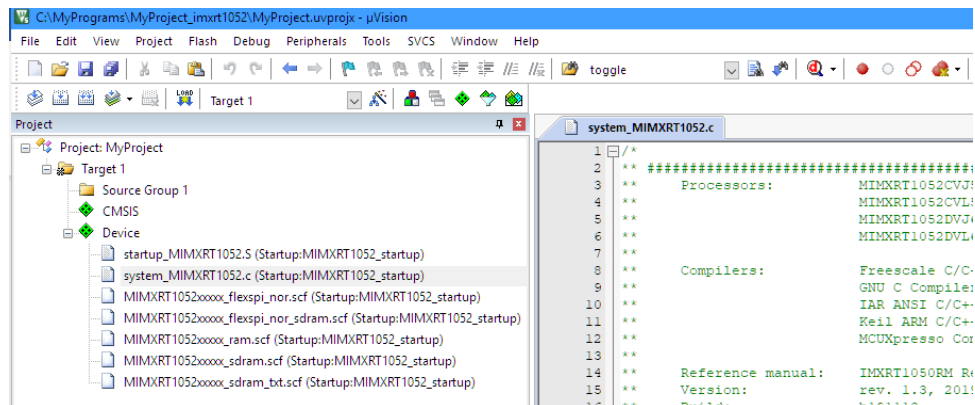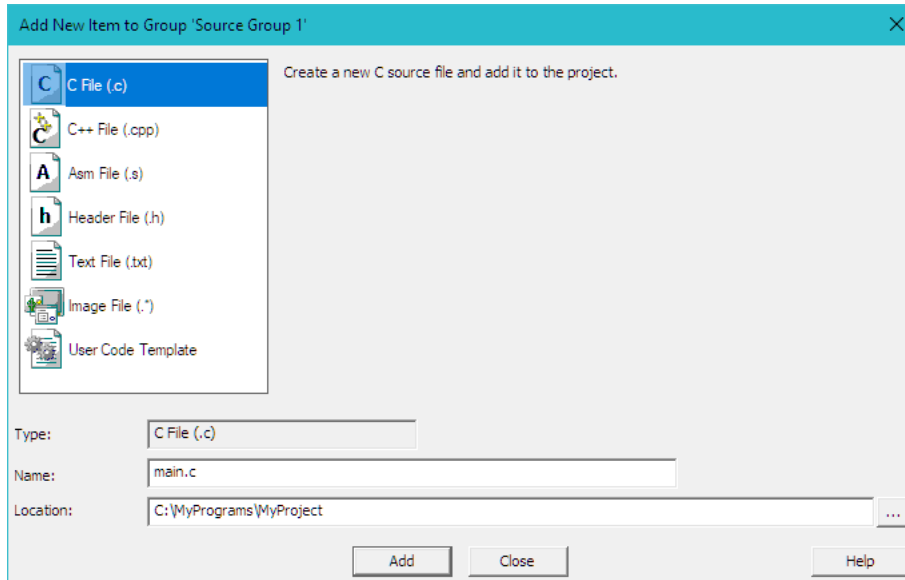

Add main.c Source Code File

Now we can add the main.c file with initial program code.

☞ In the **Project** window, right-click **Source Group 1** and open the dialog **Add New Item to Group**.

☞ Click on **C File (.c)** specify the file name, in our case main.c and click **Add**.

This creates the file *main.c* in the project group **Source Group 1**. Add following
content to the file:

```c
/* -------------------------------------------------------------------
 *   main.c file
 * ------------------------------------------------------------------*/

#include "RTE_Components.h"         // Component selection
#include CMSIS_device_header        // Device header

uint32_t volatile msTicks;          // Counter for millisecond Interval

void SysTick_Handler (void) {       // SysTick Interrupt Handler
  msTicks++;                        // Increment Counter
}

void WaitForTick (void) {
  uint32_t curTicks;
  curTicks = msTicks;               // Save Current SysTick Value
  while (msTicks == curTicks) {     // Wait for next SysTick Interrupt
    __WFE ();                       // Power-Down until next Event
  }
}

// Configure & Initialize the MCU
void Device_Initialization (void) {

  SystemInit();                     // Device initialization
  SystemCoreClockUpdate();          // Clock setup

  if (SysTick_Config (SystemCoreClock / 1000)) {    // SysTick 1ms
    ; // Handle Error
  }
}
```

```
// The processor clock is initialized by CMSIS startup + system file
int main (void) {                    // User application starts here
  Device_Initialization ();          // Configure & Initialize MCU

  while (1) {                        // Endless Loop (the Super-Loop)
    __disable_irq ();                // Disable all interrupts
    // Get_InputValues ();
    __enable_irq ();                 // Enable all interrupts
    // Process_Values ();
    WaitForTick ();                  // Synchronize to SysTick Timer
  }
}
```

For many devices the build process described in step **Build the Application Project** will succeed already after this this step.

In some cases (and in our example for MIMXRT1052) additional changes in the project configurations are required as explained in **Configure Project Options** section below.

Device Initialization

System initialization in our simple example is done in the *Device_initialization()* function using only CMSIS-Core API.

Silicon vendors provide the device-specific file **system_<device>.c** (in our case *system_MIMXRT2052.c*) that implements *SystemInit* and *SystemCoreClockUpdate* functions. This file gets automatically added to the project with the selection of **::Device::Startup** component in the **Manage Run-Time Environment** in the previous step.

Real-world examples often require complex configuration for pins and peripherals with a significant part of the system setup relying on the device hardware abstraction layer (HAL) provided by the vendor.

Section **Device Configuration Variations** explains additional details and provides examples on device configuration using external tools.

Configure Project Options

For some devices new projects cannot be built and programmed onto the device with default settings and require special configuration options. This is often a reason why starting with a ready-to-use example can be beneficial.

Click **Options for Target...** button on the toolbar to access the configuration options.

It contains multiple tabs that provide configuration options for corresponding
functionality.
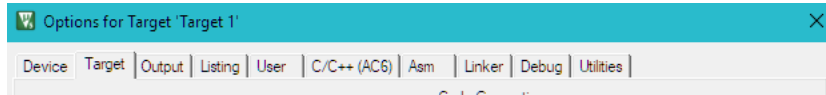
Changes required for getting started depend on the target device and software
components used in the project. Subsections below explain the modifications
required in the specified dialog tabs for the MIMXRT1052 used in our example.

C/C++ (AC6) dialog

To exclude mostly just informative warnings generated by the Arm Compiler 6
select *AC5-like Warnings* in the **Warnings** field of the **C/C++ (AC6)** tab.

Linker dialog

Complex devices or programs may require use of a scatter file to specify memory
layout. The figure below highlights the changes required in our example:
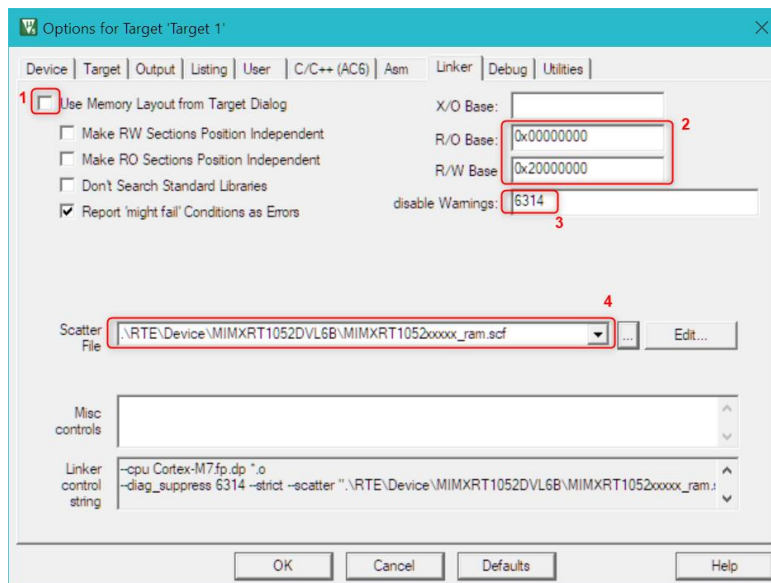
1. Unchecking the flag **Use Memory Layout from Target Dialog** enables use
    of custom scatter file provided in the item 4 below.

2. **R/O** and **R/W Bases** define the start addresses for read only (code and
    constants) and read-write areas respectively.

3. Disable warning **#6314** for unused memory objects.

4. The Device Family Pack (DFP) contains some preconfigured scatter files that are copied into the new project. To simplify project configuration, we will execute the program from the on-chip RAM and hence choose in the drop-down menu for the **Scatter file** the
.\RTE\Device\MIMXRT1052DVL6B\MIMXRT1052xxxxx_ram.scf.

Debug dialog

To ensure that the program loads to RAM and we can debug it, following changes are required in the Debug tab.
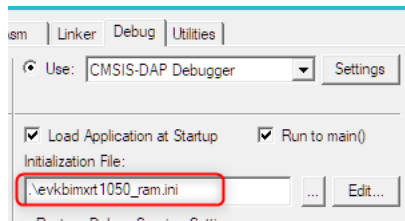
☞ In the project folder create a new file that will be used to initialize the debug session (in our case - **evkbimxrt1050_ram.ini**) and provide path to it in the **Initialization File** field.

For this example, add the following content to the file:

```
/*------------------------------------------------------------------
 * evkbimxrt1050 ram.ini file
 *----------------------------------------------------------------*/

FUNC void Setup (void) {
  SP = _RDWORD(0x00000000);          // Setup Stack Pointer
  PC = _RDWORD(0x00000004);          // Setup Program Counter
  _WDWORD(0xE000ED08, 0x00000000);   // Setup VTOR
}

FUNC void OnResetExec (void)  {      // executes upon software RESET
  Setup();                           // Setup for Running
}

LOAD %L INCREMENTAL                  // Download

Setup();                             // Setup for Running

// g, main
```
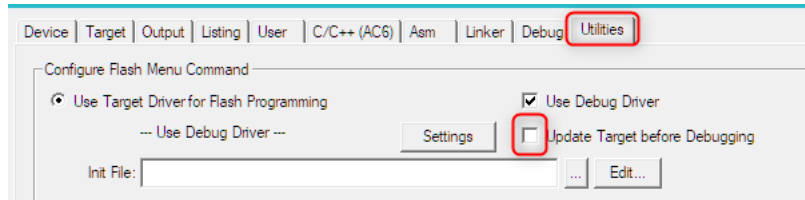
Utilities dialog

☞ In the **Utilities** dialog, uncheck the option **Update Target before Debugging** to ensure that the debugger doesn't try to load program to Flash.

Build the Application Project

Use **Rebuild** toolbar button to build the application, which compiles and links all related source files. **Build Output** shows information about the build process.

An error-free build displays program size information, zero errors, and zero warnings.

The section **Debug Applications** guides you through the steps to connect your evaluation board to the PC and download the application to the target hardware.

Project with CMSIS-RTOS2

The section shows how to setup a simple project based on CMSIS-RTOS2. The project uses device HAL to control on-board LED.

To avoid making project configuration and device initialization from scratch we take an existing blinky example in infinite-loop design delivered with the DFP and modify it to operate based on CMSIS-RTOS2 API. Following steps are required:

1. **Copy an Example**: copy an existing example and verify that it works

2. **Add CMSIS-RTO2 Component**: add CMSIS-RTOS2 API and RTX5 kernel to the

3. **Add RTOS Initialization**: add *main.c* file that initializes the device and RTOS.

4. **Configure Keil RTX5 RTOS**: modify the RTOS settings according to the application needs.

5. **Implement User Threads**: implement user code.

6. **Build and Run the program**: the step is same as explained in the previous section.

In our case we will use a simple **iled_blinky** example for IMXRT1050-EVK board.

Copy an Example
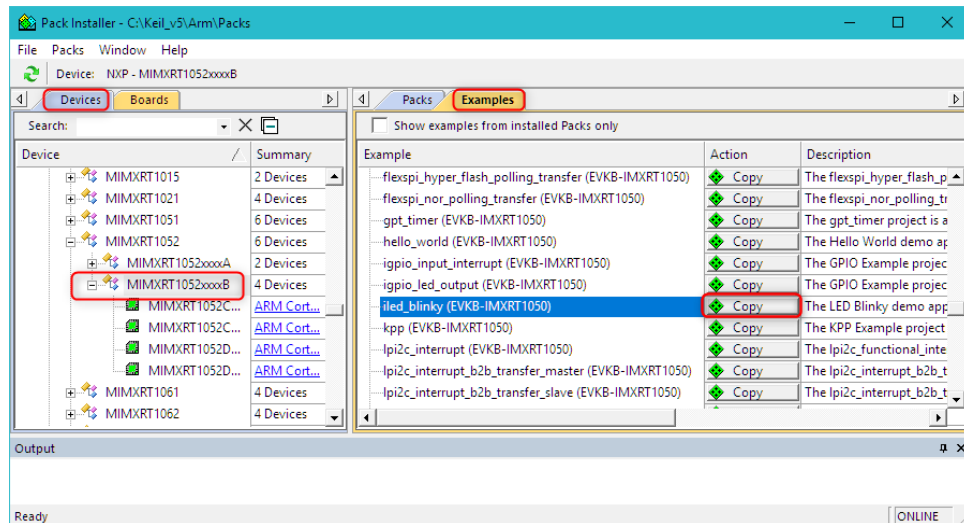
Section **Verify Installation using Example Projects** explains the steps needed to copy, build and run an example project. In our example we use target *iled_blinky debug* that executes the program from on-chip RAM.

☞ To build the project with the *iled_blinky debug* target, the SPI flash related file *fsl_flexspi_nor_boot.c* has to be excluded from the build.
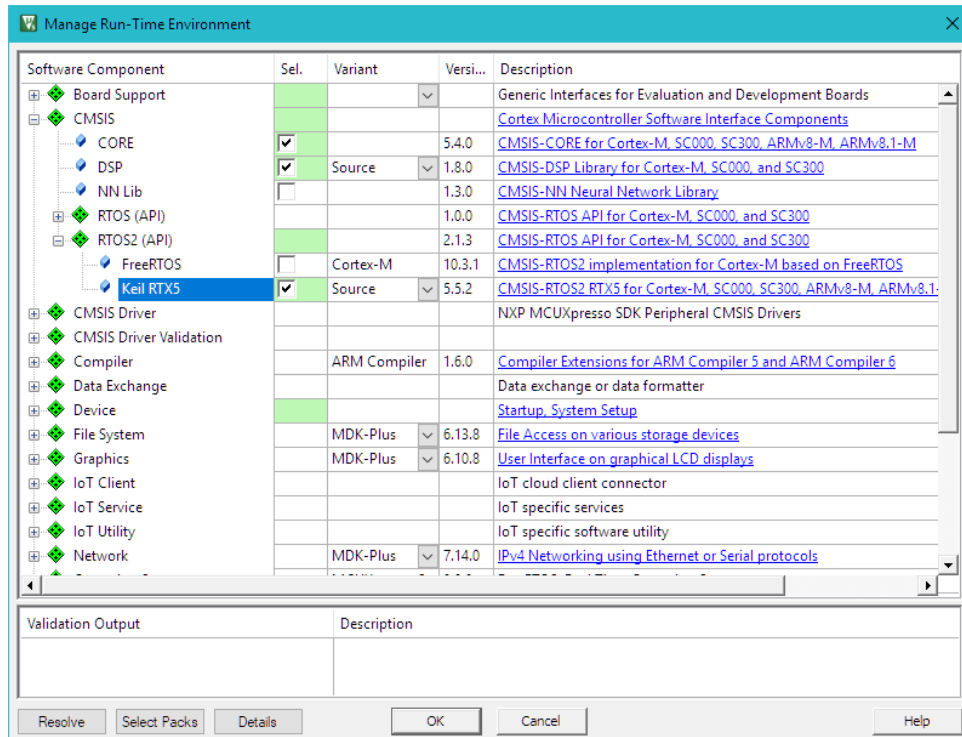
Find this file in the **Project** window under **Device** component, right-click on it, then select **Options for Component Class 'Device'** and in the **Properties** tab uncheck **Include in target build**. Press **OK**. The file will be marked with a corresponding symbol 🔧
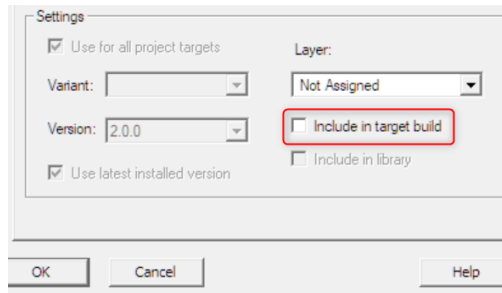
Add CMSIS-RTO2 Component

Next, add the RTOS software component:

☞ Expand **::CMSIS::RTOS2 (API)** and enable **Keil RTX5**. In the **Variant** column select **Source** to have the RTOS added to the project as a source code that also supports detailed debugging using Event Recorder. For reduced code size, use the **Library** variant instead. Press **OK**.

Keil RTX5 code appears in the **Project** window under **CMSIS** component.

☞ In our case for MIMRT1052 we need to change the **Assembler Option** so that Keil RTX5 file **irq4_cm4f.s** can be assembled correctly.

For that go to the **Options for Target.. – Asm** tab and in the dropdown menu **Assembler Option** select **armclang (Auto Select)** instead of **armclang (GNU Syntax)** configured by default in the original example. Press **OK**.
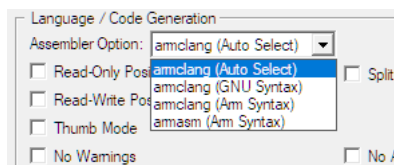

Alternatively, the assembler option can be specified for the **irq4_cm4f.s** file only. For that find this file in the **Project** window under **CMSIS** component, right-click on it, then select **Options for Component Class 'CMSIS'** and in the **Asm** tab choose *armclang (Arm Syntax)* from the drop-down menu in **Assembler Option** field. Press **OK**.

Add RTOS Initialization

Add template application code using pre-configured **User Code Templates** containing routines that resemble the functionality of the software component.

☞ In the **Project** window, right-click in the group with the source code (in our case **source** and open the dialog **Add New Item to Group**.

☞ Click on **User Code Template** to list available code templates for the software components included in the project. Select **CMSIS-RTOS2 'main' function**, verify the file name, and click **Add**.
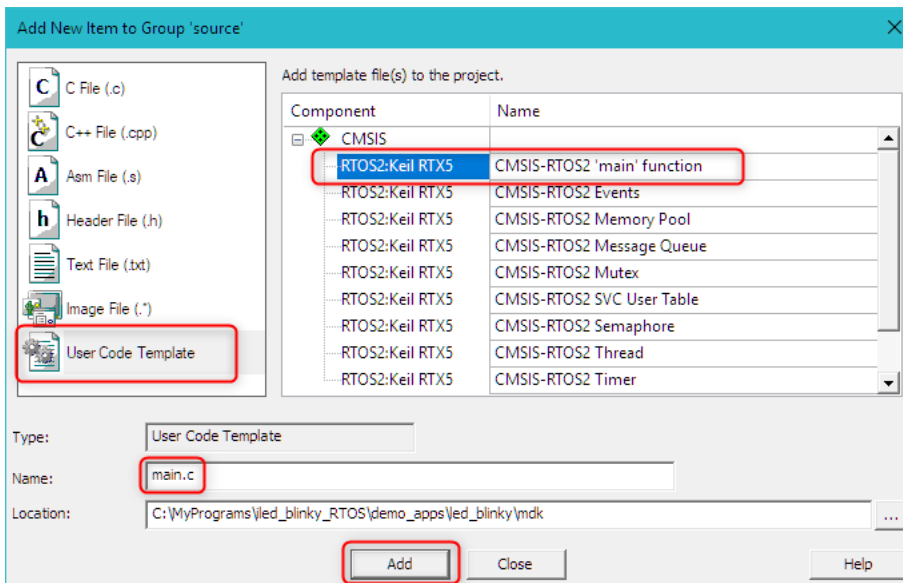
This adds the file *main.c* to the project group **source**. The file contains the necessary functions for minimal CMSIS-RTOS application.

We reuse the device initialization functions from the original *main()* function. We remove the implementation of *app_main* function as it will be placed in the other file. As a result, the *main.c* file contains following code:

```c
/*----------------------------------------------------------------
 * CMSIS-RTOS 'main' function template
 *----------------------------------------------------------------*/
#include "RTE_Components.h"
#include  CMSIS_device_header
#include "cmsis_os2.h"
#include "board.h"
#include "pin_mux.h"

extern void app_main (void *argument); // application main thread
```

```
int main (void) {

  /* Board pin init */
  BOARD_InitPins();
  BOARD_InitBootClocks();

  // System Initialization
  SystemCoreClockUpdate();
  // ...

  osKernelInitialize();                 // Initialize CMSIS-RTOS
  osThreadNew(app_main, NULL, NULL);    // Create application main thread
  osKernelStart();                      // Start thread execution
  for (;;) {}
}
```

Note the *Board_InitPins()* and *Board_InitBootClocks()* functions that configure the underlying MIMXRT1052 device. Section **Example: MCUXpresso Config Tools** explains device configuration in more details.

Configure Keil RTX5 RTOS

☞ In Project window - CMSIS group open *RTX_Config.h* file and configure according to the project requirements as explained in **Keil RTX5 Configuration**. In our example we can keep default settings.

Implement User Threads

The file *led_blinky.c*, containing the initial *main()* function, can now be rewritten using RTOS threads. We implement two user threads: *thrLED* toggling the LED and *thrSGN* acting as a signal thread that triggers *thrLED* thread with regular delays.

```
/*------------------------------------------------------------------
 * led_blinky.c file
 *----------------------------------------------------------------*/
#include "cmsis_os2.h"
#include "fsl_gpio.h"
#include "pin_mux.h"
#include "board.h"

static osThreadId_t tid_thrLED; // Thread id of thread: LED
static osThreadId_t tid_thrSGN; // Thread id of thread: SGN
/*------------------------------------------------------------------
  thrLED: blink LED
 *----------------------------------------------------------------*/
__NO_RETURN static void thrLED(void *argument) {
  (void)argument;
  uint32_t active_flag = 1U;

  for (;;) {
```

```
    osThreadFlagsWait(1U, osFlagsWaitAny, osWaitForever);
    GPIO_PinWrite(BOARD_USER_LED_GPIO, BOARD_USER_LED_PIN, active_flag);
    active_flag=!active_flag;
  }
}
/*-----------------------------------------------------------------
  thrSGN: Signal LED to change
 *----------------------------------------------------------------*/
__NO_RETURN static void thrSGN(void *argument) {
  (void)argument;
  uint32_t last;

  for (;;) {
    osDelay(500U);                     // Run delay for 500 ticks
    osThreadFlagsSet(tid_thrLED, 1U); // Set flag to thrLED
  }
}
/*-----------------------------------------------------------------
 * Application main thread
 *----------------------------------------------------------------*/
void app_main(void *argument) {
  (void)argument;

  tid_thrLED = osThreadNew(thrLED, NULL, NULL); // Create LED thread
  if (tid_thrLED == NULL) { /* add error handling */  }

  tid_thrSGN = osThreadNew(thrSGN, NULL, NULL); // Create SGN thread
  if (tid_thrSGN == NULL) { /* add error handling */  }

  osThreadExit();
}
```

Device Configuration Variations

CMSIS-CORE defines methods for device startup such as *SystemInit()* and *SystemClock_Config()* but the actual implementation details vary between different vendors.

Some devices perform a significant part of the system setup as part of the device hardware abstraction layer (HAL). In many cases the HAL components for the target platform are delivered as part of the Device Family Pack (DFP) and are available for selection in the **Manage Run-Time Environment** dialog, typically under **::Device** component.

Device vendors frequently provide a software framework that allows device configuration with external utilities.

In the following section, device startup variations are exemplified.

Example: STM32Cube

Many STM32 devices are using the **STM32Cube framework** that can be configured with a classical method using the *RTE_Device.h* configuration file or by using **STM32CubeMX** tool.

The classic **STM32Cube Framework** component provides a specific user code template that implements the system setup. Using **STM32CubeMX**, the *main.c* file and other source files required for startup are copied into the project below the **STM32CubeMX:Common Sources** group.

Setup the Project using the Classic Framework

This example creates a project for the STM32F746G-Discovery kit using the classical method. In the **Manage Run-Time Environment** window, select the following:

☞ Expand **::Device:STM32Cube Framework (API)** and enable **:Classic**.

Expand **::Device** and enable **:Startup**.
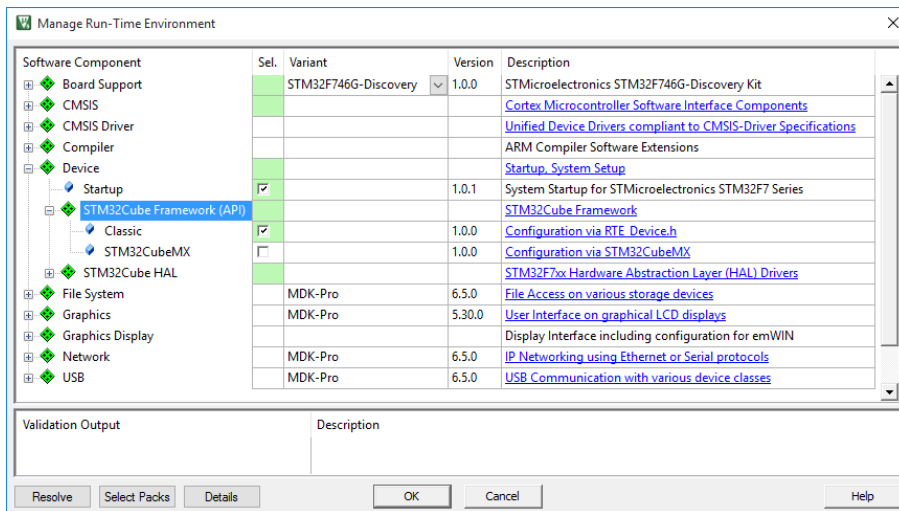
☞ Click **Resolve** to enable other required software components and then **OK**.

☞ In the **Project** window, right-click **Source Group 1** and open the dialog **Add New Item to Group**.
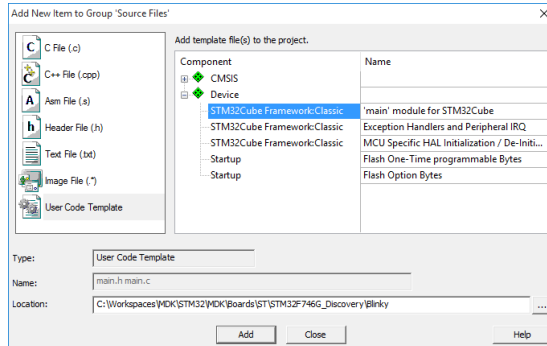
☞ Click on **User Code Template** to list available code templates for the software components included in the project. Select **'main' module for STM32Cube** and click **Add**.

The *main.c* file contains the function *SystemClock_Config()*. Here, you need to make the settings for the clock setup:

Code for *main.c*

```
:
static void SystemClock_Config (void) {
  RCC_ClkInitTypeDef RCC_ClkInitStruct;
  RCC_OscInitTypeDef RCC_OscInitStruct;

  /* Enable HSE Oscillator and activate PLL with HSE as source */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_ON;
  RCC_OscInitStruct.HSIState = RCC_HSI_OFF;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLM = 25;
  RCC_OscInitStruct.PLL.PLLN = 432;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 9;
  HAL_RCC_OscConfig(&RCC_OscInitStruct);

  /* Activate the OverDrive to reach the 216 MHz Frequency */
  HAL_PWREx_EnableOverDrive();

  /* Select PLL as system clock source and configure the HCLK, PCLK1 and
PCLK2 clocks dividers */
  RCC_ClkInitStruct.ClockType = (RCC_CLOCKTYPE_SYSCLK | RCC_CLOCKTYPE_HCLK
| RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2);
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;
  HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_7);
}
:
```
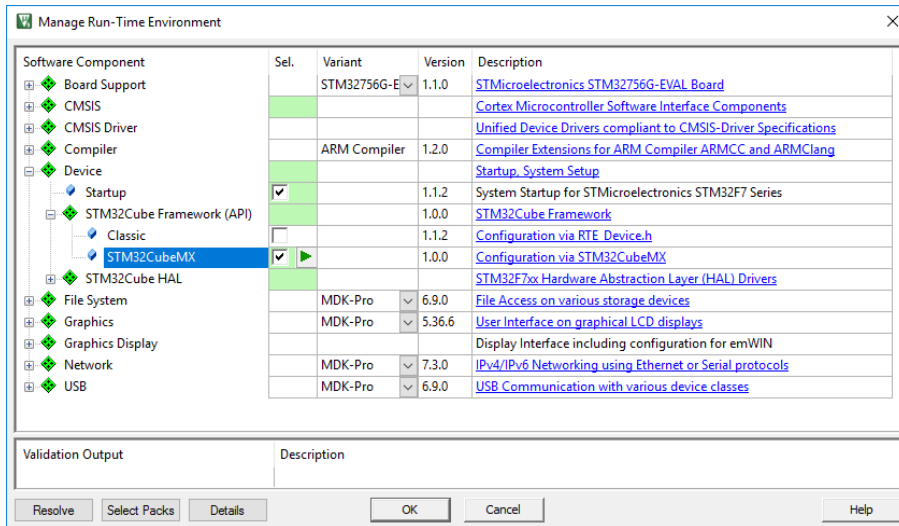
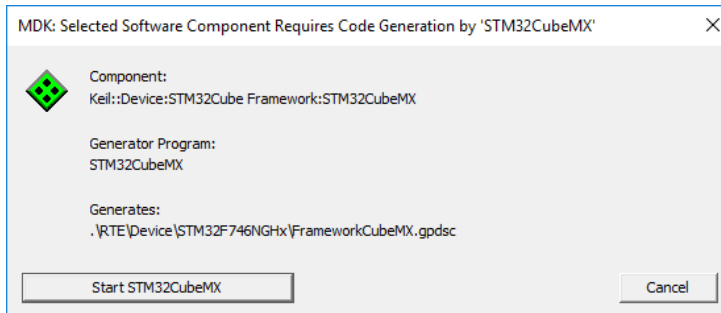Now, you can start to write your application code using this template.

Setup the Project using STM32CubeMX

This example creates the same project as before using **STM32CubeMX**. In the **Manage Run-Time Environment** window, select the following:
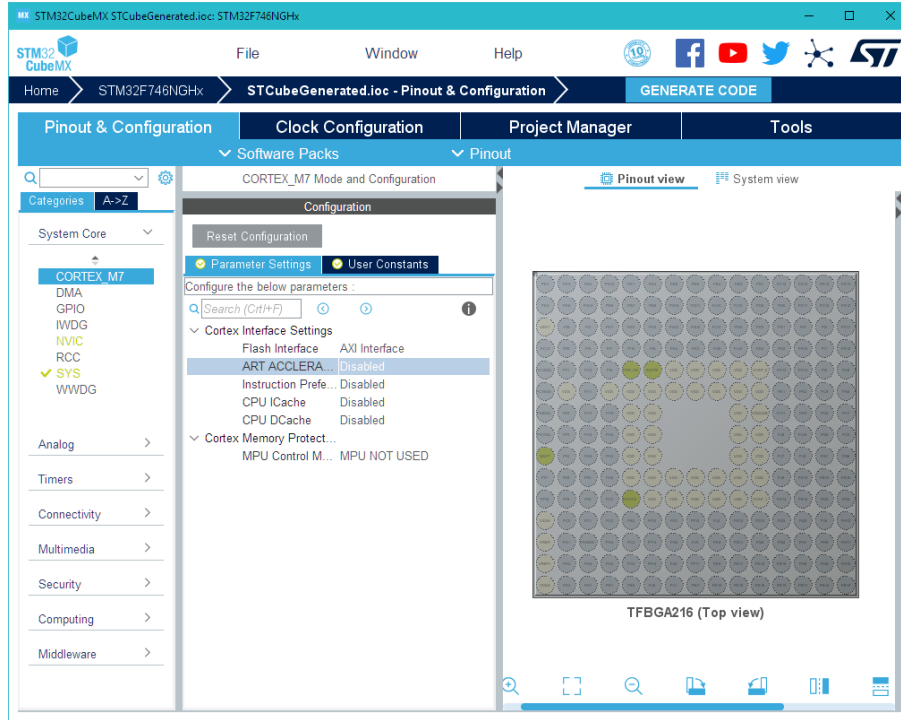
☞ Expand **::Device:STM32Cube Framework (API)** and enable **:STM32CubeMX**. Expand **::Device** and enable **:Startup**.

Software Component	Sel.	Variant	Version	Description
⊞ ◆ Board Support		STM32756G-E ∨	1.1.0	STMicroelectronics STM32756G-EVAL Board
⊞ ◆ CMSIS				Cortex Microcontroller Software Interface Components
⊞ ◆ CMSIS Driver				Unified Device Drivers compliant to CMSIS-Driver Specifications
⊞ ◆ Compiler		ARM Compiler	1.2.0	Compiler Extensions for ARM Compiler ARMCC and ARMClang
⊟ ◆ Device				Startup, System Setup
◆ Startup	☑		1.1.2	System Startup for STMicroelectronics STM32F7 Series
⊟ ◆ STM32Cube Framework (API)			1.0.0	STM32Cube Framework
◆ Classic	☐		1.1.2	Configuration via RTE_Device.h
◆ STM32CubeMX	☑ ▶		1.0.0	Configuration via STM32CubeMX
⊞ ◆ STM32Cube HAL				STM32F7xx Hardware Abstraction Layer (HAL) Drivers
⊞ ◆ File System		MDK-Pro ∨	6.9.0	File Access on various storage devices
⊞ ◆ Graphics		MDK-Pro ∨	5.36.6	User Interface on graphical LCD displays
⊞ ◆ Graphics Display				Display Interface including configuration for emWIN
⊞ ◆ Network		MDK-Pro ∨	7.3.0	IPv4/IPv6 Networking using Ethernet or Serial protocols
⊞ ◆ USB		MDK-Pro ∨	6.9.0	USB Communication with various device classes

Validation Output | Description

[Resolve] [Select Packs] [Details]   [OK] [Cancel]   [Help]

☞ Click **Resolve** to enable other required software components and then **OK**. A new window will ask you to start **STM32CubeMX**.

Component:
Keil::Device:STM32Cube Framework:STM32CubeMX

Generator Program:
STM32CubeMX

Generates:
.\RTE\Device\STM32F746NGHx\FrameworkCubeMX.gpdsc

[Start STM32CubeMX]   [Cancel]

STM32CubeMX is started with the correct device selected:

☞ Configure your device as required. When done, go to **Project ➔ Generate Code** to create a GPDSC file. µVision will notify you:
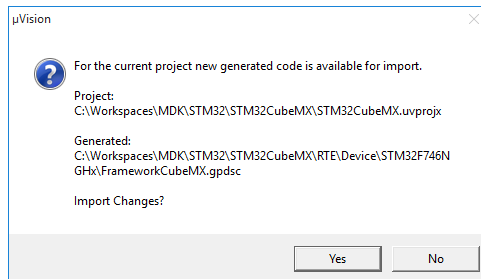
☞ Click **Yes** to import the project. The *main.c* and other generated files are added to a folder called **STM32CubeMX:Common Sources**.

Read more about device setup for a µVision project using STM32CubeMX in dedicated documentation
keil.com/pack/doc/STM32Cube/General/html/index.html.

Example: MCUXpresso Config Tools

For configuring most of its Kinetis, LPC and iMX RT devices NXP provides **MCUXpresso Config Tools**.

Enable Project for Configuration

To configure an MDK project for MCUXpresso Config Tools it has to contain special components in the Board Support and Device groups. This is already the case for many example projects available via the Pack Installer but needs to be ensured for older projects or when creating a project from scratch.

☞ Expand **::Board Support::SDK Project Template::** and enable **:project_template**. From the drop-down menu in Variant column choose either an option for target MCU or if available target board (**evkbimxrt1050** in our case). Multiple dependencies may be highlighted in yellow as required.
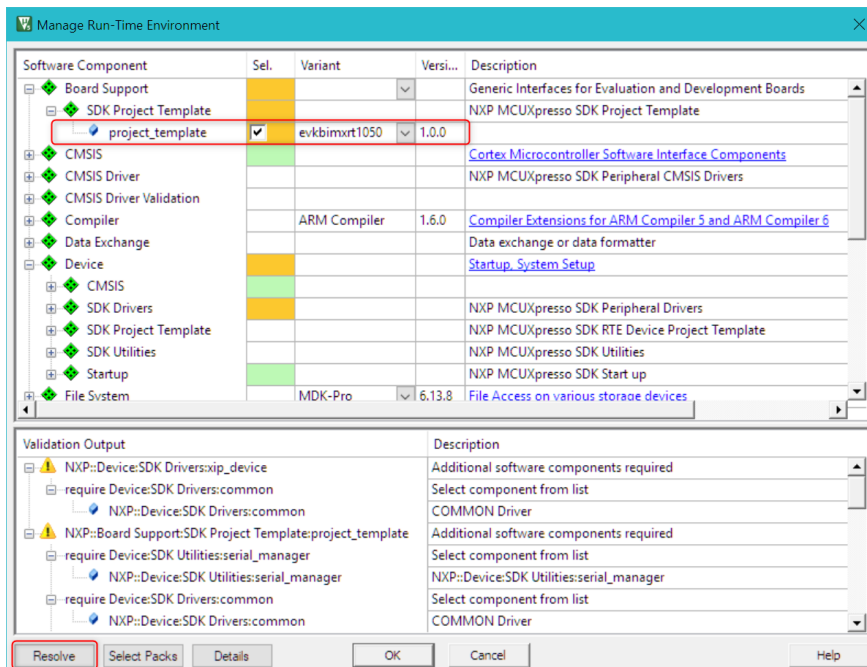
☞ Click **Resolve** to enable the required software components and then **OK**.
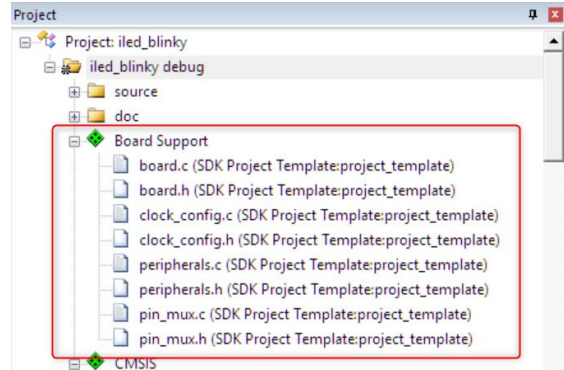
Configure the Device

When the project contains the components explained in the subsection above
MCUXpresso Config Tools can be used to create the device initialization code.

☞ Start the MCUXpresso Config Tools. **Create a new configuration dialog**
opens. The dialog can be also open from **File – New…** menu.

☞ Select option **Create a new configuration based on an existing
IDE/toolchain project** and specify the path to the µVision project. In our
case we take an example explained in section **Project with CMSIS-
RTOS2**.

☞ Press **Finish**.
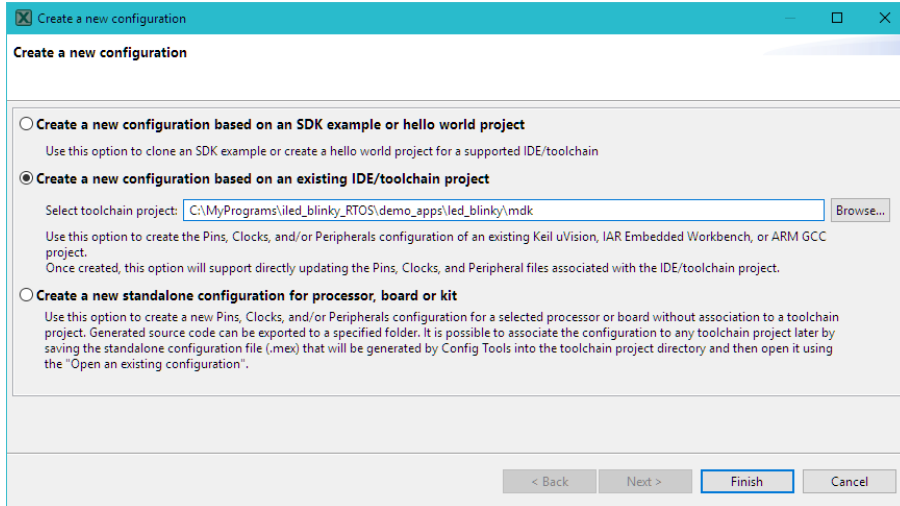
Wait until **Config Tool Overview** window opens.

☞ Use available graphical tools to configure device clocks, pins, peripherals, and DCD as required.
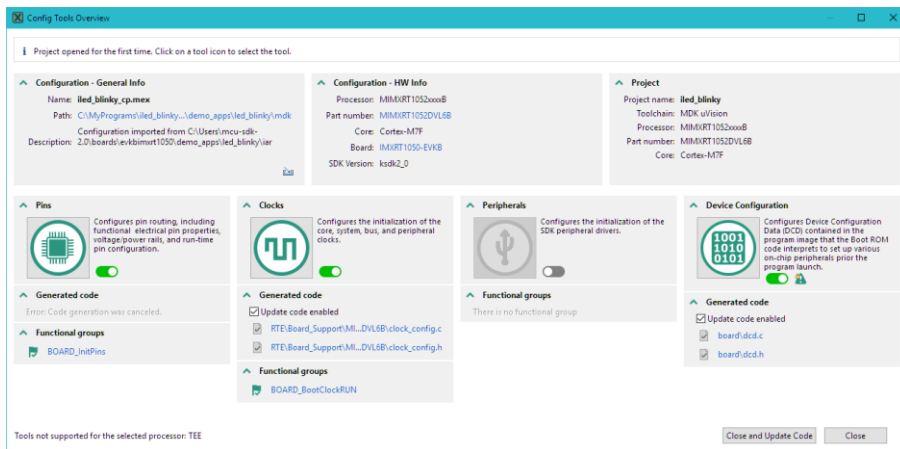
In our example we add a GPIO5 initialization for the user button available on the board:
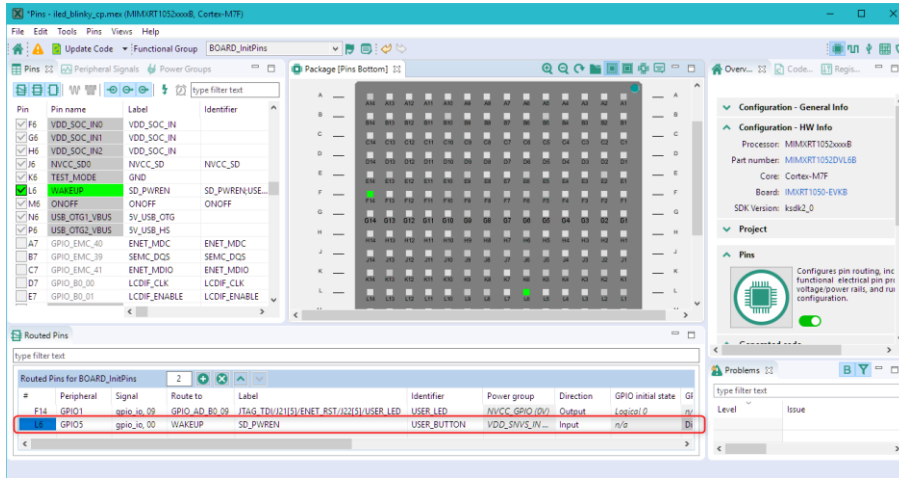
☞ Press **Update Code** button. Review the changes to be applied and press **OK**.
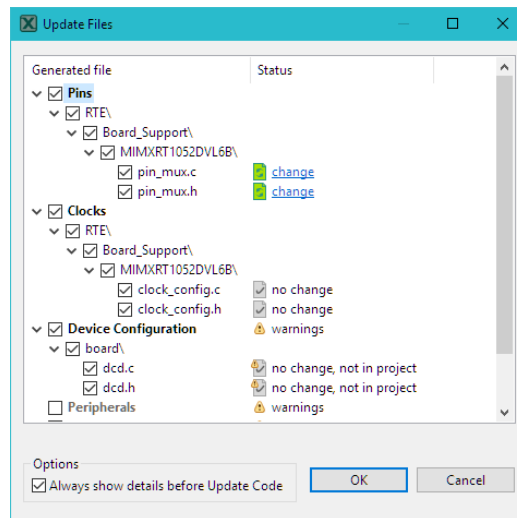
This updates the necessary files in the **Board Support** group present in the project.

Update Application Code the Device

☞ Update application code according to the new device configuration.

This may require including some header files, calling additional initialization functions in *main()* and of course implementing application logic itself.

In our example we just update the *thrSGN* thread in *led_blinky.c* file so that the signal for togging the LED is postponed as long as the user button is pressed:

```
/*----------------------------------------------------------------
  thrSGN: Signal LED to change
 *----------------------------------------------------------------*/
__NO_RETURN static void thrSGN(void *argument) {
  (void)argument;
  uint32_t last;

  for (;;) {
    osDelay(500U);             // Run delay for 500 ticks
    while (!GPIO_PinRead (BOARD_USER_BUTTON_GPIO,
           BOARD_USER_BUTTON_GPIO_PIN)){
      osDelay(10);          // Delay further while SW8 button is pressed
    }
    osThreadFlagsSet(tid_thrLED, 1U); // Set flag to thrLED
  }
}
```

Secure/non-secure programming

Embedded system programmers face demanding product requirements that include cost sensitive hardware, deterministic real time behavior, low-power operation, and secure asset protection.

Modern applications have a strong need for security. Assets that may require protection are:

- device communication (using cryptography and authentication methods)
- secret data (such as keys and personal information)
- firmware (against IP theft and reverse engineering)
- operation (to maintain service and revenue)

The TrustZone® for Armv8-M security extension is a System on Chip (SoC) and CPU system-wide approach to security and is optimized for ultra-low power embedded applications. It enables multiple software security domains that restrict access to secure memory and I/O to trusted software only.

TrustZone for Armv8-M architecture (Cortex-M23/M33/M35P/M55 cores):

- preserves low interrupt latencies for both secure and non-secure domains.

- does not impose code or cycle overhead.

- introduces efficient instructions for calls to the secure domain.

Create Armv8-M software projects

The steps to create a new software project for an Armv8-M core (Cortex-M23/M33/M35P/M55) in MDK are:

- Define the overall system and memory configuration. This has impact on:

  o Setup secure and non-secure projects

  o Add startup code and 'main' module to secure and non-secure projects.

  o Reflect this configuration in the CMSIS-Core file partition_<device>.h

- Define the API of the secure software part in a header file to allow usage from the non-secure part

- Create the application software for the secure and the non-secure part
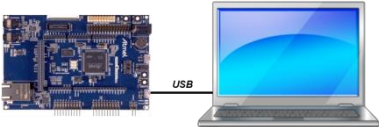
Application note 291 describes the necessary steps in detail and contains example projects and best practices for secure and non-secure programming using Armv8-M targets. It is available at **keil.com/appnotes/docs/apnt_291.asp**

Debug Applications

The Arm CoreSight™ technology integrated into the Arm Cortex-M processor-based devices provides powerful debug and trace capabilities. It enables run-control to start and stop programs, breakpoints, memory access, and Flash programming. Features like sampling, data trace, exceptions including program counter (PC) interrupts, and instrumentation trace are available in most devices. Devices offer instruction trace using Embedded Trace Macrocell (ETM), Embedded Trace Buffer (ETB), or Micro Trace Buffer (MTB) to enable analysis of the program execution. Refer to **keil.com/coresight** for a complete overview of the debug and trace capabilities.

Debugger Connection

MDK contains the µVision Debugger that connects to various debug/trace adapters and allows you to program the Flash memory. It supports traditional features like simple and complex breakpoints, watch windows, and execution control. Using trace, additional features like event/exception viewers, logic analyzer, execution profiler, and code coverage are supported.

- The ULINK*plus* and ULINK2 debug adapters interface to JTAG/SWD debug connectors and support trace with the Serial Wire Output (SWO). The ULINK*pro* debug/trace adapter also interfaces to ETM trace connectors and uses streaming trace technology to capture the complete instruction trace for code coverage and execution profiling. Refer to **keil.com/ulink** for more information.

- CMSIS-DAP based USB JTAG/SWD debug interfaces are typically part of an evaluation board or starter kit and offer integrated debug features. MDK also supports several proprietary interfaces that offer a similar technology.

- Third-party debug solutions, such as Segger J-Link or J-Trace are supported in MDK. Some starter kit boards provide the J-Link Lite technology as an on-board solution.

Using the Debugger

As an example, we will debug the *Blinky* application created in the previous chapter on hardware. You need to configure the debug connection.

Select the debug adapter and configure debug options.

From the toolbar, choose **Options for Target**, click the **Debug** tab, enable **Use**, and select the applicable debug driver.
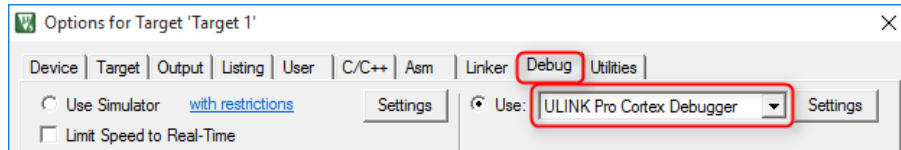

The device selection already configures the Flash programming algorithm for on-chip memory. Verify the configuration using the **Settings** button.

In our example we run the program out of RAM. But in cases when flash memory is used, the program needs to be loaded into the Flash.

From the toolbar, choose **Download.** The **Build Output** window shows messages about the download progress.
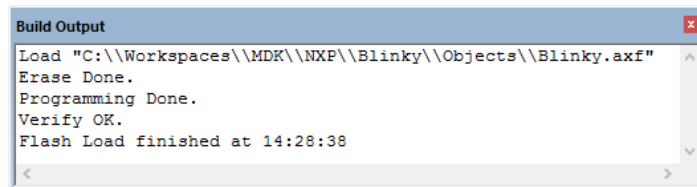
Start debugging on hardware. From the toolbar, select **Start/Stop Debug Session**.
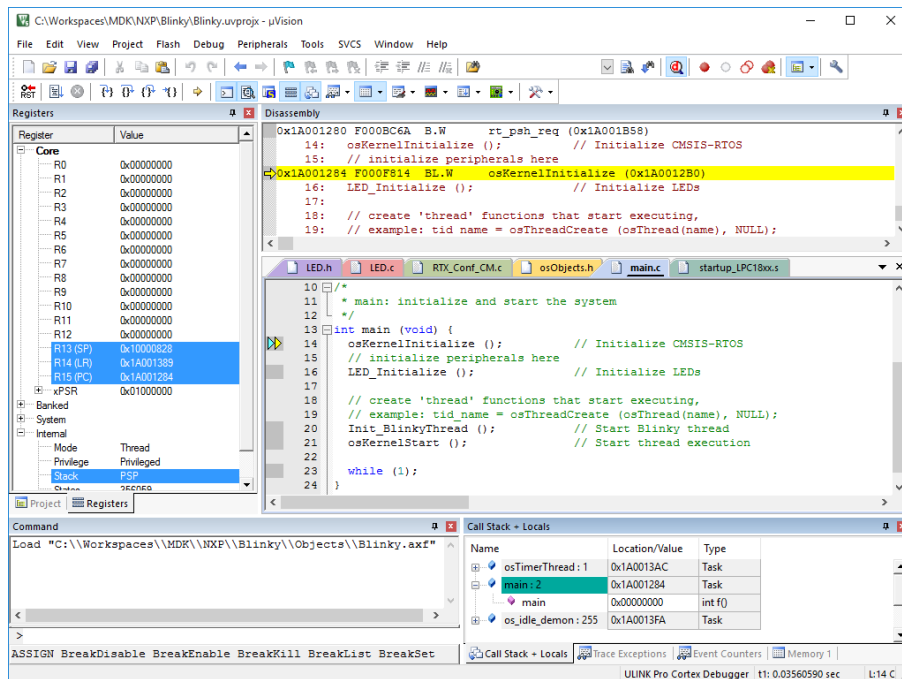
During the start of a debugging session, µVision loads the application, executes the startup code, and stops at the main C function.

Click **Run** on the toolbar. The LED flashes with a frequency of one second.

Debug Toolbar

The debug toolbar provides quick access to many debugging commands such as:

Step steps through the program and into function calls.

Step Over steps through the program and over function calls.

Step Out steps out of the current function.

Stop halts program execution.

Reset performs a CPU reset.

Show to the next statement to be executed (current PC location).

Command Window

You may also enter debug commands in the **Command** window.

```
Command                                                                          ⊠
BS \\Blinky\main.c\32                                                             ▲
BS \\Blinky\main.c\23
BS Write msTicks==100, 1, "printf(\"Write Access Breakpoint: 100 ticks reached\\n\");"
WS 1, `msTicks,0x0A
WS 1, `CORE_CLK/1000000,0x0A
WS 1, ((SysTick_Type *) ((0xE000E000UL) + 0x0010UL) ),0x0A
WS 1, `SystemCoreClock,0x0A
Write Access Breakpoint: 100 ticks reached
Write Access Breakpoint: 100 ticks reached      Comamnd Line     Dynamic Command List
Write Access Breakpoint: 100 ticks reached
◄                                       III                                    ►
  >
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet BreakAccess COVERAGE DEFINE
```

On the **Command Line** enter debug commands or press **F1** to access detailed
help information.

Disassembly Window

The **Disassembly**
window shows the
program execution in
assembly code
intermixed with the
source code (when
available). When this is
the active window, then
all debug stepping
commands work at the
assembly level.

The window margin
shows markers for

```
Disassembly
     21: void Delay (void)   {
○0x08000284 4770      BX              lr
     22:         while (msTicks < 499);
0x08000286 BF00       NOP
●0x08000288 480E       LDR             r0,[pc,#56]   ; @0x080002C4
0x0800028A 6800       LDR             r0,[r0,#0x00]
0x0800028C F5B07FF9   CMP             r0,#0x1F2
⇨0x08000290 DDFA       BLE             0x08000288
     23:         msTicks = 0;
0x08000292 2000       MOVS            r0,#0x00
◄
```

```
  LED.c      main.c      LED.h
1  /*-----------------------------------------------
2    * CMSIS-RTOS 'main' function template
3    *-----------------------------------------------
4   #include "LED.h"
5   #include "stm32f4xx.h"
```

breakpoints, bookmarks, and for the next execution statement.

Component Viewer

The **Component Viewer** shows information about:

- Software components that are provided in static memory variables or structures.
- Objects that are addressed by an object handle.

Component Viewer windows containing objects are listed in the menu **View – Watch Windows**.

The picture below is an example showing static component information for a USB HID example project:

USB Device and Host	
Property	Value
Library Version	6.9.6
Device 0	
Vendor ID	0xC251
Product ID	0x2501
Speed	Low/Full/High Speed
Endpoint 0 Maximum Packet Size	64
Number of Interfaces	1
Assigned Address	10
Configuration Status	Configured
Endpoint Activity	
Human Interface Device 0	In reports 1, Out reports 1, EP INT IN: 1, EP INT OUT: 1
Device 1	
Vendor ID	0xC251
Product ID	0x2511
Speed	Low/Full Speed
Endpoint 0 Maximum Packet Size	8
Number of Interfaces	1
Assigned Address	0
Configuration Status	Unconfigured
Endpoint Activity	
Human Interface Device 1	In reports 1, Out reports 1, EP INT IN: 1, EP INT OUT: 1

For more information refer to
keil.com/pack/doc/compiler/EventRecorder/html/cv_use.html.

Event Recorder

The **Event Recorder** shows execution status and event information and helps to analyze the operation of software components. MDK-Middleware and the Keil RTX5 already offer the required description files.

The **Event Recorder**:

- increases the visibility to the dynamic execution of an application program.
- provides filter capabilities for the different event types.
- allows unrestricted calls to event recorder functions from threads, RTOS kernel, and ISRs.
- implements recording functions that do not disable ISR on Armv7-M.
- supplies fast time-deterministic execution of event recorder functions with minimal code and timing overhead. Thus, event annotations can remain in production code without the need to create a debug or release build.

To add the Event Recorder to the example from section **Project with CMSIS-RTOS2** on page 52, do the following:

- In the **Manage Run-Time Environment** window, select the component **Compiler:Event Recorder** and also verify that the component **CMSIS:RTOS2 (API):Keil RTX5** is selected in **Source** variant. Press **OK**.
- In the **Project** window under **CMSIS** component open *RTX_Config.h* file, switch to the **Configuration Wizard** view, expand **Event Recorder Configuration** group and enable **Global Initialization**.
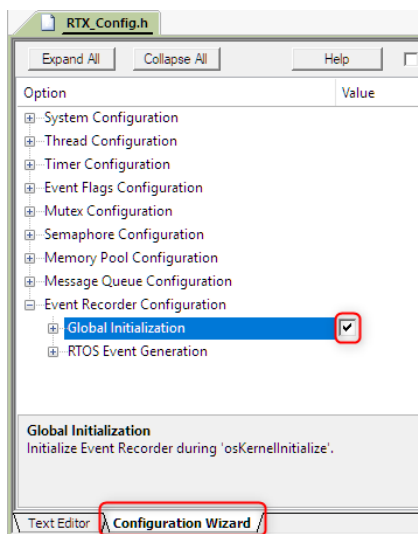
▪ In the **Project** window under **Compiler** component open
EventRecorderConf.h file, switch to the **Configuration Wizard** view,
expand **Event Recorder** group and specify 6000000000 as the **Time Stamp
Clock Frequency [Hz]**. This ensures correct timestamping for this project.
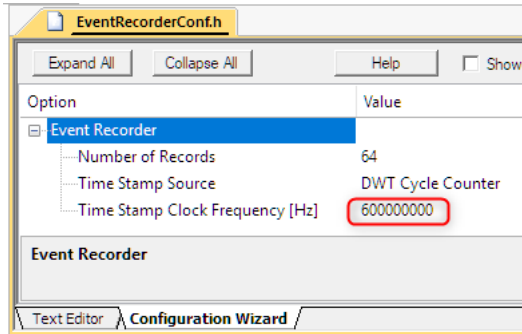
▪ Rebuild the project, download the code to the target and start a debug session.

    Open the event recorder window from the toolbar or the menu using
View – Analysis Windows – Event Recorder.

While debugging, events issued by Keil RTX5 are displayed in this window.
Event Recorder Configuration group in the *RTX_Config.h* file allows further
to configure the events to be generated by RTX and captured by Event Recorder.

The documentation explains how to use Event Recorder in a user application:
keil.com/pack/doc/compiler/EventRecorder/html/index.html

System Analyzer

The **System Analyzer** window provides a graphical analysis tool that can be used with any Arm Cortex-M based device. It shows:

- Incoming events from Compiler:Event Recorder.

- RTX5 RTOS thread events and status.

- Power measurement data (requires ULINK*plus* debug adapter).

- Exceptions (requires SWO trace and ULINK*pro* or ULINK*plus*).

- Value changes of VTREGs or variables (requires SWO trace).

Open the **System Analyzer** from the toolbar or via the menu **View** - **Analysis Windows** - **System Analyzer.**
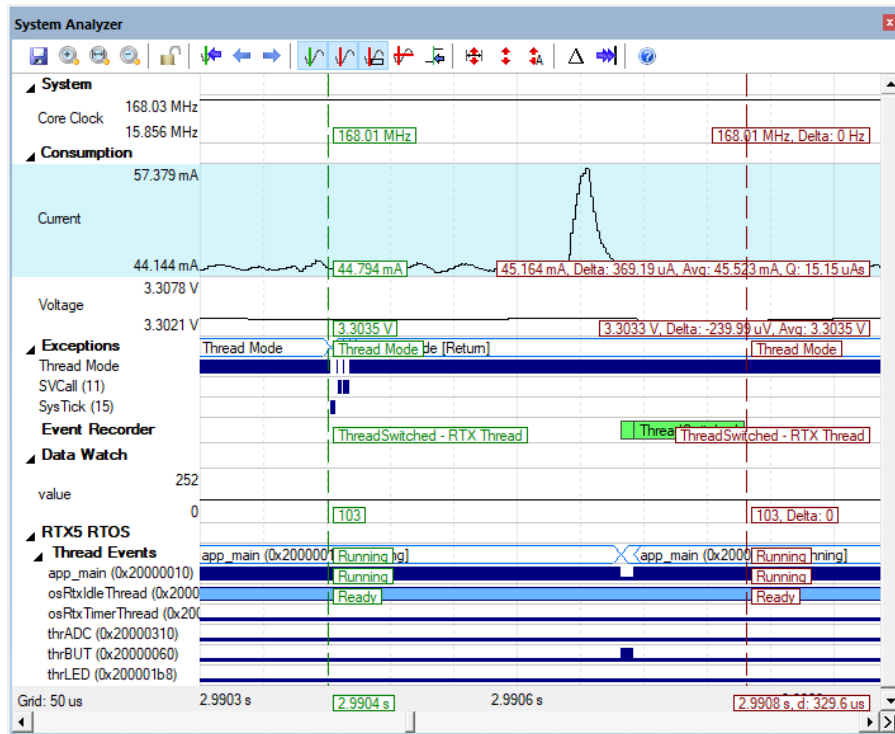
For more details refer to documentation:
keil.com/support/man/docs/uv4/uv4_db_dbg_systemanalyzer.htm

Breakpoints

You can set breakpoints

- While creating or editing your program source code. Click in the grey margin of the editor or **Disassembly** window to set a breakpoint.

- Using the breakpoint buttons in the toolbar.

- Using the menu Debug – Breakpoints.

- Entering commands in the **Command** window.

- Using the context menu of the **Disassembly** window or editor.

Breakpoints Window

You can define complex breakpoints using the **Breakpoints** window.

Open the **Breakpoints** window from the menu **Debug**.

Enable or disable breakpoints using the checkbox in the field **Current Breakpoints**. Double-click on an existing breakpoint to modify the definition.
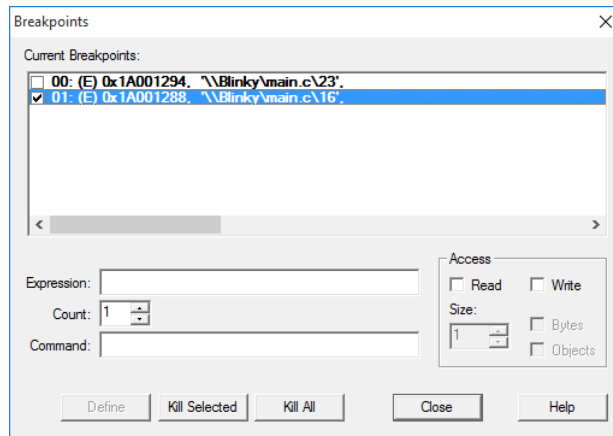
Enter an **Expression** to add a new breakpoint. Depending on the expression, one of the following breakpoint types is defined:

- **Execution Breakpoint (E)**: is created when the expression specifies a code address and triggers when the code address is reached.

- **Access Breakpoint (A)**: is created when the expression specifies a memory access (read, write, or both) and triggers on the access to this memory address. Use a compare (==) operator to compare for a specified value.

If a **Command** is specified for a breakpoint, µVision executes the command and resumes executing the target program.

The **Count** value specifies the number of times the breakpoint expression is true before the breakpoint halts program execution.

Watch Window

The **Watch** window allows you to observe
program symbols, registers, memory areas,
and expressions.

📰   Open a **Watch** window from the
toolbar or the menu using
View – Watch Windows.

Add variables to the **Watch** window with:

▪   Click on the field **<Enter expression>** and double-click or press **F2**.

▪   In the Editor when the cursor is located on a variable, use the context menu
select **Add** *<item name>* **to…**

▪   Drag and drop a variable into a **Watch** window.

▪   In the **Command** window, use the **WATCHSET** command.

The window content is updated when program execution is halted, or during
program execution when **View – Periodic Window Update** is enabled.

Call Stack and Locals Window

The **Call Stack + Locals** window
shows the function nesting and
variables of the current program
location.

🔲   Open the **Call Stack + Locals**
window from the toolbar or
the menu using **View – Call
Stack Window**.

When program execution stops, the **Call Stack + Locals** window automatically
shows the current function nesting along with local variables. Threads are shown
for applications that use the CMSIS-RTOS RTX.

Register Window

The **Register** window shows the content of the microcontroller registers.

Open the **Registers** window from the toolbar or the menu **View – Registers Window**.

You can modify the content of a register by double-clicking on the value of a register, or pressing **F2** to edit the selected value. Currently modified registers are highlighted in blue. The window updates the values when program execution halts.
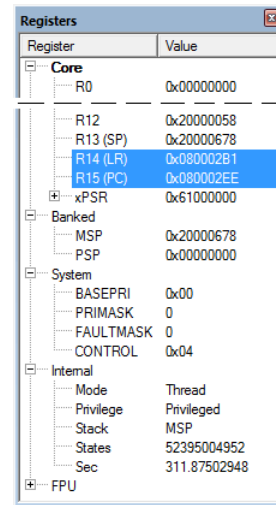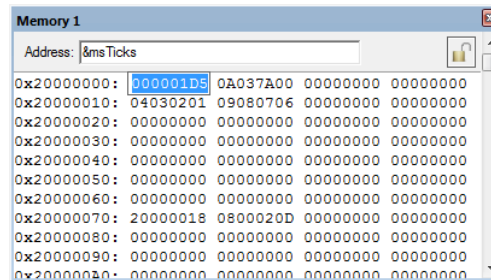
Memory Window

Monitor memory areas using **Memory Windows**.

Open a **Memory** window from the toolbar or the menu using **View – Memory Windows**.

- Enter an expression in the **Address** field to monitor the memory area.

- To modify memory content, use the **Modify Memory at …** command from context menu of the **Memory** window double-click on the value.

- The **Context Menu** allows you to select the output format.

- To update the **Memory Window** periodically, enable **View – Periodic Window Update**. Use **Update Windows** in the **Toolbox** to refresh the windows manually.

Stop refreshing the **Memory** window by clicking the **Lock** button. You can use the Lock feature to compare values of the same address space by viewing the same section in a second **Memory** window.

Peripheral Registers

Peripheral registers are memory mapped registers to which a processor can write
to and read from to control a peripheral. The menu **Peripherals** provides access
to **Core Peripherals**, such as the Nested Vector Interrupt Controller or the
System Tick Timer. You can access device peripheral registers using the **System
Viewer**.

NOTE
The content of the menu Peripherals changes with the selected microcontroller.

System Viewer

System Viewer windows display information
about device peripheral registers.

Open a peripheral register from the toolbar
or the menu **Peripherals – System
Viewer**.

With the **System Viewer,** you can:

- View peripheral register properties and
  values. Values are updated periodically
  when **View — Periodic Window Update**
  is enabled.

- Change property values while debugging.

- Search for specific properties using **TR1 Regular Expressions** in the search
  field. The appendix of the *__uVision User's Guide__* describes the syntax of
  regular expressions.

For details about accessing and using peripheral registers, refer to the online
documentation.
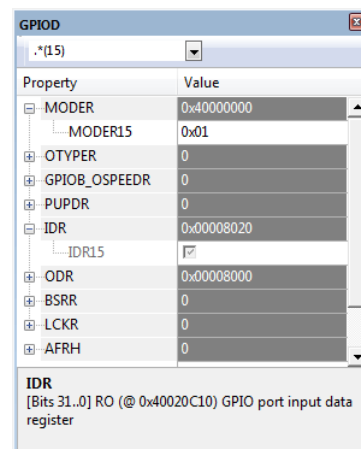
Trace

Run/stop debugging, as described previously, has some limitations that become apparent when testing time-critical programs, such as motor control or complex communication applications. As an example, breakpoints and single stepping commands change the dynamic behavior of the system. As an alternative, use the trace features explained in this section to analyze running systems.

Arm Cortex-M processors integrate CoreSight logic that is able to generate the following trace information using:

- **Data Watchpoints** record memory accesses with data value and program address and, optionally, stop program execution.

- **Exception Trace** outputs details about interrupts and exceptions.

- **Instrumented Trace** communicates program events and enables printf-style debug messages and the RTOS Event Viewer.

- **Instruction Trace** streams the complete program execution for recording and analysis.
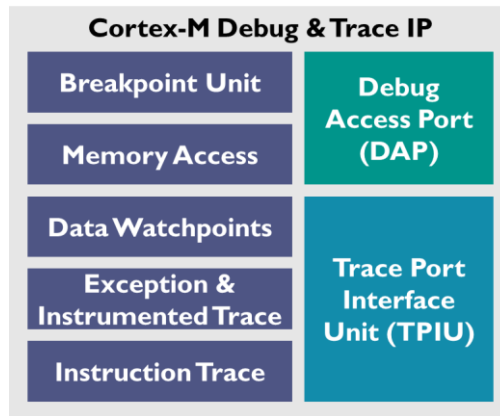
The **Trace Port Interface Unit** (TPIU) is available on most Cortex-M3, Cortex-M4, and Cortex-M7 processor-based microcontrollers and outputs above trace information via:

- **Serial Wire Trace Output** (SWO) works only in combination with the Serial Wire Debug mode (not with JTAG) and does not support Instruction Trace.

- **4-Pin Trace Output** is available on high-end microcontrollers and has the high bandwidth required for Instruction Trace.

- On some microcontrollers, the trace information can be stored in an on-chip **Trace Buffer** that can be read using the standard debug interface.

- Cortex-M3, Cortex-M4, and Cortex-M7 has an optional **Embedded Trace Buffer** (ETB) that stores all trace data described above.

- Cortex-M0+ has an optional **Micro Trace Buffer** (MTB) that supports instruction trace only.

The required trace interface needs to be supported by both the microcontroller and the debug adapter. The following table shows supported trace methods of various debug adapters.

Feature	ULINK*pro*	ULINK*plus*	ULINK2
Serial Wire Output (SWO)	✓	✓	✓
Maximum SWO Clock Frequency	200 MHz	60 MHz	3.75 MHz
4-Pin Trace Output for Streaming Trace	✓		
Embedded Trace Buffer (ETB) Support	✓	✓	✓
Micro Trace Buffer (MTB) Support	✓	✓	✓

Trace with Serial Wire Output

To use the serial wire trace output (SWO), use the following steps:

Click **Options for Target** on the toolbar and select the **Debug** tab. Verify that you have selected and enabled the correct *debug adapter*.
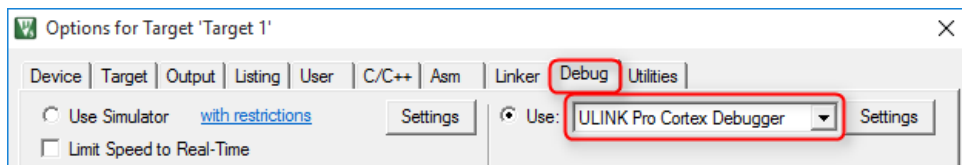
Click the **Settings** button. In the **Debug** dialog, select the debug **Port:** *SW* and set the **Max Clock** frequency for communicating with the debug unit of the device.
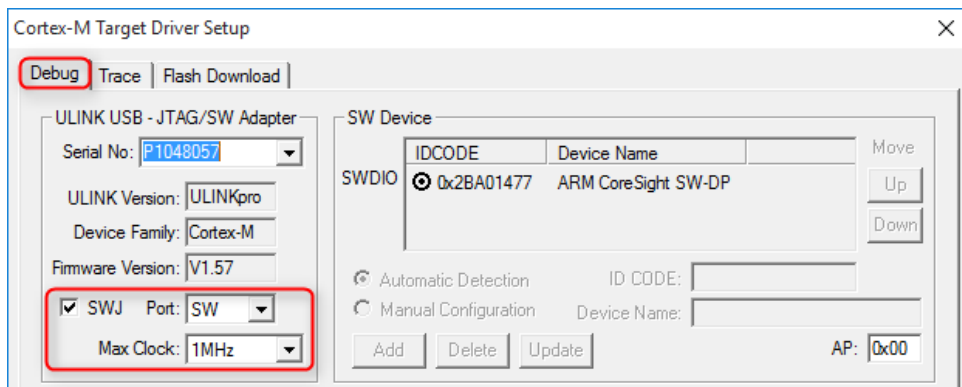
☞ Click the **Trace** tab. Ensure the **Core Clock** matches the System Core Clock the MCU is running at. Set **Trace Enable** and select the **Trace Events** you want to monitor.

Enable **ITM Stimulus Port 0** for `printf`-style debugging when using ITM as the output channel.
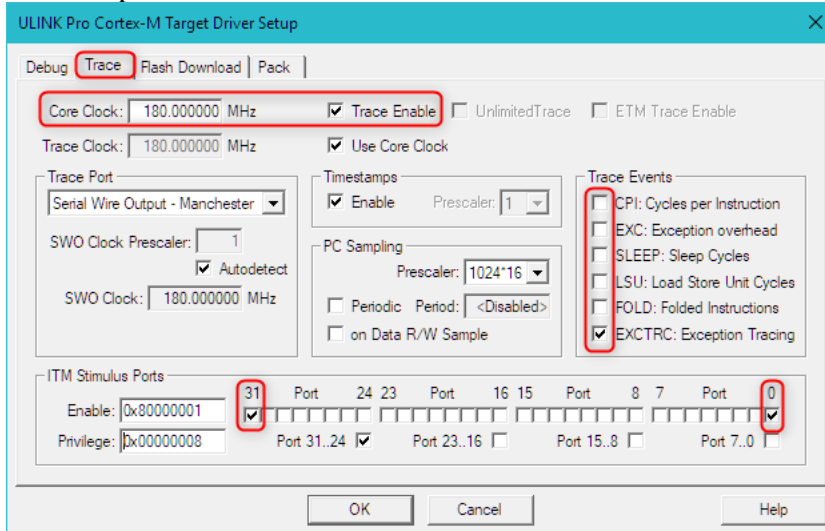
Trace Exceptions

The **Exception Trace** window displays statistical data about exceptions and interrupts.

Click on **Trace Windows** and select **Trace Exceptions** from the toolbar or use the menu **View – Trace – Trace Exceptions** to open the window.

Num	Name	Count	Total Time	Min Time In	Max Time...	Min Time Out	Max Time Out	First Time [s]	Last Time [s]
6	UsageFault	0	0 s						
11	SVCall	0	0 s						
12	DebugMonitor	0	0 s						
14	PendSV	0	0 s						
15	SysTick	1258	74.643 us	59.524 ns	59.524 ns	136.905 ns	1.000 ms	0.00103092	1.25403151
16	WWDG	0	0 s						
17	PVD	0	0 s						
18	TAMP_STAMP	0	0 s						
19	RTC_WKUP	0	0 s						

To retrieve data in the **Trace Exceptions** window:

- Set **Trace Enable** in the Debug Settings Trace dialog as described above.

- Enable **EXCTRC: Exception Tracing**.

- Set **Timestamps Enable**.

NOTE
The variable accesses configured in the Logic Analyzer are also shown in the
Trace Data Window.

Logic Analyzer

The Logic Analyzer window displays changes of up to four variable values over time. To add a variable to the Logic Analyzer, right click it in while in debug mode and select **Add <variable> to… - Logic Analyzer**. Open the Logic Analyzer window by choosing **View - Analysis Windows - Logic Analyzer**.
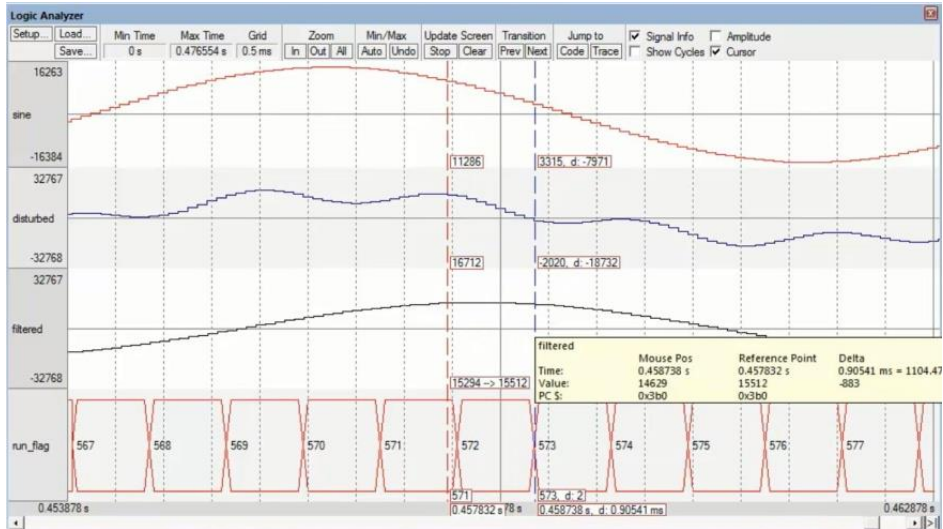
To retrieve data in the **Logic Analyzer** window:

- Set **Trace Enable** in the Debug Settings Trace dialog as described above.

- Set Timestamps Enable.

NOTE
*The variable accesses monitored in the Logic Analyzer are also shown in the Trace Data Window. Refer to the **µVision User's Guide – Debugging** for more information.*

Debug (printf) Viewer

The **Debug (printf) Viewer** window displays data streams that are transmitted sequentially through the **ITM Stimulus Port 0**. To enable *printf( )* debugging**,** use the

Compiler:I/O software component as described on page 40.

This *fputc( )* function redirects any *printf( )* messages (as shown below) to the **Debug (printf) Viewer**.

```
int seconds;                           // Second counter
   :
while (1) {
  LED_On ();                           // Switch on
  delay ();                            // Delay
  LED_Off ();                          // Switch off
  delay ();                            // Delay
  printf ("Seconds=%d\n", seconds++);  // Debug output
 }
```

Click on **Serial Windows** and select **Debug (printf) Viewer** from the toolbar or use the menu **View – Serial Windows – Debug (printf) Viewer** to open the window.
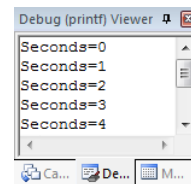
To retrieve data in the **Debug (printf) Viewer** window:
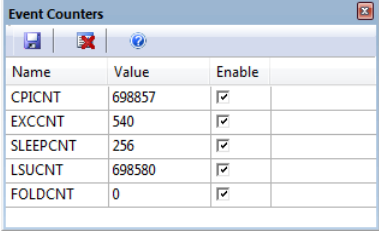
- Set **Trace Enable** in the Debug Settings Trace dialog as described above.

- Set Timestamps Enable.

- Enable ITM Stimulus Port **0**.

- Alternatively, on targets that do not support ITM (such as Arm Cortex-M0/M0+), you can use the event recorder to display printf messages. The Compiler component documentation explains how to enable this feature: **keil.com/pack/doc/compiler/RetargetIO/html/_retarget__examples_er.html**

Event Counters

Event Counters displays cumulative numbers, which show how often an event is triggered.

From toolbar use **Trace Windows – Event Counters**

From menu **View – Trace – Event Counters**

To retrieve data in this window:

- Set **Trace Enable** in the Debug Settings Trace dialog as described above.

- Enable **Event Counters** as needed in the dialog.

Event counters are performance indicators:

- **CPICNT**: Exception overhead cycle: indicates Flash wait states.

- **EXCCNT**: Extra Cycle per Instruction: indicates exception frequency.

- **SLEEPCNT**: Sleep Cycle: indicates the time spend in sleep mode.

- **LSUCNT**: Load Store Unit Cycle: indicates additional cycles required to execute a multi-cycle load-store instruction.

- **FOLDCNT**: Folded Instructions: indicates instructions that execute in zero cycles.

Trace with 4-Pin Output

Using the 4-pin trace output provides all the features described in the section **Trace with Serial Wire Output**, but has a higher trace communication bandwidth. Instruction trace is also possible.

The **ULINKpro debug/trace adapter** supports this parallel 4-pin trace output (also called ETM Trace) which gives detailed insight into program execution.

NOTE
*Refer to the **µVision User's Guide – Debugging** for more information about the features described below.*

When used with ULINKpro, MDK can stream the instruction trace data for the following advanced analysis features:

- **Code Coverage** marks code that has been executed and gives statistics on code execution. This helps to identify sporadic execution errors and is frequently a requirement for software certification.

- The **Performance Analyzer** records and displays execution times for functions and program blocks. It shows the processor cycle usage and enables you to find hotspots in algorithms for optimization.

- The **Trace Data Window** shows the history of executed instructions for Cortex-M devices.

Trace with On-Chip Trace Buffer

- In some cases, trace output pins are not available on the microcontroller or target hardware. As an alternative, an on-chip **Trace Buffer** can be used that supports the **Trace Data Window**.

MDK-Middleware
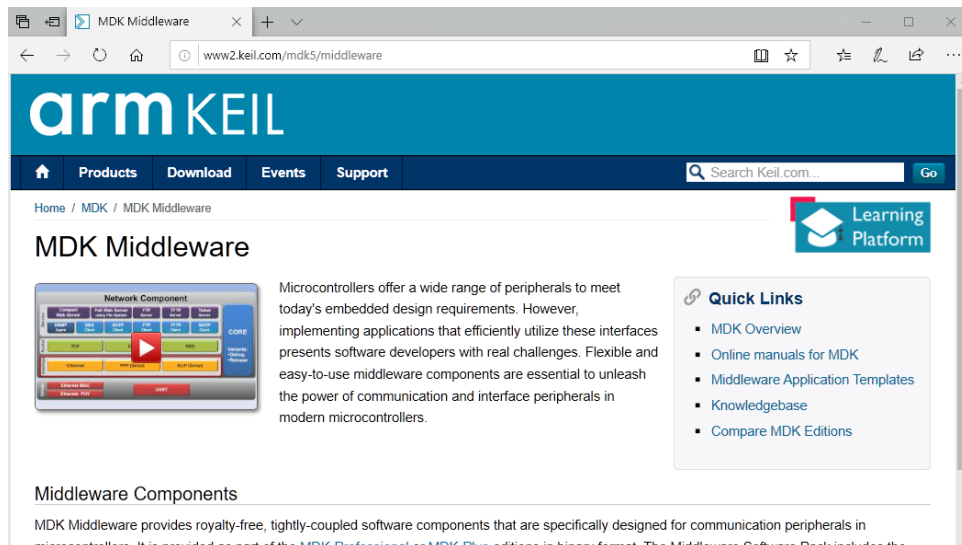
Today's microcontroller devices offer a wide range of communication peripherals to meet many embedded design requirements. Middleware is essential to make efficient use of these complex on-chip peripherals.

NOTE

This chapter describes the middleware that is part of MDK-Professional and MDK-Plus. MDK also works with some third-party middleware stacks. Refer to **keil.com/pack** *for a list of public software packs.*

The **MDK-Middleware** software pack includes royalty-free middleware with components for **TCP/IP networking**, **USB Host** and **USB Device** communication, **File System** for data storage, and a **graphical user interface**.

Refer to **keil.com/middleware** for more information.

This web page provides an overview of the middleware and links to:

- MDK-Middleware **User's Guide**
- **Device List** along with information about device-specific drivers
- Information about **Example Projects** with usage instructions

The Middleware interfaces to the device peripherals using device-specific CMSIS-Drivers. Refer to **CMSIS-Driver** on page 32 for more information.

Combining several components is common for a microcontroller application. The **Manage Run-Time Environment** dialog makes it easy to select and combine different MDK-Middleware components. It is even possible to expand the middleware component list with third-party components that are supplied as a software pack.

Typical examples for the usage of MDK-Middleware are:

- Web server with storage capabilities: Network and File System Component

- USB memory stick: USB Device and File System Component

- Industrial control unit with display and logging functionality: Graphics, USB Host, and File System Component

- Refer to the **FTP Server Example** on page 96 that exemplifies a combination of several middleware components.

The following sections give an overview for each software component of the MDK-Middleware.

NOTE
*A thirty days evaluation license for MDK-Professional is delivered with each installation. Refer to the **Installation** chapter on page 9 for more information.*

Network Component

The **Network Component** uses TCP/IP communication protocols and contains support for services, protocol sockets, and physical communication interfaces. It supports IPv4 and IPv6 connections.
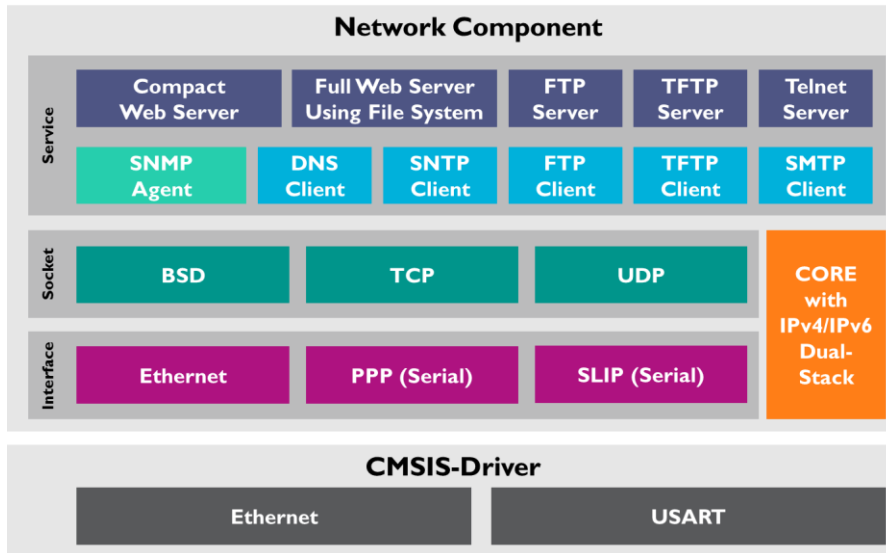
The various **services** provide program templates for common networking tasks.

- **Compact Web Server** stores web pages in ROM whereas the **Full Web Server** uses the **File System** component for page data storage. Both servers support dynamic page content using CGI scripting, AJAX, and SOAP technologies.

- **FTP** or **TFTP** support file transfer. FTP provides full file manipulation commands, whereas TFTP can boot load remote devices. Both are available for the client and server.

- **Telnet Server** provides a command line interface over an IP network.

- **SNMP Agent** reports device information to a network manager using the Simple Network Management Protocol.

- **DNS Client** resolves domain names to the respective IP address. It makes use of a freely configurable name server.

- **SNTP Client** synchronizes clocks and enables a device to get an accurate time signal over the data network.

- **SMTP Client** sends status emails using the Simple Mail Transfer Protocol.

All **Services** rely on a communication socket that can be either **TCP** (a connection-oriented, reliable full-duplex protocol), **UDP** (transaction-oriented protocol for data streaming), or **BSD** (Berkeley Sockets interface).

The physical **interface** can be either **Ethernet** (for LAN connections) or a serial connection such as **PPP** (for a direct connection between two devices) or **SLIP** (Internet Protocol over a serial connection).

Depending on the interface, the Network Component relies on a **CMSIS-Driver** to be present for providing the device-specific hardware interface. Ethernet requires an **Ethernet MAC** and **PHY** driver, whereas serial connections (PPP/SLIP) require a **UART** or a **Modem** driver.

The **Network Core** is available in a *Debug* variant with extensive diagnostic messages and a *Release* variant that omits these diagnostics. It supports IP communication using IPv4 and IPv6. To see events coming from the network component in the event recorder, you need to enable a debug variant.

File System Component

The **File System Component** allows your embedded applications to create, save, read, and modify files in storage devices such as RAM, NAND or NOR Flash, memory cards, or USB memory sticks.
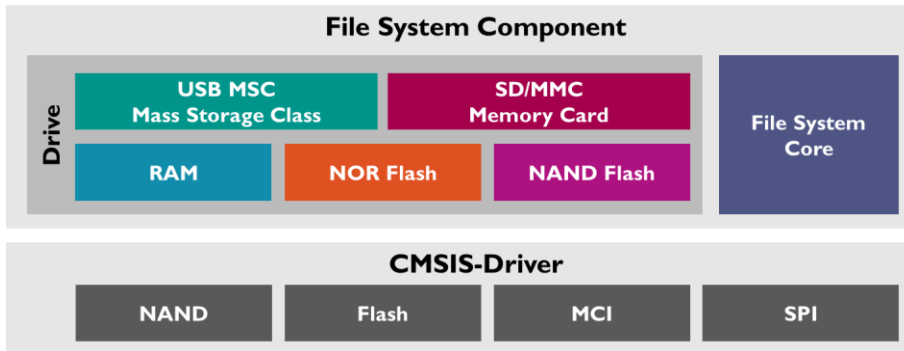
Each storage device is accessed and referenced as a **Drive**. The File System Component supports multiple drives of the same type. For example, you might have more than one memory card in your system.

The **File System Core** is thread-safe, supports simultaneous access to multiple drives, and uses a FAT system available in two file name variants: Short File Name (SFN) and Long File Name (LFN) with up to 255 characters. It also provides a *Debug* variant with extensive diagnostic messages and a *Release* variant that omits these diagnostics. To see events coming from the file system component in the event recorder, you need to enable a debug variant.

To access the physical media, for example NAND and NOR Flash chips, or memory cards using MCI or SPI, **CMSIS-Driver** have to be present.

USB Component

The **USB Device component** implements USB Host and Device functionality and uses standard device driver classes that are available on most computer systems, avoiding host driver development.
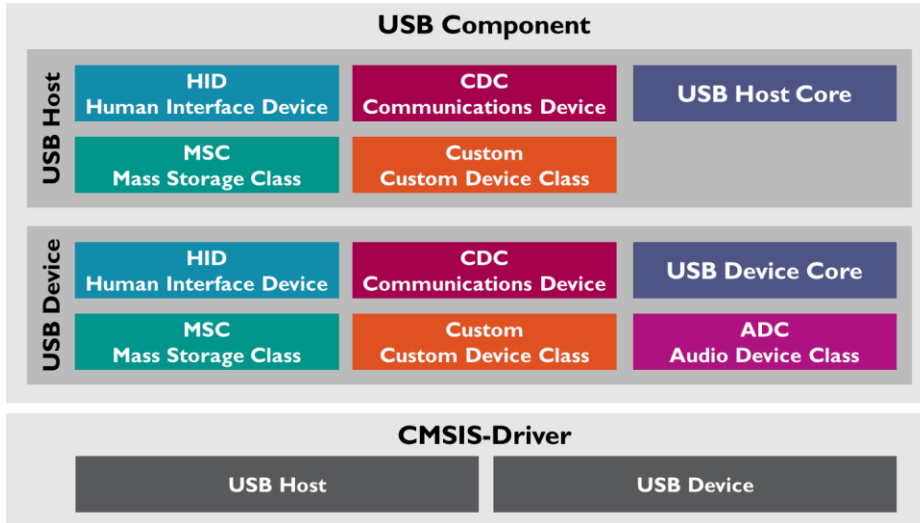
- **Human Interface Device Class** (HID) implements a keyboard, joystick or mouse. However, HID can also be used for simple data exchange.

- Use the **Mass Storage Class** (MSC) for file exchange (for example a USB memory stick).

- **Communication Device Class** (CDC) implements a virtual serial port (using the sub-class ACM) or a network connection (using the sub-class NCM).

- **Audio Device Class** (ADC) performs audio streaming.

- Use the **Custom Class** for new or unsupported USB classes.

The USB Component supports **Composite USB devices** that implement multiple device classes.

This component requires a **USB CMSIS-Driver** to be present. Depending on the application, it has to comply with the USB 1.1 (Full-Speed USB) and/or the USB 2.0 (High-Speed USB) specification.

The **USB Core** is available in a *Debug* variant with extensive diagnostic messages and a *Release* variant that omits these diagnostics. To see events coming from the USB component in the event recorder, you need to enable the debug variant.

Graphics Component

The **Graphics Component** is a comprehensive library that includes everything you need to build graphical user interfaces.
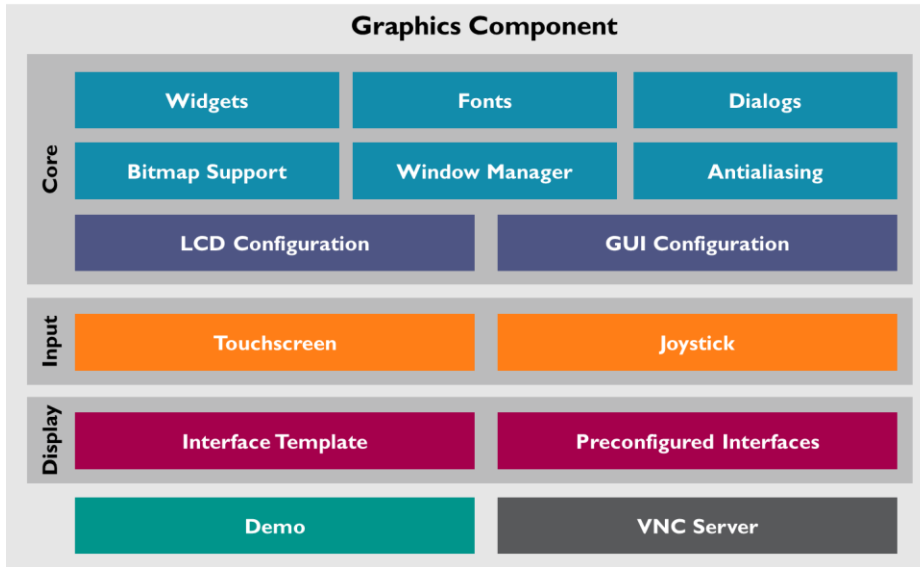
Core functions include:

- A **Window Manager** to manipulate any number of windows or dialogs.

- Ready-to-use **Fonts** and window elements**,** called **Widgets**, and **Dialogs.**

- **Bitmap Support** including JPEG and other common formats.

- **Anti-Aliasing** for smooth display.

- Flexible, configurable **Display** and **User Interface** parameters.

- The user interface can be controlled using input devices like a **Touch Screen** or a **Joystick**.

The Graphics Component interfaces to a wide range of display controllers using **preconfigured interfaces** for popular displays. Adapt the **interface template** to add support for new displays.

The **VNC Server** allows remote control of your graphical user interface via TCP/IP using the **Network Component**.

Demo shows all main features and is a rich source of code snippets for the GUI.

Mbed IoT Componentes

Keil MDK provides interfaces to Mbed software components that enable secure communication and Internet of Things (IoT) connectivity.

- **Mbed TLS** adds cryptographic and SSL/TLS capabilities with a library collection optimized for embedded systems.

- **Mbed Crypto** supports a wide range of cryptographic operations and provides a reference implementation of the cryptography interface of the Arm Platform Security Architecture (PSA).

FTP Server Example

The FTP server example is a reference application that shows a combination of several middleware components. Refer to **Verify Installation using Example Projects** on page 14 for more information on the various example projects that are available.

When using an FTP Server, you can exchange and manipulate files over a TCP/IP network. The middleware documentation has more details about the FTP Server and the reference application:

Several middleware components are the building blocks of this FTP server. A **File System** is required to handle the file manipulation. Various parts of the **Network** component build up the networking interface.

The following software components from the MDK-Middleware are required to create the FTP Server example:

As explained before, **CMSIS-Driver** provides the interface between the microcontroller peripherals and the MDK-Middleware.

The **Manage Run-Time Environment** dialog shows the software components selected for the FTP Server example:

Using Middleware

Create your own applications using MDK-Middleware components. For more information, refer to the MDK-Middleware User's Guide that has sections for every component describing:

- **Example projects** outline key product features of software components. The examples are tested, implemented, and proven on several evaluation boards.

- **Resource Requirements** describe the thread and stack resources for CMSIS-RTOS and the memory footprint.

- **Create an Application** contains the required steps for using the components in an embedded application.

- **Reference** contains the API and file documentation.
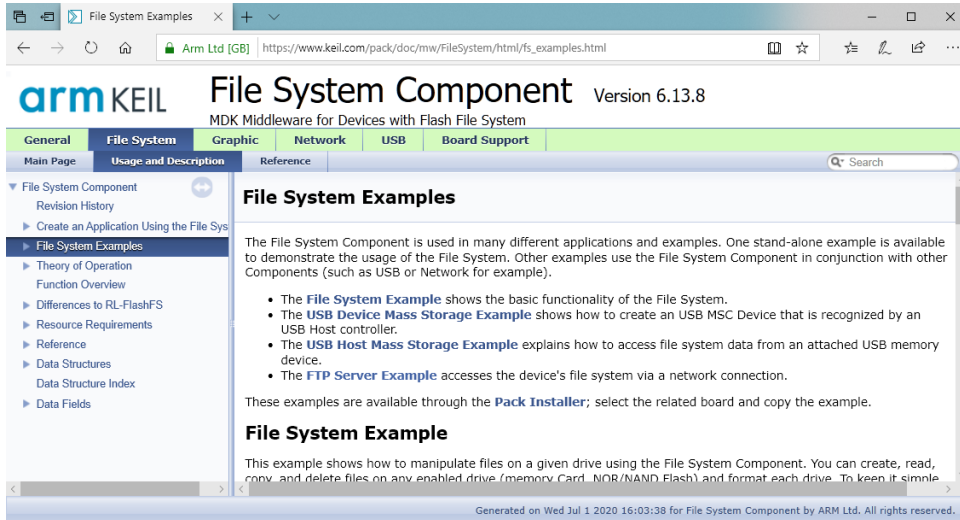
The learning platform **keil.com/learn** offers several tutorials and videos that exemplify typical use cases of the middleware. Refer also to these application notes:

- USB Host Application with File System and Graphical User Interface:
  keil.com/appnotes/docs/apnt_268.asp

- Web-Enabled MEMS Sensor Platform:
  keil.com/appnotes/docs/apnt_271.asp

- Web-Enabled Voice Recorder:
  keil.com/appnotes/docs/apnt_272.asp

- Analog/Digital Data Logger with USB Device Interface:
  keil.com/appnotes/docs/apnt_273.asp

The generic steps to use the various middleware components are:

- **Add Software Components:** in the **Manage Run-Time Environment** dialog select the software components that are required for your application.

- ▪ **Configure Middleware**: adjust the parameters of the software components in the related configuration files.

- ▪ **Configure Drivers**: identify and configure the peripheral interfaces that connect the middleware components to physical I/O pins of the microcontroller.

- ▪ **Implement Application Features**: use the API functions of the selected components to implement the application specific behaviour. Code templates help you to create the related source code.

- ▪ **Build and Download**: after compiling and linking of the application use the steps described in the chapter **Using the Debugger** to download the image to your target hardware.

- ▪ **Verify and Debug**: test utilities along with debug and trace features are described in the chapter **Create Applications**.

USB Device HID Example

While above steps are generic and apply to all components of the MDK-Middleware, the following USB Device HID example shows these steps in practice. This example creates a USB HID Device application that connects a microcontroller to a host computer via USB. On the PC the utility program *HIDClient.exe* is used to control LEDs on the development board.

This USB Device HID example uses the MIMXRT1050-EVK development board populated with a MIMXRT1052DVL6B microcontroller. It is based on the project created in section **Project with CMSIS-RTOS2** along with the source files *main.c*, *led_blinky.c* and the configuration files.

NOTE
You must adapt the code and configurations when using this example on other starter kits or evaluation boards.
The HID USB example is also available as a pre-built project in Pack Installer for many microcontroller device families supporting USB CMSIS_Driver.

Add Software Components

To create the USB Device HID example, start with the project described in section **Project with CMSIS-RTOS2**.

❖ Use the **Manage Run-Time Environment** dialog to add specific software components.

From CMSIS-Driver component:

▪ Select from **::CMSIS Driver:USB Device (API)** an appropriate driver suitable for your application. Some devices may have specific drivers for USB full-speed and high-speed whereas other microcontrollers may have a combined driver. Here, select **USB1**.
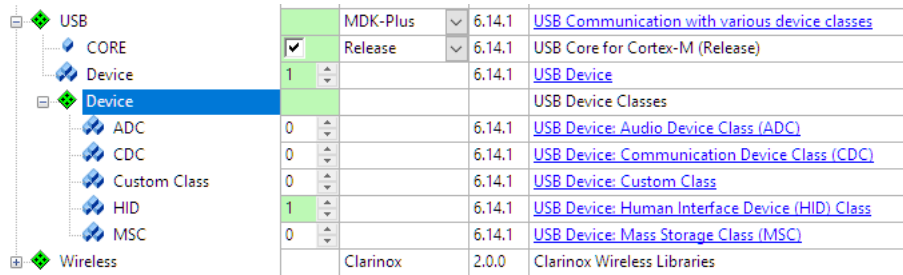

From *Device* component:

▪ Implementation of the USB CMSIS-Driver often relies on the vendor-specific HAL functions that also need to be added to the project.

In our case in **::Device:SDK Drivers** add **osa_bm** component to expose operating system abstraction used by the CMSIS-Driver. Other required HAL components are already selected in the initial CMSIS-RTOS2 example.

From *USB Component*:

▪ Select **::USB:CORE** to include the basic functionality required for USB communication.

▪ Set **::USB:Device** to '1' to create one USB Device instance.

▪ Set **::USB:Device:HID** to '1' to create a HID Device Class instance. If you select multiple instances of the same class or include other device classes, you will create a Composite USB Device.

⊟ ◆ USB		MDK-Plus ∨	6.14.1	USB Communication with various device classes
⌐ ● CORE	☑	Release ∨	6.14.1	USB Core for Cortex-M (Release)
◆ Device	1 ▲▼		6.14.1	USB Device
⊟ ◆ Device				USB Device Classes
◆ ADC	0 ▲▼		6.14.1	USB Device: Audio Device Class (ADC)
◆ CDC	0 ▲▼		6.14.1	USB Device: Communication Device Class (CDC)
◆ Custom Class	0 ▲▼		6.14.1	USB Device: Custom Class
◆ HID	1 ▲▼		6.14.1	USB Device: Human Interface Device (HID) Class
◆ MSC	0 ▲▼		6.14.1	USB Device: Mass Storage Class (MSC)
⊞ ◆ Wireless		Clarinox	2.0.0	Clarinox Wireless Libraries

TIP: Click on the hyperlinks in the Description column to view detailed documentation for each software component.

NOTE
For MDK-Middleware version older than v7.4.0, you also need to add the Keil RTX5 compatibility layer. Please select ::CMSIS:RTOS (API):Keil RTX5 if not present in the project yet.

Configure Middleware

Every MDK-Middleware component has a set of configuration files that adjusts application specific parameters and determines the driver interfaces. Access these configuration files from the **Project** window in the component class group. They usually have names like *<Component>_Config_0.c* or *<Component>_Config_0.h*.

Some of the settings in these files require corresponding settings in the driver and device configuration file (*RTE_Device.h*) that is subject of the next section.

For the USB HID Device example, there are two configuration files available: *USBD_Config_0.c* and *USBD_Config_HID_0.h*.
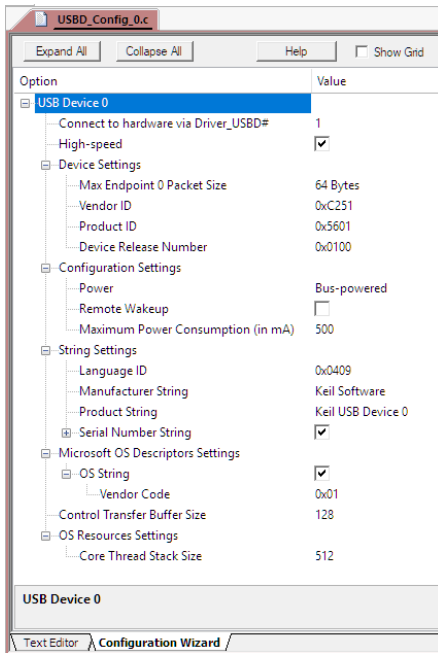
The file *USBD_Config_0.c* contains a number of important settings for the specific USB Device:

- The setting **Connect to Hardware via Driver_USBD#** specifies the *control struct* that reflects the peripheral interface, in this case, the USB controller used as device interface. For microcontrollers with only one USB controller the number is '1'. Refer to **CMSIS-Driver** section for more information.

- Select **High-Speed** if supported by the USB controller. Using this setting requires a driver that supports USB high-speed communication.

- Set the Max Endpoint 0 Packet Size to *64*.

- Set the **Vendor ID (VID)** to a private VID. The USB Implementer's Forum **www.usb.org/developers/vendor** provides more information on how to apply for a valid vendor ID.

- Every device needs a unique **Product ID**. The host computer's operating system uses it together with the VID to find a suitable driver for your device.

- Set the **Manufacturer** and the **Product String** to identify the USB device in PC operating systems.

The file *USBD_Config_HID_0.h* contains device class specific Endpoint settings. In our example, no changes are required.

Configure Drivers

Drivers have certain properties that define attributes such as I/O pin assignments, clock configuration, or usage of DMA channels. For many devices, the *RTE_Device.h* configuration file contains these driver properties. It typically requires configuration of the actual peripheral interfaces used by the application. Depending on the microcontroller device, you can enable different hardware peripherals, specify pin settings, or change the clock settings for your implementation.

In our example no changes from default driver configuration are required.

Implement Application Features

Now, create the code that implements the application specific features.

The middleware provides **User Code Templates** as starting point for the application software.

☞ In the **Project** window, right-click **Source Group 1** and open the dialog **Add New Item to Group**. Select the user code template from **::USB:Device:HID** - **USB Device HID (Human Interface Device)** and click **Add**.
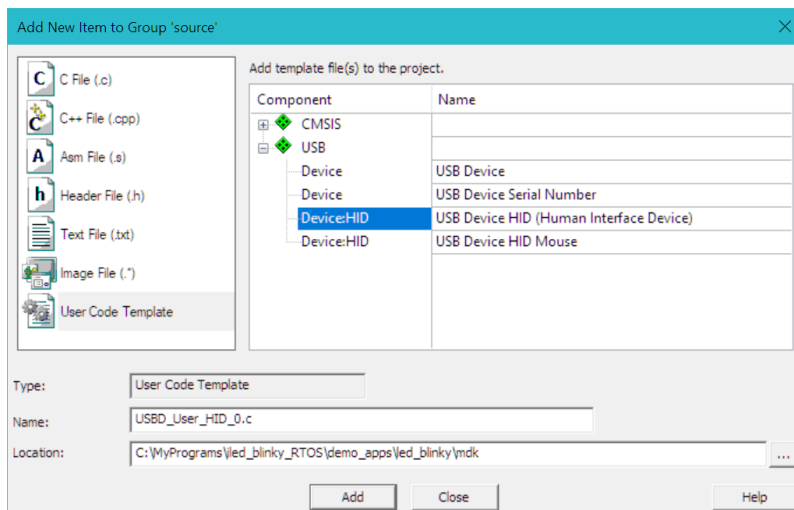
To connect the PC USB application to the microcontroller device, modify the function *USBD_HID0_SetReport()*, which handles data coming from the USB Host. For this example, the data will be created with the utility **HIDClient.exe**.

☞ Open the file *USBD_User_HID_0.c* in the editor and modify the code as shown below. This will control the LED on the evaluation board.

```c
#include "fsl_gpio.h"                          // Access to GPIO functions
#include "board.h"                             // Access to board LED defines
 :
bool USBD_HID0_SetReport (uint8_t rtype, uint8_t req, uint8_t rid,
                          const uint8_t *buf, int32_t len) {
  (void)req;
  (void)rid;
  (void)buf;
  (void)len;

  switch (rtype) {
    case HID_REPORT_OUTPUT:
      GPIO_PinWrite(BOARD_USER_LED_GPIO, BOARD_USER_LED_GPIO_PIN, *buf);
      break;
    case HID_REPORT_FEATURE:
      break;
    default:
      break;
  }
  return true;
}
```

In the file *led_blinky.c* we need to turn off the periodic LED blinking since the LED will be now controlled from the PC via USB. Also an additional RTOS thread is created to initialize the USB, read the button state and report it via the USB.

☞ Open the file *led_blinky.c* in the editor and modify the code as shown below.

```c
#include "cmsis_os2.h"
#include "fsl_gpio.h"
#include "pin_mux.h"
#include "board.h"
#include "rl_usb.h"

static osThreadId_t tid_thrLED; // Thread id of thread: LED
static osThreadId_t tid_thrSGN; // Thread id of thread: SGN
static osThreadId_t tid_thrUSB; // Thread id of thread: USB
/*------------------------------------------------------------------
  thrLED: blink LED
 *------------------------------------------------------------------*/
__NO_RETURN static void thrLED(void *argument) {
  (void)argument;
  uint32_t active_flag = 1U;

  for (;;) {
    osThreadFlagsWait(1U, osFlagsWaitAny, osWaitForever);
 // GPIO_PinWrite(BOARD_USER_LED_GPIO, BOARD_USER_LED_PIN, active_flag);
    active_flag=!active_flag;
  }
}
/*------------------------------------------------------------------
  thrSGN: Signal LED to change
 *------------------------------------------------------------------*/
```

```c
__NO_RETURN static void thrSGN(void *argument) {
  (void)argument;
  uint32_t last;

  for (;;) {
    osDelay(500U);                    // Run delay for 500 ticks
    osThreadFlagsSet(tid_thrLED, 1U); // Set flag to thrLED
  }
}
/*----------------------------------------------------------------------
  thrUSB: Init USB and report button state
 *---------------------------------------------------------------------*/
__NO_RETURN static void thrUSB(void *argument) {
  (void)argument;
  uint8_t but, but_state;

  USBD_Initialize        (0U);       // USB Device 0 Initialization
  USBD_Connect           (0U);       // USB Device 0 Connect

  for (;;) {
    but = GPIO_PinRead(BOARD_USER_BUTTON_GPIO,
                  BOARD_USER_BUTTON_GPIO_PIN)^1U;
    if (but != but_last) {
      but_last = but;
      USBD_HID_GetReportTrigger(0U, 0U, &but, 1U);
    }
    osDelay(100U);                 // 100 ms delay for sampling buttons
  }
}

/*----------------------------------------------------------------------
 * Application main thread
 *---------------------------------------------------------------------*/
void app_main(void *argument) {
  (void)argument;

  tid_thrLED = osThreadNew(thrLED, NULL, NULL); // Create LED thread
  if (tid_thrLED == NULL) { /* add error handling */  }

  tid_thrSGN = osThreadNew(thrSGN, NULL, NULL); // Create SGN thread
  if (tid_thrBUT == NULL) { /* add error handling */  }

  tid_thrUSB = osThreadNew(thrUSB, NULL, NULL); // Create USB thread
  if (tid_thrUSB == NULL) { /* add error handling */  }

  osThreadExit();
}
```

Build and Download

Build the project and download it to the target as explained in chapters **Create Applications** and **Using the Debugger**.

Verify and Debug

Connect the development board to your PC using another USB cable. This provides the connection to the USB device peripheral of the microcontroller.

Once the board is connected, a notification appears that indicates the installation of the device driver for the USB HID Device.

The utility program ***HIDClient.exe*** that is part of MDK enables testing of the connection between the PC and the development board. This utility is located the MDK installation folder *.\Keil\ARM\Utilities\HID_Client\Release*.
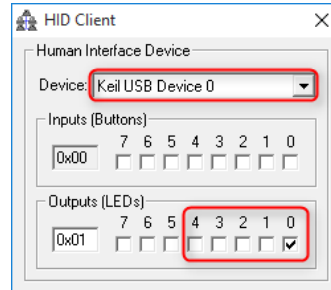
To test the functionality of the USB HID device run the ***HIDClient.exe*** utility and follow these steps:

- Select the Device to establish the communication channel. In our example, it is "*Keil USB Device 0*".

- Test the application by changing the **Outputs (LEDs)** checkboxes. The respective LEDs shall switch accordingly on the development board.

If you are having problems connecting to the development board, you can use the debugger to find the root cause.

From the toolbar, select **Start/Stop Debug Session**.

Use debug windows to narrow down the problem. Breakpoints help you to stop at certain lines of code so that you can examine the variable contents.

NOTE

Debugging of communication protocols can be difficult. When starting the debugger or using breakpoints, communication protocol timeouts may exceed making it hard to debug the application. Therefore, use breakpoints carefully.

In case that the USB communication fails, disconnect USB, reset your target hardware, run the application, and reconnect it to the PC.

Index