Getting Started
Create Applications with MDK Version 5 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.

Copyright © 1997-2014 ARM Germany GmbH
All rights reserved.

µVision®, Cortex®, CoreSight™, MDK™, RL-ARM™, ULINK®, Device Database®, and ARTX™ 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 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 Keil MDK Version 5 Microcontroller Development Kit available from ARM. To provide you with the very best software tools for developing Cortex-M processor based embedded applications we design our tools to make software engineering easy and productive. ARM also offers therefore 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 Core, the Software Packs, and describes the product installation along with the use of example projects.

CMSIS is ground-up 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.

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

Middleware gives further details on the middleware that is available for users of the MDK-Professional edition.

Using Middleware explains how to create applications that uses the middleware available with MDK-Professional and contains essential tips and tricks to get you started quickly.
Contents

Preface ........................................................................................................................................... 3

Contents ........................................................................................................................................ 4

MDK Introduction .......................................................................................................................... 6
MDK Core ....................................................................................................................................... 6
Software Packs ............................................................................................................................. 6
MDK Editions .................................................................................................................................. 7
Installation ...................................................................................................................................... 8
  Software and Hardware Requirements ......................................................................................... 8
  Install MDK Core ......................................................................................................................... 8
  Install Software Packs ................................................................................................................. 9
  Verify Installation using Example Projects .............................................................................. 10
  Use Software Packs .................................................................................................................... 14
Access Documentation .................................................................................................................. 16
Request Assistance ....................................................................................................................... 16

CMSIS ............................................................................................................................................ 17
CMSIS-CORE ................................................................................................................................. 18
  Using CMSIS-CORE .................................................................................................................... 18
CMSIS-RTOS RTX .......................................................................................................................... 21
  Software Concepts ...................................................................................................................... 21
  Using CMSIS-RTOS RTX ............................................................................................................ 22
  CMSIS-RTOS RTX API Functions .............................................................................................. 27
  CMSIS-RTOS User Code Templates ......................................................................................... 28
CMSIS-DSP ..................................................................................................................................... 37

Create Applications ..................................................................................................................... 39
Blinky with CMSIS-RTOS RTX ...................................................................................................... 40
Blinky with Infinite Loop Design ................................................................................................... 48

Debug Applications ....................................................................................................................... 51
Debugger Connection ...................................................................................................................... 51
Using the Debugger ....................................................................................................................... 52
  Debug Toolbar ........................................................................................................................... 53
  Command Window ...................................................................................................................... 54
  Disassembly Window ................................................................................................................... 54
  Breakpoints ................................................................................................................................. 55
  Watch Window ........................................................................................................................... 56
  Call Stack and Locals Window ................................................................................................... 56
  Register Window ......................................................................................................................... 57
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Window</td>
<td>57</td>
</tr>
<tr>
<td>Peripheral Registers</td>
<td>58</td>
</tr>
<tr>
<td>Trace</td>
<td>59</td>
</tr>
<tr>
<td>Trace with Serial Wire Output</td>
<td>60</td>
</tr>
<tr>
<td>Trace Exceptions</td>
<td>60</td>
</tr>
<tr>
<td>Logic Analyzer</td>
<td>63</td>
</tr>
<tr>
<td>Debug (printf) Viewer</td>
<td>64</td>
</tr>
<tr>
<td>Event Counters</td>
<td>65</td>
</tr>
<tr>
<td>Trace with 4-Pin Output</td>
<td>66</td>
</tr>
<tr>
<td>Trace with On-Chip Trace Buffer</td>
<td>66</td>
</tr>
<tr>
<td>RTOS Debugging</td>
<td>67</td>
</tr>
<tr>
<td>System and Thread Viewer</td>
<td>67</td>
</tr>
<tr>
<td>Event Viewer</td>
<td>68</td>
</tr>
<tr>
<td>Middleware</td>
<td>69</td>
</tr>
<tr>
<td>Network Component</td>
<td>71</td>
</tr>
<tr>
<td>File System Component</td>
<td>73</td>
</tr>
<tr>
<td>USB Device Component</td>
<td>74</td>
</tr>
<tr>
<td>USB Host Component</td>
<td>75</td>
</tr>
<tr>
<td>Graphics Component</td>
<td>76</td>
</tr>
<tr>
<td>Driver Components</td>
<td>77</td>
</tr>
<tr>
<td>FTP Server Example</td>
<td>78</td>
</tr>
<tr>
<td>Using Middleware</td>
<td>80</td>
</tr>
<tr>
<td>USB HID Example</td>
<td>81</td>
</tr>
<tr>
<td>Add Software Components</td>
<td>82</td>
</tr>
<tr>
<td>Configure Middleware</td>
<td>84</td>
</tr>
<tr>
<td>Configure Drivers</td>
<td>86</td>
</tr>
<tr>
<td>Adjust System Resources</td>
<td>87</td>
</tr>
<tr>
<td>Implement Application Features</td>
<td>88</td>
</tr>
<tr>
<td>Build and Download</td>
<td>91</td>
</tr>
<tr>
<td>Verify and Debug</td>
<td>91</td>
</tr>
<tr>
<td>Index</td>
<td>93</td>
</tr>
</tbody>
</table>
MDK Introduction

The Keil Microcontroller Development Kit (MDK) helps you to create embedded applications for ARM Cortex-M processor-based devices. MDK is a powerful, yet easy to learn and easy to use development system. MDK Version 5 consists of the MDK Core plus device-specific Software Packs, which can be downloaded and installed based on the requirements of your application.


MDK Core

The MDK Core includes all the components that you need to create, build, and debug an embedded application for Cortex-M processor based microcontroller devices. The Pack Installer manages Software Packs that can be added any time to the MDK Core. This makes new device support and middleware updates independent from the toolchain.

Software Packs

Software Packs contain device support, CMSIS libraries, middleware, board support, code templates, and example projects.
MDK Editions

MDK provides the tools and the environment to create and debug applications using C/C++ or assembly language and is available in various editions. Each edition includes the µVision IDE, debugger, compiler, assembler, linker, middleware libraries, and the RTX RTOS.

- **MDK-Professional** contains extensive middleware libraries for sophisticated embedded applications and all features of **MDK-Standard**.
- **MDK-Standard** supports Cortex-M, selected Cortex-R, and ARM7/9 processor based microcontrollers.
- **MDK-Cortex-M** supports Cortex-M processor based microcontrollers.
- **MDK-Lite** is code size restricted to 32 KB and intended for product evaluation, small projects, and the educational market.


License Types

With the exception of **MDK-Lite**, the MDK editions require activation using a license code. The following licenses types are available:

- **Single-User License** (Node-Locked) grants the right to use the product by one developer on two computers at the same time.
- **Floating-User License** or **FlexLM License** grants the right to use the product on several computers by a number of developers at the same time.

For further details, refer to the *Licensing User’s Guide* at [www.keil.com/support/man/docs/license](http://www.keil.com/support/man/docs/license).
Installation

Software and Hardware Requirements
MDK has the following minimum hardware and software requirements:

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

Install MDK Core
Download MDK-ARM v5 from www.keil.com/download - Product Downloads and run the installer.

Follow the instructions to install the MDK Core on your local computer. The installation also adds the Software Packs for ARM CMSIS and MDK-Professional Middleware.

After the MDK Core installation is complete, the Pack Installer is started 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.
Install Software Packs

The Pack Installer is a utility for managing Software Packs on the local computer.

The Pack Installer runs automatically during the installation but also may be run from µVision using the menu item Project – Manage – Pack Installer. To get access to devices and example projects you should 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 [www.keil.com/pack](http://www.keil.com/pack).

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

**TIP:** The Device Database at [www.keil.com/dd2](http://www.keil.com/dd2) lists all available devices and provides download access to the related Software Packs. If the Pack Installer cannot access [www.keil.com/pack](http://www.keil.com/pack) you can manually install Software Packs using the menu command File – Import or by double-clicking *.PACK files.
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 videos on [www.keil.com/mdk5/blinky](http://www.keil.com/mdk5/blinky) that explain how to connect and work with an evaluation kit.

Copy an Example Project

In the Pack Installer, select the tab *Examples*. Use filters 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**

Now 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 with the debugger.

The step-by-step instructions show you how to execute these tasks. After copying the example, µVision starts and looks similar to the picture below.

![µVision Interface](image)

**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 Window]

**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 and there is no 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 that you are using is selected and enabled. For example, **CMSIS-DAP Debugger** is a debug adapter that is part of several starter kits.

![Options for Target Window]
Click the Utilities tab to verify Flash programming. Enable Use Debug Driver to perform flash download via the debug adapter you selected on the Debug tab.

**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’s guide of the starter kit.

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

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.
Use Software Packs

Software Packs contain information for the µVision Device Database and software components that are available as building blocks for the application software.

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

Start µVision and use the menu item Project - New µVision Project. After you have selected a project directory and specified the project name, you will be asked to select a target device.

![Select Device for Target 'Target 1':](image)

**TIP:** You will see only devices that are part of the installed Software Packs. If you are missing a device, use the Pack Installer to add the related Software Pack.
After selecting the device, the **Manage Run-Time Environment** window shows the related software components for this device.

![Manage Run-Time Environment](image)

**TIP:** The links in the column *Description* provide access to the documentation of each software component.

**NOTE**
*The notation ::<Component Class>::<Group>::<Name>* is used to refer to components. For example, ::CMSIS:CORE refers to the component CMSIS-CORE selected in the dialog above.*
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.

You can access the latest information on the web at www.keil.com/mdk5.

Request Assistance

If you have suggestions or you have discovered an issue with the software, please report them to us. Support and information channels are accessible at www.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.
CMSIS

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

**NOTE**

This chapter is intended as reference section. The chapter *Create Applications* on page 39 shows you how to use CMSIS for creating application code.

The CMSIS, defined in close cooperation with various silicon and software vendors, 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-RTOS**: Provides standardized real-time operating systems and enables therefore software templates, middleware, libraries, and other components that can work across supported RTOS systems. This manual explains the usage of the CMSIS-RTOS RTX 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-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 uses no 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 central files:

**NOTE**

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

- The `startup_<device>.s` file with reset handler and exception vectors.
- The `system_<device>.c` configuration file for basic device setup (clock and memory BUS).
- The `<device>.h` include file for user code access to the microcontroller device.

The `<device>.h` header 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.
Adding Software 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:

![Manage Run-Time Environment](image)

The µVision environment adds the related files.

Source Code Example

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

Example of using the CMSIS-CORE layer

```c
#include "stm32f10x.h" // File name depends on device used

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;
    while (msTicks == curTicks) {
        __WFE (); // Wait for next SysTick Interrupt
    }
}

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_IRQHandler, 1); // Set Timer priority
    NVIC_EnableIRQ (TIM1_UP_IRQHandler); // Enable Timer Interrupt
}

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

// The processor clock is initialized by CMSIS startup + system file
void main (void) {
    Device_Initialization (); // Configure & Initialize MCU
    while (1) {
        disable_irq (); // Disable all interrupts
        Get_InputValues ();
        enable_irq (); // Enable all interrupts
        Process_Values ();
        WaitForTick (); // Synchronize to SysTick Timer
    }
}

For more information, refer to the CMSIS-CORE documentation. In the Project window, right-click the group CMSIS, and choose Open Documentation.
CMSIS-RTOS RTX

This section introduces the CMSIS RTOS RTX Real-Time Operating System, describes the advantages, and explains configuration settings and features of this RTOS.

NOTE
MDK is compatible with many third-party RTOS solutions, which you may use. However, CMSIS-RTOS RTX is well integrated into MDK, is feature-rich and tailored towards the requirements of deeply embedded systems.

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 preemptive 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 CMSIS-RTOS RTX, 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 handled by the RTOS Kernel, using a proven code base.
- The RTOS provides 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 and allow Top-Half/Bottom-Half handling of your interrupts.

Using CMSIS-RTOS RTX

CMSIS-RTOS RTX is implemented as a library and exposes the functionality through the header file `cmsis_os.h`.

Execution of the CMSIS-RTOS RTX starts with the function `main()` as the first thread. This has the benefit that developers can initialize other middleware libraries that create threads internally, but the remaining part of the user application uses just the `main` thread. Consequently, the usage of the RTOS can be invisible to the application programmer, but libraries can use CMSIS-RTOS RTX features.

The software component `::CMSIS:RTOS:Keil RTX` must be used together with the components `::CMSIS:CORE` and `::Device:Startup`. Selecting these components provides the following central CMSIS-RTOS RTX files:

**NOTE**

In the actual file names, `<device>` is the name of the microcontroller device; `<device core>` represents the device processor family.
• The file `RTX_<core>.lib` is the library with RTOS functions.

• The configuration file `RTX_Conf_CM.c` for defining thread options, timer configurations, and RTX kernel settings.

• The header file `cmsis_os.h` exposes the RTX functionality to the user application.

• The function `main()` is executed as a thread.

Once these files are part of the project, developers can start using the CMSIS-RTOS RTX functions. The code example shows the use of CMSIS-RTOS RTX functions:

**Example of using CMSIS-RTOS RTX functions**

```c
#include "cmsis_os.h" // CMSIS RTOS header file

void job1 (void const *argument) { // Function 'job1'
    // execute some code
    osDelay (10); // Delay execution for 10ms
}

osThreadDef (job1, osPriorityLow, 1, 0); // Define job1 as thread

int main (void) {
    osKernelInitialize (); // Initialize RTOS kernel
    // setup and initialize peripherals
    osThreadCreate (osThread(job1), NULL); // Create the thread
    osKernelStart (); // Start kernel & job1 thread
}
```
Header File cmsis_os.h

The file cmsis_os.h is a template header file for the CMSIS-RTOS RTX and contains:

- CMSIS-RTOS API function definitions.
- Definitions for parameters and return types.
- Status and priority values used by CMSIS-RTOS API functions.
- Macros for defining threads and other kernel objects such as mutex, semaphores, or memory pools.

All definitions are prefixed with os to give a unique name space for the CMSIS-RTOS functions. Definitions that are prefixed os_ are not be used in the application code but are local to this header file. All definitions and functions that belong to a module are grouped and have a common prefix, for example, osThread for threads.

Define and Reference Object Definitions

With the #define osObjectsExternal, objects are defined as external symbols. This allows creating a consistent header file for the entire project as shown below:

Example of a header file: osObjects.h

```c
#include "cmsis_os.h"       // CMSIS RTOS header
extern void thread_1 (void const *argument);  // Function prototype
osThreadDef (thread_1, osPriorityLow, 1, 100);  // Thread definition
osPoolDef(MyPool, 10, long);   // Pool definition
```

This header file, called osObjects.h, defines all objects when included in a C/C++ source file. When #define osObjectsExternal is present before the header file inclusion, the objects are defined as external symbols. Thus, a single consistent header file can be used throughout the entire project.

Consistent header file usage in a C file

```c
#define osObjectExternal       // Objects defined as external symbols
#include "osObjects.h"        // Reference to the CMSIS-RTOS objects
```

For details, refer to the online documentation section Header File Template: cmsis_os.h.
CMSIS-RTOS RTX Configuration

The file `RTX_Conf_CM.c` contains the configuration parameters of the CMSIS-RTOS RTX. A copy of this file is part of every project using the RTX component.

You can set parameters for the thread stack, configure the Tick Timer, set Round-Robin time slice, and define user timer behaviour for threads.

For more information about configuration options, open the RTX documentation from the Project window, right-click the file `RTX_Conf_CM.c` and choose Open documentation. The section Configuration of CMSIS-RTOS RTX describes all available settings. The following highlights the most important settings that need adaptation in your application.

Thread Stack Configuration

Threads are defined in the code with the function `osThreadDef()`. The parameter `stacksz` specifies the stack requirement of a thread and has an impact on the method for allocating stack. CMSIS-RTOS RTX offer two methods for allocating stack requirements in the file `RTX_Conf_CM.c`:

- Using a fixed memory pool: if the parameter `stacksz` is 0, then the value specified for Default Thread stack size [bytes] sets the stack size for the thread function.
- Using a user space: if `stacksz` is not 0, then the thread stack is allocated from a user space. The total size of this user space is specified by **Total stack size [bytes] for threads with user-provided stack size**.

**Number of concurrent running threads** specifies the maximum number of threads that allocate the stack from the fixed size memory pool.

**Default Thread stack size [bytes]** specifies the stack size (in words) for threads defined without a user-provided stack.

**Main Thread stack size [bytes]** is the stack requirement for the `main()` function.

**Number of threads with user-provided stack size** specifies the number of threads defined with a specific stack size.

**Total stack size [bytes] for threads with user-provided stack size** is the combined requirement (in words) of all threads defined with a specific stack size.

---

**NOTE**

Consider these settings carefully. If you do not allocate enough memory or you do not specify enough threads, your application will not work.

---

For details, refer to the online documentation section **Configuration of CMSIS-RTOS RTX – Thread Stack and Processor Mode**.
RTX Kernel Timer Tick Configuration

CMSIS-RTOS RTX functions provide delays in units of milliseconds derived from the **Timer tick value**. We recommend configuring the Timer Tick to generate 1-millisecond intervals. Configuring a longer interval may reduce energy consumption, but has an impact on the granularity of the timeouts.

<table>
<thead>
<tr>
<th>RTX Kernel Timer Tick Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Cortex-M SysTick timer as RTX Kernel Timer</td>
</tr>
<tr>
<td>Timer clock value [Hz]</td>
</tr>
<tr>
<td>Timer tick value [us]</td>
</tr>
</tbody>
</table>

It is good practice to enable **Use Cortex-M SysTick timer as RTX Kernel Timer**. This selects the built-in SysTick timer with the processor clock as the clock source. In this case, the **Timer clock value** should be **identical** to the CMSIS variable `SystemCoreClock` of the startup file `system_<device>.c`.

For details, refer to the online documentation section **Configuration of CMSIS-RTOS RTX – Tick Timer Configuration**.

CMSIS-RTOS RTX API Functions

The table below lists the various API function categories that are available with the CMSIS-RTOS RTX.

<table>
<thead>
<tr>
<th>API Category</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Management</td>
<td>Define, create, and control thread functions.</td>
<td>29</td>
</tr>
<tr>
<td>Timer Management</td>
<td>Create and control timer and callback functions.</td>
<td>31</td>
</tr>
<tr>
<td>Signal Management</td>
<td>Control or wait for signal flags.</td>
<td>32</td>
</tr>
<tr>
<td>Mutex Management</td>
<td>Synchronize thread execution with a Mutex.</td>
<td>33</td>
</tr>
<tr>
<td>Semaphore Management</td>
<td>Control access to shared resources.</td>
<td>33</td>
</tr>
<tr>
<td>Memory Pool Management</td>
<td>Define and manage fixed-size memory pools.</td>
<td>35</td>
</tr>
<tr>
<td>Message Queue Management</td>
<td>Control, send, receive, or wait for messages.</td>
<td>35</td>
</tr>
<tr>
<td>Mail Queue Management</td>
<td>Control, send, receive, or wait for mail.</td>
<td>36</td>
</tr>
</tbody>
</table>
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 and select CMSIS-RTOS Thread and click Add.
Thread Management

The Thread Management function group allows defining, creating, and controlling thread functions in the system. The function *main()* is a special thread function that is started at system initialization and has the initial priority *osPriorityNormal*.

The CMSIS-RTOS RTX assumes threads are scheduled as shown in the figure *Thread State and State Transitions*. Thread states change as described below:

- A thread is created using the function *osThreadCreate()* . This puts the thread into the READY or RUNNING state (depending on the thread priority).

- CMSIS-RTOS is pre-emptive. The active thread with the highest priority becomes the RUNNING thread provided it is not waiting for any event. The initial priority of a thread is defined with the function *osThreadDef()* but may be changed during execution using the function *osThreadSetPriority()*.

- The RUNNING thread transfers into the WAITING state when it is waiting for an event.

- Active threads can be terminated any time using the function *osThreadTerminate()* . Threads can also terminate by exit from the usual *forever loop* and just a *return* from the thread function. Threads that are terminated are in the INACTIVE state and typically do not consume any dynamic memory resources.
Single Thread Program

A standard C program starts execution with the function *main()*). Usually, this function is an endless loop in an embedded application and can be thought of as a single thread that is executed continuously. For example:

Main function as endless loop; Single thread design, no RTOS used

```c
int main (void) {
    int counter = 0;

    while (1) { // Loop forever
        counter++; // Increment counter
    }
}
```

Simple RTX Program using Round-Robin Task Switching

```c
#include "cmsis_os.h"

int counter1;
int counter2;

void job1 (void const *arg) {
    while (1) { // Loop forever
        counter1++; // Increment counter1
    }
}

void job2 (void const *arg) {
    while (1) { // Loop forever
        counter2++; // Increment counter2
    }
}

osThreadDef (job1, osPriorityNormal, 1, 0); // Define thread for job1
osThreadDef (job2, osPriorityNormal, 1, 0); // Define thread for job2

int main (void) { // main() runs as thread
    osKernelInitialize (); // Initialize RTX

    osThreadCreate (osThread (job1), NULL); // Create and start job1
    osThreadCreate (osThread (job2), NULL); // Create and start job2

    osKernelStart (); // Start RTX kernel

    while (1) {
        osThreadYield (); // Next thread
    }
}
```
Preemptive Thread Switching

Threads with the same priority need a Round-Robin timeout or an explicit call of the `osDelay()` function to execute other threads. In the example above, if `job2` has a higher priority than `job1`, execution of `job2` starts instantly. `Job2` preempts execution of `job1` (this is a very fast task switch requiring a few ms only).

**Start job2 with Higher Thread Priority**

```c
: osThreadDef (osThread (job2), osPriorityAboveNormal, 1, 0);
:
}
```

Timer Management

Timer Management functions allow creating and controlling of timers and callback functions in the system. A callback function is called when a period expires whereby both one-shot and periodic timers are possible. A timer can be started, restarted, or stopped.

Timers are handled in the thread `osTimerThread()`. Callback functions run under control of this thread and may use other CMSIS-RTOS API calls.

The figure below shows the behaviour of a periodic timer. For one-shot timers, the timer stops after execution of the callback function.

![Behavior of a Periodic Timer](image)

RTX allows creating one-shot timers and timers that execute periodically.
One-Shot and Periodic Timers

#include "cmsis_os.h"

void Timer1_Callback (void const *arg); // Timer callback
void Timer2_Callback (void const *arg); // Prototype functions

osTimerDef (Timer1, Timer1_Callback); // Define timers
osTimerDef (Timer2, Timer2_Callback);

uint32_t exec1; // Callback function arguments
uint32_t exec2;

void TimerCreate_example (void) {
    osTimerId id1; // Timer identifiers
    osTimerId id2;

    // Create one-shot timer
    exec1 = 1;
    id1 = osTimerCreate (osTimer(Timer1), osTimerOnce, &exec1);
    if (id1 != NULL) {
        // One-shot timer created
    }

    // Create periodic timer
    exec2 = 2;
    id2 = osTimerCreate (osTimer(Timer2), osTimerPeriodic, &exec2);
    if (id2 != NULL) {
        // Periodic timer created
    }
}

Signal Management

Signal Management functions allow you to control or wait for signal flags. Each thread has assigned signal flags.
Mutex Management

Mutex Management functions synchronize the execution of threads and protect access to a shared resource, for example, a shared memory image.

The CMSIS-RTOS Mutex template provides function bodies you can use to add your code.

In the **Project** window, right click a group, select **Add New Item to Group**, choose **User Code Template**, and select **CMSIS-RTOS Mutex**.

Semaphore Management

Semaphore Management functions manage and protect access to shared resources. For example, a Semaphore can manage the access to a group of identical peripherals. Although they have a simple set of calls to the operating system, they are the classic solution in preventing race conditions. However, they do not resolve resource deadlocks. RTX ensures that atomic operations used with semaphores are not interrupted.

The number of available resources is specified as parameter of the `osSemaphoreCreate()` function. Each time a Semaphore token is obtained with `osSemaphoreWait()`, the semaphore count is decremented. When the semaphore count is 0, no Semaphore token can be obtained. Semaphores are released with `osSemaphoreRelease();` this function increments the semaphore count.

The example creates and initializes a Semaphore object to manage access to shared resources. The parameter `count` specifies the number of available resources. The `count` value 1 creates a binary semaphore.
Thread management using one semaphore

```c
#include "cmsis_os.h" // CMSIS-RTOS RTX header file

osThreadId tid_thread1; // ID for thread 1
osThreadId tid_thread2; // ID for thread 2
osSemaphoreId semID; // Semaphore ID
osSemaphoreDef (semaphore); // Semaphore definition

// Thread 1 - High Priority - Active every 3ms
void thread1 (void const *argument) {
  int32_t val;
  while (1) {
    osDelay(3); // Pass control for 3ms
    val = osSemaphoreWait (semID, 1); // Wait 1ms for free token
    if (val > 0) { // If free token acquired
      // do your job
      osSemaphoreRelease (semID); // Return token to semaphore
    }
  }
}

// Thread 2 - Normal Priority -
// Looks for a free semaphore and uses resources whenever available
void thread2 (void const *argument) {
  while (1) {
    osSemaphoreWait (semID, osWaitForever); // Wait for free semaphore
    osSemaphoreRelease (semID); // Return token to semaphore
  }
}

// Thread definitions
osThreadDef (thread1, osPriorityHigh, 1, 0);
osThreadDef (thread2, osPriorityNormal, 1, 0);

void StartApplication (void) {
  semID = osSemaphoreCreate (osSemaphore(semaphore), 1);
  tid_thread1 = osThreadCreate (osThread(thread1), NULL);
  tid_thread2 = osThreadCreate (osThread(thread2), NULL);
}
```

The CMSIS-RTOS Semaphore template provides function bodies you can use to add your code.

 Fet In the Project window, right click a group, select Add New Item to Group, choose User Code Template and select CMSIS-RTOS Semaphore.
Memory Pool Management

The Memory Pool Management provides thread-safe and fully reentrant allocation functions for fixed sized memory pools. These functions have a deterministic execution time that is independent of the pool usage. Built-in memory allocation routines enable you to use the system memory dynamically by creating memory pools and use fixed sized blocks from the memory pool. The memory pool needs a proper initialization to the size of the object.

The CMSIS-RTOS Memory Pool template provides function bodies you can use to add your code.

In the Project window, right click a group, select Add New Item to Group, choose User Code Template, and select CMSIS-RTOS Memory Pool.

Message Queue Management

Message Queue Management functions allow to control, send, receive, or wait for messages. A message can be an integer or pointer value that is sent to a thread or interrupt service routine.

The CMSIS-RTOS Message Queue template provides function bodies you can use to add your code.

In the Project window, right-click a group, select Add New Item to Group, choose User Code Template, and select CMSIS-RTOS Message Queue.
Mail Queue Management

The Mail Queue Management function group allows controlling, sending, receiving, or waiting for mail. A mail is a memory block that is sent to a thread or to an interrupt service routine.

The CMSIS-RTOS Mail Queue template provides function bodies you can use to add your code.

In the Project window, right click a group, select Add New Item to Group, choose User Code Template, and select CMSIS-RTOS Mail Queue.
CMSIS-DSP

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

When enabling the software component ::CMSIS:DSP in the dialog Manage Run-Time Environment, the optimum library for the selected device is included into the project.

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] = {
  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.
Create Applications

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

NOTE
The example code in this section works for the STM32F4-Discovery evaluation board (populated with STM32F407VG). Adapt the code and port pin configurations when using another starter kit or board.

The tutorial creates the project *Blinky* in the two basic design concepts:

- **RTOS Design using CMSIS-RTOS RTX.**
- **Infinite Loop Design for bare-metal systems without RTOS Kernel.**

For the project *Blinky*, you will create the following application files:

- **main.c** This file contains the `main()` function that initializes the RTOS kernel, the peripherals, and starts thread execution.
- **LED.c** The file contains functions to initialize and control the GPIO port and the thread function `blink_LED()`. The `LED_Initialize()` function initializes the GPIO port pin. The functions `LED_On()` and `LED_Off()` control the port pin that interface to the LED.
- **LED.h** The header file contains the function prototypes for the functions in **LED.c** and is included into the file **main.c**.

In addition, you will configure the system clock and the CMSIS-RTOS RTX.
**Blinky with CMSIS-RTOS RTX**

The section explains the creation of the project using the following steps:

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

- **Configure the Device Clock Frequency**: configure the system clock frequency for device and the CMSIS-RTOS RTX kernel.

- **Create the Source Code Files**: add and create the application files.

- **Build the Application Image**: compile and link the application code for download to on-chip Flash memory of a microcontroller device.

Using the Debugger on page 52 guides you through the steps to connect your evaluation board to the PC and to download the application to the target hardware.

**Setup the Project**

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

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

Next, the dialog **Select Device for Target** opens.

Select a device and click **OK**.

The device selection defines essential tool settings such as compiler controls, the memory layout for the linker, and the Flash programming algorithms.
The dialog **Manage Run-Time Environment** opens and shows the software components that are installed and available for the selected device.

![Manage Run-Time Environment dialog](image)

- Expand **::CMSIS:RTOS** and enable **:Keil RTX**.

  ![Validation Output](image)

  The **Validation Output** field shows dependencies to other software components. In this case, the component **::Device:Startup** is required.

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

- Click **Resolve**.

  This resolves all dependencies and enables other required software components (here, **::CMSIS:Core** and **::Device:Startup**).

- Click **OK**.

  The selected software components are included into the project together with the startup file, the RTX configuration file, and the 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.
Configure the Device Clock Frequency

The system or core clock is defined in the `system_<device>.c` file. The core clock also is the input clock frequency for the RTOS Kernel Timer and, therefore, the RTX configuration file needs to match this setting.

The clock configuration for an application depends on various factors such as the clock source (XTAL or on-chip oscillator), and the requirements for memory and peripherals. Silicon vendors provide the device-specific file `system_<device>.c` and therefore it is required to read the related documentation.

**TIP:** Open the reference manual from the **Books** window for detailed information about the microcontroller clock system.

The STM32F4-Discovery Kit runs with an external 8MHz XTAL. However, the PLL generates a core clock frequency of 168MHz. Make the following modifications:

- Set the PLL parameters in the file `system_stm32f4xx.c`.

To edit the file `system_stm32f4xx.c`, expand the group **Device** in the **Project** window, double-click on the file name, and modify the code as shown below.

**Set PLL Parameters in system_stm32f4xx.c**

```c
/*PLL Parameters**********************************/
/* PLL_VCO = (HSE_VALUE or HSI_VALUE / PLL_M) * PLL_N */
#define PLL_M 8
#define PLL_N 336

/* SYSCLK = PLL_VCO / PLL_P */
#define PLL_P 2

/* USB OTG FS, SDIO and RNG Clock = PLL_VCO / PLLQ */
#define PLL_Q 7
```
Customize the CMSIS-RTOS RTX Kernel

In the Project window, expand the group CMSIS, open the file `RTX_Conf_CM.c`, and click the tab Configuration Wizard at the bottom of the editor.

Expand RTX Kernel Timer Tick Configuration and set the Timer clock value to match the core clock value.

TIP: You may copy the compiler define settings and `system_<device>.c` from example projects. Right click on the filename in the editor and use Open Containing Folder to locate the file.
Create the Source Code Files

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

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 CMSIS-RTOS ‘main’ function and click Add.

This adds the file main.c to the project group Source Group 1. Now, add user code to this file.

Right-click on a blank line in the file main.c and select Insert ‘#include files’. Include the header file cmsis_os.h for the CMSIS-RTOS RTX.

Then, add the code below to create a function blink_LED that blinks LEDs on the evaluation kit. Define blink_LED as an RTOS thread using osThreadDef() and start it with osThreadCreate().
Code for main.c

```c
/*---------------------------------------------
 * CMSIS-RTOS 'main' function template
 *---------------------------------------------*/

#define osObjectsPublic // Define objects in main module
#include "osObjects.h"  // RTOS object definitions
#include "cmsis_os.h"   // ARM::CMSIS:RTOS:Keil RTX
#include "LED.h"        // Initialize and set GPIO Port

/*
 * main: initialize and start the system
 */
int main (void) {
    osKernelInitialize (); // Initialize CMSIS-RTOS
    // initialize peripherals here
    LED_Initialize ();     // Initialize LEDs

    // create 'thread' functions that start executing,
    // example: tid_name = osThreadCreate (osThread(name), NULL);
    Init_BlinkyThread ();  // Start Blinky thread
    osKernelStart ();      // Start thread execution

    while (1);
}  
```

Create an empty C-file named LED.c using the dialog Add New Item to Group and add the code to initialize and access the GPIO port pins that control the LEDs.

Tips: Right-click in the editor and select Insert '#include files to include the CMSIS-CORE device header file stm32f4xx.h.
Code for LED.c

```c
#include "stm32f4xx.h" // Device header
#include "cmsis_os.h" // RTOS: Keil RTX header

void blink_LED (void const *argument); // Prototype function

osThreadDef (blink_LED, osPriorityNormal, 1, 0); // Define blinky thread

void LED_Initialize (void) {
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN; // Enable Port D clock
    GPIOD->MODER |= GPIO_MODER_MODER13_0; // Port D.13 output
}

void LED_On (void) {
    GPIOD->BSRR = (1<<13); // LED on: set Port
}

void LED_Off (void) {
    GPIOD->BSRRH = (1<<13); // LED off: clear Port
}

void blink_LED(void const *argument) {
    for (;;) {
        LED_On (); // Switch LED on
        osDelay (500); // Delay 500 ms
        LED_Off (); // Switch off
        osDelay (500); // Delay 500 ms
    }
}

void Init_BlinkyThread (void) {
    osThreadCreate (osThread(blink_LED), NULL); // Create thread
}
```

Create an empty header file named LED.h using the dialog Add New Item to Group and define the function prototypes of LED.c.

Code for LED.h

```c
#include "stm32f4xx.h" // Device header
#include "cmsis_os.h" // RTOS: Keil RTX header

void LED_Initialize (void); // Initialize GPIO
void LED_On (void); // Switch Pin on
void LED_Off (void); // Switch Pin off

void blink_LED (void const *argument); // Blink LEDs in a thread
void Init_BlinkyThread (void); // Initialize thread
```
Build the Application Image

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.

![Build Output Image]

The section **Using the Debugger** on page 52 guides you through the steps to connect your evaluation board to the workstation and to download the application to the target hardware.

**TIP:** You may verify the correct clock and RTOS configuration of the target hardware by checking the one-second interval of the LED.
Blinky with Infinite Loop Design

Based on the previous example, we create a Blinky application with the infinite loop design and without using CMSIS-RTOS RTX functions. The project contains the user code files:

main.c This file contains the main() function, the function Systick_Init() to initialize the System Tick Timer and its handler function SysTick_Handler(). The function Delay() waits for a certain time.

LED.c The file contains functions to initialize the GPIO port pin and to set the port pin on or off. The LED_Reset() function initializes the GPIO port pin. The functions LED_On() and LED_Off() enable or disable the port pin.

LED.h The header file contains the function prototypes created in LED.c and must be included into the file main.c.

Open the Manage Run-Time Environment and deselect the software component ::CMSIS:RTOS:Keil RTX.

Open the file main.c and add the code to initialize the System Tick Timer, write the System Tick Timer Interrupt Handler, and the delay function.

```c
#include "LED.h" // Initialize and set GPIO Port
#include "stm32f4xx.h" // Device header

int32_t volatile msTicks = 0; // Interval counter in ms

// Set the SysTick interrupt interval to 1ms
void Systick_Init (void) {
  if (SysTick_Config (SystemCoreClock / 1000)) {
    // handle error
  }
}

// SysTick Interrupt Handler function called automatically
void Systick_Handler (void) {
  msTicks++; // Increment counter
}

// Wait until msTick reaches 0
void Delay (void) {
  while (msTicks < 499); // Wait 500ms
  msTicks = 0; // Reset counter
}
```
```c
int main (void) {
    // initialize peripherals here
    LED_Initialize (); // Initialize LEDs
    SysTick_Init ();   // Initialize SysTick Timer

    while (1) {
        LED_On ();       // Switch on
        Delay ();        // Delay
        LED_Off ();      // Switch off
        Delay ();        // Delay
    }
}
```

Open the file `LED.c` and remove unnecessary functions. The code should look like this.

```c
#include "stm32f4xx.h" // Device header

// Initialize GPIO Prot
void LED_Initialize (void) {
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN; // Enable Port D clock
    GPIOD->MODER |= GPIO_MODER_MODER13_0; // Port D.13 output
}

/* Turn LED on */
void LED_On (void) {
    GPIOD->BSRRL = (1<<13); // LED on: set Port
}

/* Turn LED off */
void LED_Off (void) {
    GPIOD->BSRRH = (1<<13); // LED off: clear Port
}
```

Open the file `LED.h` and modify the code.

```c
/* file: LED.h
 *------------------------------------------------------------------------*/
void LED_Initialize (void); // Initialize LED Port Pins
void LED_On (void);        // Set LED on
void LED_Off (void);       // Set LED off
```
Build the Application Image

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

The section Using the Debugger on page 52 guides you through the steps to connect your evaluation board to the PC and to download the application to the target hardware.

**TIP:** You may verify the correct clock configuration of the target hardware by checking the one-second interval of the LED.
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 PC sampling, data trace, exceptions including interrupts, and instrumentation trace are available in most devices. Devices integrate instruction trace using ETM, ETB, or MTB to enable analysis of the program execution. Refer to www.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 ULINK2 and ULINK-ME Debug adapters interface to JTAG/SWD debug connectors and support trace with the Serial Wire Output (SWO). The ULINKpro 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 www.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. In addition, several proprietary interfaces that offer a similar technology are supported.

- MDK supports third-party debug solutions such as Segger J-Link or J-Trace. Some starter kit boards provide the J-Link Lite technology as an on-board solution.
Using the Debugger

Next, you will debug the *Blinky* application created in the previous chapter on hardware. You need to configure the debug connection and Flash programming utility.

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.

Configure Flash programming options.

Switch to the dialog **Utilities** and enable **Use Debug Driver**.

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

Program the application into Flash memory.

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 statement that executes next (current PC location).
Command Window

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

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 breakpoints, bookmarks, and for the next execution statement.
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 window Debug – Breakpoints.
- Entering commands in the Command window.
- Using the context menu of the Disassembly window or editor.

Breakpoints Window

You can define sophisticated 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. A compare (==) operator may be used 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. Tasks are shown for applications that use the CMSIS-RTOS RTX.
Register Window

The Register window shows the content of 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-click on the value of a register, or press 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 µVision 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 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.

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 and Cortex-M4 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** available on high-end microcontrollers and has the high bandwidth required for Instruction Trace.

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

- Cortex-M3 and Cortex-M4 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.

<table>
<thead>
<tr>
<th>Feature</th>
<th>ULINKpro</th>
<th>ULINKpro-D</th>
<th>ULINK2</th>
<th>ST-Link v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Management</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Maximum SWO clock frequency</td>
<td>200 MHz</td>
<td>200 MHz</td>
<td>3.75 MHz</td>
<td>2 MHz</td>
</tr>
<tr>
<td>4-Pin Trace Output for Streaming</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Embedded Trace Buffer (ETB)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Micro Trace Buffer (MTB)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

**Trace with Serial Wire Output**

To use the Serial Wire Trace Output (SWO), use the following steps:

1. Click **Options for Target** on the toolbar and select the **Debug** tab. Verify that you have selected and enabled the correct **debug adapter**.

2. Click the **Settings** button. On the **Debug** dialog, select the debug **Port** and set the **Max Clock** frequency for communicating with the debug unit of the device.
Click the **Trace** tab. Ensure the **Core Clock** has the right setting. Set **Trace Enable** and select the **Trace Events** you want to monitor.

Enable **ITM Stimulus Port 0** for printf-style debugging.

Enable **ITM Stimulus Port 31** to view RTOS Events.

![Cortex-M Target Driver Setup](image)

**NOTE**

When many trace features are enabled, the Serial Wire Output communication can overflow. The μVision Status Bar displays such connection errors.

**The ULINKpro Debug/Trace Adapter has high trace bandwidth and such communication overflows are rare. Enable only the trace features that are currently required to avoid overflows in the trace communication.**
**Trace Exceptions**

The *Exception Trace* window displays statistical data about traced 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.

![Trace Exceptions](image)

To retrieve data in the *Trace Exceptions* window:

- Set *Trace Enable* in the Debug Settings Trace dialog as described above.
- Enable *EXCTR: Exception Tracing*.
- Set *Timestamps Enable*.

*NOTE*

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

The Logic Analyzer Window displays changes of variable values over time. Up to four variables can be monitored. 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 Location 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 use the **Debug (printf) Viewer**, add the following `fputc` function that uses the CMSIS function `ITM_SendChar` to your source code.

```c
#include <stdio.h>
#include "stm32f4xx.h" // Device header

struct __FILE { int handle; };
FILE __stdout;
FILE __stdin;
int fputc(int c, FILE *f) {
    ITM_SendChar(c);
    return (c);
}
```

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

```c
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**.
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.
- **SLEEP_CNT**: 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. In addition also instruction trace is possible.

The ULINKpro Debug/Trace Adapter supports this parallel 4-Pin trace output (also called ETM Trace) allows 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 no available on the microcontroller or target hardware. As an alternative, an on-chip **Trace Buffer** can be used that supports the Trace Data Window.
RTOS Debugging

When using a Real-time Operating System for an application, RTOS-aware debugging is important. MDK supports various real-time operating systems, including CMSIS-RTOS RTX.

System and Thread Viewer

The System and Thread Viewer window shows system state information. Open this window with the menu Debug – OS Support – System and Thread Viewer.
Event Viewer

The Event Viewer uses the ITM stimulus port 31 to transmit debug and thread information and requires Trace setup as described on page 59. It shows the thread-switching of CMSIS-RTOS RTX over time. Open this window with the menu Debug – OS Support – Event Viewer.

To retrieve data in the Event Viewer window:

- Set Trace Enable in the Debug Settings Trace dialog as described above.
- Enable ITM Stimulus Port 31.
- Set Timestamps Enable.
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. MDK also works with middleware available from several other vendors.

Refer to [www.keil.com/dd2/pack](http://www.keil.com/dd2/pack) for a list of public Software Packs.

MDK-Professional provides a Software Pack that 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 [www.keil.com/mdk5/middleware](http://www.keil.com/mdk5/middleware) for more information about the middleware provided as part of MDK-Professional.

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

- MDK-Professional 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 drivers. Refer to **Driver Components** on page 77 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
MDK-Professional Middleware. 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-Professional 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 78 that exemplifies a combination of several middleware components.

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

The Network Component uses TCP/IP communication protocols and contains support for services, protocol sockets, and physical communication interfaces.

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 for page data storage. Both servers support dynamic page content using 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 support 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 Standard Socket 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 certain Drivers 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.
**File System Component**

The **File System Component** allows your embedded applications to create, save, read, and modify files in storage devices such as RAM, Flash memory, Memory Cards, or USB 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 8.3 file names and long file names with up to 255 characters.

To access the physical media, for example NAND and NOR Flash chips, or memory cards using MCI or SPI, **Drivers** have to be present.
USB Device Component

The **USB Device component** implements a USB device interface 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 be used for simple data exchange.

- **Mass Storage Class** (MSC) is used for file exchange (for example an USB stick).

- **Communication Device Class** (CDC) implements a virtual serial port.

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

**Composite USB devices** implement multiple device classes.

This component requires a **USB Device 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.
USB Host Component

The **USB Host Component** implements a USB Host interface and supports Mass Storage and Human Interface Device classes.

- **HID** connects to any HID class equipment.
- **MSC** connects any USB memory stick to your device.

This component requires a **USB Host Driver** to be present. Depending on the application, it must comply with the USB 1.1 (Full-Speed USB) and/or the USB 2.0 (High-Speed USB) specification.
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 may 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 or a flexible **interface template** that may be adapted to 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
Driver Components

Device-specific **drivers** provide the interface between the middleware and the microcontroller peripherals. These drivers are not limited to the MDK-Professional 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 [www.keil.com/dd2](http://www.keil.com/dd2) lists drivers included in the Software Pack for the device.

The middleware components have various configuration files that connect to these drivers. For most devices, the **RTE_Device.h** file configures the drivers to the actual pin connection of the microcontroller device.

The middleware 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 [www.keil.com/cmsis/driver](http://www.keil.com/cmsis/driver) for detailed information about the API interface of these CMSIS drivers.

**NOTE**

*Application Note 250: Creating a Software Pack with a New Peripheral Driver* available on [www.keil.com/appnotes](http://www.keil.com/appnotes) explains how to create a new peripheral driver that does not exist in a Software Pack.

### FTP Server Example

The FTP Server Example is one of the reference application samples that shows a combination of several middleware components. Refer to **Verify Installation using Example Projects** on page 10 for more information the various example projects available.

Using a FTP Server, you can exchange and manipulate files over a TCP/IP network. The Middleware documentation has more details on 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 diagram represents the software components that are used from the MDK-Professional Middleware to create the FTP Server example.
As explained before, **Drivers** provide the interface between the microcontroller peripherals and the MDK-Professional Middleware.

The FTP example selects the components shown below in the **Manage Run-Time Environment** dialog.
Using Middleware

The chapter explains how to create an application that uses MDK-Professional Middleware components. For more information on the topics in this chapter, refer to the MDK-Professional Middleware User’s Guide that has sections for every component describing:

- **Example projects** outline key product features of the software components, are tested, implemented, and proven on several evaluation boards and can be used as reference application or starting point for your development.

- **Resource Requirements** explain for every software component the thread and stack resources for CMSIS-RTOS and memory footprint.

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

- **Reference** documents the files of the component and each API function.

The generic steps to use the various middleware components are:

- **Add Software Components**: Select in the Manage Run-Time Environment dialog the software components that are required in 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 with physical I/O pins of the microcontroller.

Adjust System Resources: The middleware components use RTOS, memory, and stack resources and this may imply configurations, for example to CMSIS-RTOS RTX.

Implement Application Features: Use the API functions of the middleware components to implement the application specific behaviour. Code templates help you 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 on page 52 to download the image to target hardware.

Verify and Debug: Test utilities along with the debug and trace features described in the chapter Debug Applications on page 50 allows testing of the application.

USB HID Example

While above steps are generic and apply to all components of the MDK-Professional Middleware, the USB HID example described in the following sections shows these steps in practice. This example creates an 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 the LEDs on the development board.

The USB HID example described in the following sections uses the STM32F4-Discovery Kit populated with a STM32F407VG microcontroller. It is based on the project Blinky with CMSIS-RTOS RTX on page 40 along with the source files main.c, LED.c, LED.h, and the configuration files.

NOTE
You must adapt the code and port pin configurations when using this example on other starter kits or evaluation boards.
Add Software Components

To create the USB HID Device example, start with the project Blinky with CMSIS-RTOS RTX that is described on page 40.

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

From the USB Device Component described on page 74:

- 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.

From the Driver Components described on page 77:

- Select from ::Drivers: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.

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

The picture below shows the Manage Run-Time Environment dialog after adding these components.
### Manage Run-Time Environment

<table>
<thead>
<tr>
<th>Software Component</th>
<th>Sel.</th>
<th>Variant</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Board Support</td>
<td></td>
<td></td>
<td>1.0.0</td>
<td>ST STM32F4 Discovery Board</td>
</tr>
<tr>
<td>CMSIS</td>
<td></td>
<td></td>
<td>3.20.0</td>
<td>CMSIS CORE for Cortex-M, SC000, and SC300</td>
</tr>
<tr>
<td>CORE</td>
<td></td>
<td></td>
<td>1.4.1</td>
<td>CMSIS DSP Library for Cortex-M, SC000, and SC300</td>
</tr>
<tr>
<td>DSP</td>
<td></td>
<td></td>
<td>1.0</td>
<td>CMSIS RTOS API for Cortex-M, SC000, and SC300</td>
</tr>
<tr>
<td>RTOS (API)</td>
<td></td>
<td></td>
<td>4.74.0</td>
<td>CMSIS-RTOS RTX implementation for Cortex-M, SC000, and SC300</td>
</tr>
<tr>
<td>Device</td>
<td></td>
<td></td>
<td></td>
<td>Startup, System Setup</td>
</tr>
<tr>
<td>GPIO</td>
<td></td>
<td></td>
<td>1.0.0</td>
<td>GPIO driver used by RTE Drivers for STM32F4 Series</td>
</tr>
<tr>
<td>Startup</td>
<td></td>
<td></td>
<td>1.3.0</td>
<td>System Startup for STMicroelectronics STM32F4 Series</td>
</tr>
<tr>
<td>ST Lib</td>
<td></td>
<td></td>
<td></td>
<td>Unified Device Drivers</td>
</tr>
<tr>
<td>Drivers</td>
<td></td>
<td></td>
<td></td>
<td>File System</td>
</tr>
<tr>
<td>USB Device (API)</td>
<td></td>
<td></td>
<td>1.10</td>
<td>USB Device Driver API for Cortex-M</td>
</tr>
<tr>
<td>Graphics</td>
<td></td>
<td></td>
<td></td>
<td>MDK Pro 5.0.4 File access on various storage devices</td>
</tr>
<tr>
<td>MDK-Pro</td>
<td></td>
<td></td>
<td>5.2.2.1</td>
<td>MDK Pro 5.2.2.1 User Interface on graphical LCD displays</td>
</tr>
<tr>
<td>MDK-Pro</td>
<td></td>
<td></td>
<td>5.0.4</td>
<td>MDK-Pro 5.0.4 IP Networking using Ethernet or Serial protocols</td>
</tr>
<tr>
<td>Network</td>
<td></td>
<td></td>
<td></td>
<td>MDK-Pro 5.0.4 USB Communication with various device classes</td>
</tr>
<tr>
<td>USB</td>
<td></td>
<td></td>
<td></td>
<td>MDK-Pro 5.0.4 USB Communication with various device classes</td>
</tr>
<tr>
<td>CORE</td>
<td></td>
<td></td>
<td>5.0.4</td>
<td>USB Core for Cortex-M</td>
</tr>
<tr>
<td>Device</td>
<td></td>
<td></td>
<td>5.0.4</td>
<td>USB Device</td>
</tr>
<tr>
<td>Host</td>
<td></td>
<td></td>
<td>5.0.4</td>
<td>USB Host</td>
</tr>
<tr>
<td>Device</td>
<td></td>
<td></td>
<td></td>
<td>USB Device Classes</td>
</tr>
<tr>
<td>ADC</td>
<td></td>
<td></td>
<td>5.0.4</td>
<td>USB Device Audio Device Class (ADC)</td>
</tr>
<tr>
<td>CDC</td>
<td></td>
<td></td>
<td>5.0.4</td>
<td>USB Device Communication Device Class (CDC)</td>
</tr>
<tr>
<td>Custom Class</td>
<td></td>
<td></td>
<td>5.0.4</td>
<td>USB Device Custom Class</td>
</tr>
<tr>
<td>HID</td>
<td></td>
<td></td>
<td>5.0.4</td>
<td>USB Device Human Interface Device (HID) Class</td>
</tr>
<tr>
<td>MSC</td>
<td></td>
<td></td>
<td>5.0.4</td>
<td>USB Device Mass Storage Class (MSC)</td>
</tr>
<tr>
<td>Host</td>
<td></td>
<td></td>
<td></td>
<td>USB Host Classes</td>
</tr>
</tbody>
</table>
Configure Middleware

Every MDK-Professional Middleware component has a set of configuration files that adjusts application specific parameters and determines the driver interfaces. These configuration files are accessed in the Project window in the component class group and 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 that is used as device interface. For microcontrollers with only one USB Controller the number is ‘0’. Refer to “Driver Components” on page 77 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 **Vendor ID (VID)** to a private VID. The USB Implementer’s Forum ([http://www.usb.org/developers/vendor/](http://www.usb.org/developers/vendor/)) provides more information on how to apply for a valid Vendor ID.

Every device needs a unique **Product ID**. Together with the VID, it is used by the Host computer's operating system to find a driver for your device.

The **Manufacturer** and the **Product String** can be set to identify the USB Device in PC operating systems.

The file *USBD_Cfg_HID_0.h* contains Device Class specific Endpoint Settings Numbers. For this 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. This `RTE_Device.h` file typically requires configuration of the actual peripheral interfaces that are 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.

The USB HID Device example requires the following settings:

- The STM32F4-Discovery Kit has an external 8 MHz oscillator. Under Clock Configuration, change the value of High-speed External Clock to 8000000.

- Enable USB OTG Full-Speed, expand this section, and make sure to select only the Device [Driver_USBD0]. This `Driver_USBD0` identifies the control struct of the peripheral interface and matches with the setting Connect to Hardware via Driver_USBD# in the step Configure Middleware.

- You may disable Endpoints 2 and 3 to reduce the memory footprint, since the HID device requires a single Endpoint only.

NOTE
The Clock Configuration values in the `RTE_Device.h` file are important for the middleware components and must match the settings in the `system_stm32f4xx.c` file.
Adjust System Resources

Every middleware component has certain memory and RTOS resources requirements. The section “Resource Requirements” in the MDK-Professional Middleware User’s Guide documents the requirements for each component.

Most middleware components use the CMSIS-RTOS so it is important that the RTOS is configured to support the requirements.

For CMSIS-RTOS RTX, the RTX_Cnf_CM.c file configures thread and stack settings. Refer to “CMSIS-RTOS RTX Configuration” on page 25 for more information.

For the USB HID Device example, the following settings apply:

- The ::USB:Device component requires one thread (called USBDn_CoreThread) and a user-provided stack of 512 bytes.
- The ::USB:Device:HID component also requires one thread (called USBD_HIDn_Thread) and a user-provided stack of 512 bytes.

Reflect these requirements with the settings in the RTX_Cnf_CM.c file:

- **Number of concurrent running threads**: 6 (default) is enough to run the two threads of the USB Device component concurrently. Adjust this setting if the user application executes additional threads.
- **Default Thread stack size [bytes]**: This setting is not important as the USB component runs on user-provided stack.
- **Main Thread stack size [bytes]**: 512. Stack is required for the API calls that initialize the USB Device component.
- **Number of threads with user-provided stack size:** 2. Specifies the two threads (for ::USB:Device and ::USB:Device:HID) with a user-provided stack.

- **Total stack size [bytes] for threads with user-provided stack size:** 1024. Specifies the total stack size of the two threads.

- **The Timer Clock value [Hz]** 168000000. Matches the system clock.

### Implement Application Features

Now, create the code that implements the application specific features. This includes modifications to the files `main.c`, `LED.c`, and `LED.h` that were initially created for the project “Blinky with CMSIS-RTOS RTX”.

The middleware provides **User Code Templates** as starting point for the application software. For the USB HID Device example add the user code template ::USB:Device:HID with the default file name `USBD_User_HID_0.c`.

ู่ In the **Project** window, right-click **Source Group 1** and open the dialog **Add New Item to Group**. Select the ::USB:Device:HID user code template and click **Add**.
To connect the PC USB application to the microcontroller device, the function `USBD_HID0_SetReport` needs modifications. This function handles data coming from the USB Host, which are created in our example with the utility program `HIDClient.exe`.

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

```c
#include "LED.h" // access functions to LEDs

bool USBD_HID0_SetReport (uint8_t rtype, uint8_t req, uint8_t rid, const uint8_t *buf, int32_t len) {
  uint8_t i;
  switch (rtype) {
    case HID_REPORT_OUTPUT:
      for (i = 0; i < 4; i++) {
        if (*buf & (1 << i))
          LED_On (i);
        else
          LED_Off (i);
      }
      break;
    case HID_REPORT_FEATURE:
      break;
  }
  return true;
}
```
Expand the functions in the file LED.c to control several LEDs on the STM32F4-Discovery Kit and remove the thread that blinks the LED as it is no longer required.

Open the file LED.c in the editor and modify the code as shown below.

```c
#include "stm32f4xx.h" // Device header
#include "LED.h" // Initialize and set GPIO Port

void LED_Initialize(void) {
  RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN; // Enable Port D clock
  GPIOD->MODER |= GPIO_MODER_MODER12_0; // Port D.12 output
  GPIOD->MODER |= GPIO_MODER_MODER13_0; // Port D.13 output
  GPIOD->MODER |= GPIO_MODER_MODER14_0; // Port D.14 output
  GPIOD->MODER |= GPIO_MODER_MODER15_0; // Port D.15 output
}

const int Pin_LED[] = { 12, 13, 14, 15 }; // LED Port mapping

void LED_On(unsigned int num) {
  GPIOD->BSRRL = (1<<Pin_LED[num]); // LED on: set Port
}

void LED_Off(unsigned int num) {
  GPIOD->BSRRH = (1<<Pin_LED[num]); // LED off: clear Port
}
```

Open the file LED.h in the editor and modify it to coincide with the changes to LED.c. The functions LED_On and LED_Off now have a parameter.

```c
#include "LED.h" // Initialize and set GPIO Port

void LED_Init(void);
void LED_On(unsigned int num);
void LED_Off(unsigned int num);
```

Change the file main.c as shown below. Instead of starting the thread that blinks the LED, add code to initialize and start the USB Device Component. Refer to the middleware user’s guide for further details.

```c
#include "cmsis_os.h" // CMSIS-RTOS
#include "rl_usb.h" // Middleware USB Component
#include "LED.h" // Initialize and set GPIO Port
```
/*
 * main: initialize and start the system
 */
int main (void) {
    osKernelInitialize (); // Initialize CMSIS-RTOS
    // initialize peripherals here
    LED_Initialize (); // Initialize LEDs
    USBD_Initialize (0); // USB Device 0 Initialization
    USBD_Connect (0); // USB Device 0 Connect
    osKernelStart (); // Start thread execution
    while (1);
}

Build and Download

Build the project and download it to the target as explained in chapters Create Applications on page 39 and Using the Debugger on page 50.

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”.

- Test the application by changing the Outputs (LEDs) checkboxes. The respective LEDs will 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.
Use debug windows to narrow down the problem. For example, use the Call Stack + Locals window to examine the value of local variables in the `USBD_User_HID_0.c` file. 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. Use therefore 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

A
Add New Item to Group.................. 45, 87
Adding Software Components ........... 19
Applications
Add Source Code .......................... 44
Build........................................ 47
Configure Device Clock Frequency .. 42
Create........................................ 39
Customize CMSIS-RTOS RTX......... 43
Debug.......................................... 51
Setup the Project........................ 40

B
Breakpoints
Access ....................................... 55
Command .................................... 55
Execution................................... 55
Build Output.................................. 12, 13, 47, 52

C
CMSIS........................................ 17
CORE .......................................... 18
DSP ............................................. 37
RTX........................................... 21
RTX API ...................................... 27
User Code Template ..................... 28
cmsis_os.h.................................. 23, 24, 44
CMSIS-DAP ................................... 51
Code Coverage ............................. 66
Compare memory areas .................. 57
CoreSight .................................. 59

D
Debug
Breakpoints .................................. 55
Breakpoints Window ..................... 55
Command Window ....................... 54
Connection .................................. 51
Disassembly Window .................... 54
Memory Window ............................ 57
Peripheral Registers .................... 58
Register Window .......................... 57
Stack and Locals Window ............. 56
Start Session ............................. 53
System Viewer Window ................. 58
Toolbar .................................... 53
Using Debugger........................... 52


Watch Window ............................ 56
Debug (printf) Viewer .................. 64
Debug tab ................................. 12, 52, 60
Device Database ......................... 9
device.h ..................................... 18
Documentation ............................ 16

E
Example Projects ......................... 10, 69, 78

F
File System
FAT ............................................. 73
Flash............................................ 73

G
Graphics Component
Anti-Aliasing ............................... 76
Bitmap Support ............................ 76
Demo ........................................ 76
Dialogs ...................................... 76
Display ...................................... 76
Fonts ........................................ 76
Joystick ..................................... 76
Touch Screen .............................. 76
User Interface ............................. 76
VNC Server ................................. 76
Widgets ...................................... 76
Window Manager ......................... 76

H
HIDClient.exe .............................. 90

I
Infinite Loop Design ..................... 21

L
Legacy Support ............................ 6

M
MDK
Core ........................................ 6
Core Install ................................ 8
Editions ..................................... 7
Installation Requirements ............ 8
Introduction ............................... 6
License Types ............................. 7
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Component</td>
<td>71</td>
</tr>
<tr>
<td>BSD</td>
<td>71</td>
</tr>
<tr>
<td>DNS Client</td>
<td>71</td>
</tr>
<tr>
<td>Ethernet</td>
<td>72</td>
</tr>
<tr>
<td>FTP</td>
<td>71</td>
</tr>
<tr>
<td>Modem</td>
<td>72</td>
</tr>
<tr>
<td>PPP</td>
<td>72</td>
</tr>
<tr>
<td>SLIP</td>
<td>72</td>
</tr>
<tr>
<td>SMTP Client</td>
<td>71</td>
</tr>
<tr>
<td>SNMP Agent</td>
<td>71</td>
</tr>
<tr>
<td>SNTP Client</td>
<td>71</td>
</tr>
<tr>
<td>TCP</td>
<td>71</td>
</tr>
<tr>
<td>Telnet Server</td>
<td>71</td>
</tr>
<tr>
<td>TFTP</td>
<td>71</td>
</tr>
<tr>
<td>UART</td>
<td>72</td>
</tr>
<tr>
<td>UDP</td>
<td>71</td>
</tr>
<tr>
<td>Web Server</td>
<td>71</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Middleware</td>
<td>69</td>
</tr>
<tr>
<td>Add Software Components</td>
<td>80, 81</td>
</tr>
<tr>
<td>Adjust System Resources</td>
<td>80, 86</td>
</tr>
<tr>
<td>Configure</td>
<td>80, 83</td>
</tr>
<tr>
<td>Configure Drivers</td>
<td>80, 85</td>
</tr>
<tr>
<td>Create an Application</td>
<td>80</td>
</tr>
<tr>
<td>Debug</td>
<td>81, 90</td>
</tr>
<tr>
<td>Drivers</td>
<td>77</td>
</tr>
<tr>
<td>Example projects</td>
<td>80</td>
</tr>
<tr>
<td>File System Component</td>
<td>73</td>
</tr>
<tr>
<td>FTP Server Example</td>
<td>78</td>
</tr>
<tr>
<td>Graphics Component</td>
<td>76</td>
</tr>
<tr>
<td>Implement Application Features</td>
<td>81, 87</td>
</tr>
<tr>
<td>Network Component</td>
<td>71</td>
</tr>
<tr>
<td>Resource Requirements</td>
<td>80</td>
</tr>
<tr>
<td>USB Device Component</td>
<td>74</td>
</tr>
<tr>
<td>USB HID Example</td>
<td>81</td>
</tr>
<tr>
<td>USB Host Component</td>
<td>75</td>
</tr>
<tr>
<td>Using</td>
<td>80</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Pool Management</td>
<td>35</td>
</tr>
<tr>
<td>Message Queue Management</td>
<td>35</td>
</tr>
<tr>
<td>Mutex Management</td>
<td>33</td>
</tr>
<tr>
<td>Preemptive Thread Switching</td>
<td>31</td>
</tr>
<tr>
<td>Semaphore Management</td>
<td>33</td>
</tr>
<tr>
<td>Signal Management</td>
<td>32</td>
</tr>
<tr>
<td>Single Thread Program</td>
<td>30</td>
</tr>
<tr>
<td>Thread Management</td>
<td>29</td>
</tr>
<tr>
<td>Timer Management</td>
<td>31</td>
</tr>
<tr>
<td>RTOS Debugging</td>
<td>67</td>
</tr>
<tr>
<td>Event Viewer</td>
<td>68</td>
</tr>
<tr>
<td>ITM Stimulus</td>
<td>68</td>
</tr>
<tr>
<td>System and Thread Viewer</td>
<td>67</td>
</tr>
<tr>
<td>RTOS Design</td>
<td>22</td>
</tr>
<tr>
<td>RTX</td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>25</td>
</tr>
<tr>
<td>Using RTX</td>
<td>22</td>
</tr>
<tr>
<td>RTX_Conf_CM.c</td>
<td>23, 86</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Packs</td>
<td>6</td>
</tr>
<tr>
<td>Install</td>
<td>9</td>
</tr>
<tr>
<td>Use</td>
<td>14</td>
</tr>
<tr>
<td>Verify Installation</td>
<td>10</td>
</tr>
<tr>
<td>Start/Stop Debug Session</td>
<td>13, 53, 91</td>
</tr>
<tr>
<td>Support</td>
<td>16</td>
</tr>
<tr>
<td>system_&lt;device&gt;.c</td>
<td>18, 27, 42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace</td>
<td>59</td>
</tr>
<tr>
<td>4-Pin Trace Output</td>
<td>59, 66</td>
</tr>
<tr>
<td>Data Watchpoints</td>
<td>59</td>
</tr>
<tr>
<td>Debug (printf) Viewer</td>
<td>64</td>
</tr>
<tr>
<td>ETB</td>
<td>59</td>
</tr>
<tr>
<td>Event Counters</td>
<td>65</td>
</tr>
<tr>
<td>Exception Trace</td>
<td>59</td>
</tr>
<tr>
<td>iputc function</td>
<td>64</td>
</tr>
<tr>
<td>Instruction Trace</td>
<td>59</td>
</tr>
<tr>
<td>Instrumented Trace</td>
<td>59</td>
</tr>
<tr>
<td>ITM Stimulus</td>
<td>61, 64</td>
</tr>
<tr>
<td>ITM_SendChar</td>
<td>64</td>
</tr>
<tr>
<td>Logic Analyzer</td>
<td>63</td>
</tr>
<tr>
<td>MTB</td>
<td>59</td>
</tr>
<tr>
<td>SWO</td>
<td>59, 60</td>
</tr>
<tr>
<td>TPIU</td>
<td>59</td>
</tr>
<tr>
<td>Trace Buffer</td>
<td>59</td>
</tr>
<tr>
<td>Trace Buffer</td>
<td>66</td>
</tr>
<tr>
<td>Trace Data Window</td>
<td>66</td>
</tr>
<tr>
<td>Trace Exceptions</td>
<td>62</td>
</tr>
</tbody>
</table>
ULINK ............................................... 51
ULINKpro ....................................... 61, 66
USB Device
  ADC ............................................. 74
  CDC ............................................. 74

Composite Device ............................. 74
HID ............................................... 74
MSC ............................................... 74
USB Host
  HID ............................................... 75
  MSC ............................................... 75
User Code Templates ....................... 44, 87