embOS

Real-Time Operating System

CPU & Compiler specifics for Cortex-M using Embedded Studio

Document: UM01061
Software Version: 5.20.0.0
Revision: 0
Date: March 19, 2025

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

The information written in this document is assumed to be accurate without guarantee. The information in this manual is subject to change for functional or performance improvements without notice. SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions in this document. SEGGER disclaims any warranties or conditions, express, implied or statutory for the fitness of the product for a particular purpose. It is your sole responsibility to evaluate the fitness of the product for any specific use.

Copyright notice

You may not extract portions of this manual or modify the PDF file in any way without the prior written permission of SEGGER. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

© 2010-2025 SEGGER Microcontroller GmbH, Monheim am Rhein / Germany

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel. +49 2173-99312-0
Fax. +49 2173-99312-28
E-mail: support@segger.com*
Internet: www.segger.com

*By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at https://www.segger.com/legal/privacy-policy/.

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please inform us and we will try to assist you as soon as possible. Contact us for further information on topics or functions that are not yet documented.

Print date: March 19, 2025

Software	Revision	Date	By	Description
5.20.0.0	0	250319	MC	New software version.
5.18.3.0	0	240711	MC	New software version.
5.18.0.0	0	221028	TS	New software version.
5.16.0.0	0	211210	MC	New software version.
5.14.0.0	0	210505	TS	New software version.
5.12.0.1	0	201208	TS	New software version.
5.12.0.0	0	201124	TS/MM	New software version. Chapter “ARMv8-M Stack limit register PSPLIM” added. Chapter “Libraries” updated with libraries for Cortex-M33.
5.10.1.0	0	200617	MC	New software version.
5.8.2.0	0	200107	TS	New software version.
5.8.1.0	0	191126	MC	New software version.
5.06	1	190709	MM	Minor modifications. Chapter “Libraries” updated with libraries for Cortex-M23.
5.06	0	190220	TS	New software version.
5.02a	0	180725	TS	New software version.
5.00	1	180518	TS	Chapter “CPU and compiler specifics” updated.
5.00	0	180503	MM	Chapter “embOS Thread Script” updated.
4.40	0	180111	TS/MC	Chapter “Thread-Local Storage TLS” added.
4.38	0	170921	TS	New software version.
4.36	0	170731	MC	New software version.
4.34	0	170411	MC	New software version.
4.32	0	170109	MM/MC	Chapter “embOS Thread script” updated.
4.24	0	160706	MC	Chapter “VFP support” updated.
4.16	0	160404	RH	Added new chapter: embOS Thread Script
4.14	0	151211	MC	New software version.
4.12b	0	151005	MC	New software version.
4.12	0	150821	SC	New software version.
4.10b	0	150609	SC	Initial version.

 © 2010-2025 SEGGER Microcontroller GmbH

 © 2010-2025 SEGGER Microcontroller GmbH

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler).
- The C programming language.
- The target processor.
- DOS command line.

If you feel that your knowledge of C is not sufficient, we recommend *The C Programming Language* by Kernighan and Richie (ISBN 0--13--1103628), which describes the standard in C programming and, in newer editions, also covers the ANSI C standard.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections.

© 2010-2025 SEGGER Microcontroller GmbH

 © 2010-2025 SEGGER Microcontroller GmbH

Table of contents

1 Using embOS 9

1.1 Installation 10
1.2 First Steps 11
1.3 The example application OS_StartLEDBlink.c 12
1.4 Stepping through the sample application 13

2 Build your own application 16

2.1 Introduction 17
2.2 Required files for an embOS 17
2.3 Change library mode 17
2.4 Select another CPU 17

3 Libraries 18

3.1 Naming conventions for prebuilt libraries 19

4 CPU and compiler specifics 20

4.1 Interrupt and thread safety 21
4.2 Thread-Local Storage TLS 22
4.2.1 API functions 22
4.2.1.1 OS_TLS_Set() 23
4.2.1.2 OS_TLS_SetTaskContextExtension() 24
4.3 SEGGER Stack Overflow Prevention 25
4.3.1 API functions 25
4.3.1.1 OS_STOP_Set() 26
4.3.1.2 OS_STOP_SetTaskContextExtension() 27
4.4 Arm erratum 837070 28
4.5 Arm Stack limit register PSPLIM 29
4.5.1 API functions 29
4.5.1.1 OS_PSPLIM_Set() 30
4.5.1.2 OS_PSPLIM_SetTaskContextExtension() 31
4.6 Arm TrustZone support 32
4.6.1 API functions 32
4.6.1.1 OS_ARM_TZ_SetSecureStatePSP() 33
4.6.1.2 OS_ARM_TZ_SetTaskContextExtension() 34
4.7 Arm Pointer Authentication (PAC) and Branch target identification (BTI) 35

5 Stacks 36

5.1 Task stack for Cortex-M 37
5.2 System stack for Cortex-M 37

 © 2010-2025 SEGGER Microcontroller GmbH

5.3 Interrupt stack for Cortex-M 37

6 Interrupts 38

6.1 What happens when an interrupt occurs? 39
6.2 Defining interrupt handlers in C 39
6.3 Interrupt vector table 39
6.4 Interrupt-stack switching 40
6.5 Zero latency interrupts 40
6.6 Interrupt priorities 40
6.7 Interrupt nesting 42
6.8 Interrupt enable behavior 43
6.9 Interrupt handling 44
6.9.1 API functions 44
6.9.1.1 OS_ARM_ISRInit() 45
6.9.1.2 OS_ARM_InstallISRHandler() 46
6.9.1.3 OS_ARM_EnableISR() 47
6.9.1.4 OS_ARM_DisableISR() 48
6.9.1.5 OS_ARM_ISRSetPrio() 49

7 CMSIS 50

7.1 Introduction 51
7.2 The generic CMSIS start project 52
7.3 Device specific files needed for embOS with CMSIS 52
7.4 Device specific functions/variables needed for embOS with CMSIS 52
7.5 CMSIS generic functions needed for embOS with CMSIS 53
7.6 Customizing the embOS CMSIS generic start project 53
7.7 Adding CMSIS to other embOS start projects 53
7.8 Interrupt and exception handling with CMSIS 55
7.8.1 Enable and disable interrupts 55
7.8.2 Setting the Interrupt priority 55

8 Floating Point (FP) support 56

8.1 ARM Floating-point Extension 57
8.2 Using embOS libraries with floating-point support 57
8.3 Using the FPU in interrupt service routines 57
8.4 FPU default behavior 57
8.5 GCC FP compiler options 58

9 RTT and SystemView 59

9.1 SEGGER Real Time Transfer 60
9.2 SEGGER SystemView 60

10 embOS Thread Script 61

10.1 Introduction 62
10.2 How to use it 62

11 Technical data 68

11.1 Resource Usage 69

 © 2010-2025 SEGGER Microcontroller GmbH

Chapter 1

Using embOS

© 2010-2025 SEGGER Microcontroller GmbH

1.1 Installation

This chapter describes how to get started with embOS. You should follow these steps to become familiar with embOS.

embOS is shipped as a zip-file in electronic form. To install it, you should extract the zip-file to any folder of your choice while preserving its directory structure (i.e. keep all files in their respective sub directories). Ensure the files are not read-only after extraction. Assuming that you are using an IDE to develop your application, no further installation steps are required.

> **Note**
>
> The projects at `/Start/BoardSupport/<DeviceManufacturer>/<Board>` assume a relative location for the `/Start/Lib` and `/Start/Inc` folders. If you copy a BSP folder to another location, you will need to adjust the include paths of the project accordingly.

At `/Start/BoardSupport/<DeviceManufacturer>/<Board>` you should find several example start projects, which you may adapt to write your application. To do so, follow the instructions of section *First Steps* on page 11.

In order to become familiar with embOS, consider using the example projects (even if you will not use the IDE for application development).

If you do not or do not want to work with an IDE, you may copy either all library files or only the library that is used with your project into your work directory. embOS does in not rely on an IDE, but may be used without an IDE just as well, e.g. using batch files or a make utility.

 © 2010-2025 SEGGER Microcontroller GmbH

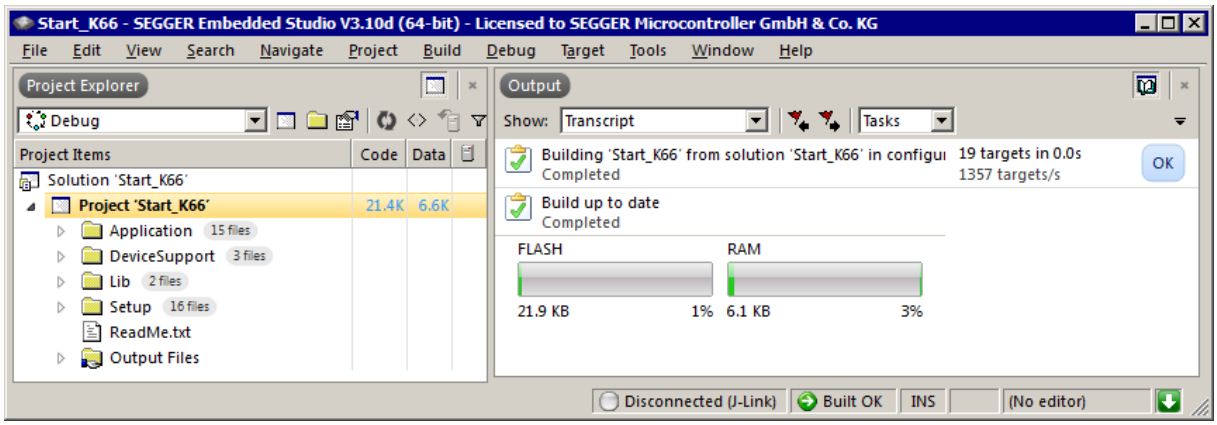
1.2 First Steps

After installation of embOS, you can create your first multitasking application. You received several ready-to-go sample workspaces and projects as well as all required embOS files inside the subfolder `Start`. The subfolder `Start/BoardSupport` contains the workspaces and projects, sorted into manufacturer- and board-specific subfolders. It is a good idea to use one of the projects as a starting point for any application development.

To get your new application running, you should:

- Create a directory for your development.
- Copy the whole `Start` folder from your embOS shipment into the directory.
- Clear the read-only attribute of all files in the copied `Start` folder.
- Open one sample workspace/project in `Start/BoardSupport/<DeviceManufacturer>/<Board>` with your IDE (for example, by double clicking it).
- Build the project. It should be built without any error or warning messages.

After building the project of your choice, the screen should look like this:



For additional information, you should open the ReadMe.txt file that is part of every BSP. It describes the different configurations of the project and, if required, gives additional information about specific hardware settings of the supported evaluation board(s).

 © 2010-2025 SEGGER Microcontroller GmbH

1.3 The example application OS_StartLEDBlink.c

The following is a printout of the example application `OS_StartLEDBlink.c`. It is a good starting point for your application (the actual file shipped with your port of embOS may differ slightly).

What happens is easy to see:

After initialization of embOS, two tasks are created and started. The two tasks get activated and execute until they run into a delay, thereby suspending themselves for the specified time, and eventually continue execution.

```
/*********************************************************************
*                     SEGGER Microcontroller GmbH                    *
*                        The Embedded Experts                        *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------
File    : OS_StartLEDBlink.c
Purpose : embOS sample program running two simple tasks, each toggling
          an LED of the target hardware (as configured in BSP.c).
*/

#include "RTOS.h"
#include "BSP.h"

static OS_STACKPTR int StackHP[128], StackLP[128];  // Task stacks
static OS_TASK         TCBHP, TCBLP;                // Task control blocks

static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_TASK_Delay(50);
  }
}

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_TASK_Delay(200);
  }
}

/*********************************************************************
*
*       main()
*/
int main(void) {
  OS_Init();    // Initialize embOS
  OS_InitHW();  // Initialize required hardware
  BSP_Init();   // Initialize LED ports
  OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
  OS_TASK_CREATE(&TCBLP, "LP Task",  50, LPTask, StackLP);
  OS_Start();   // Start embOS
  return 0;
}

/************************** End of file ****************************/
```

 © 2010-2025 SEGGER Microcontroller GmbH

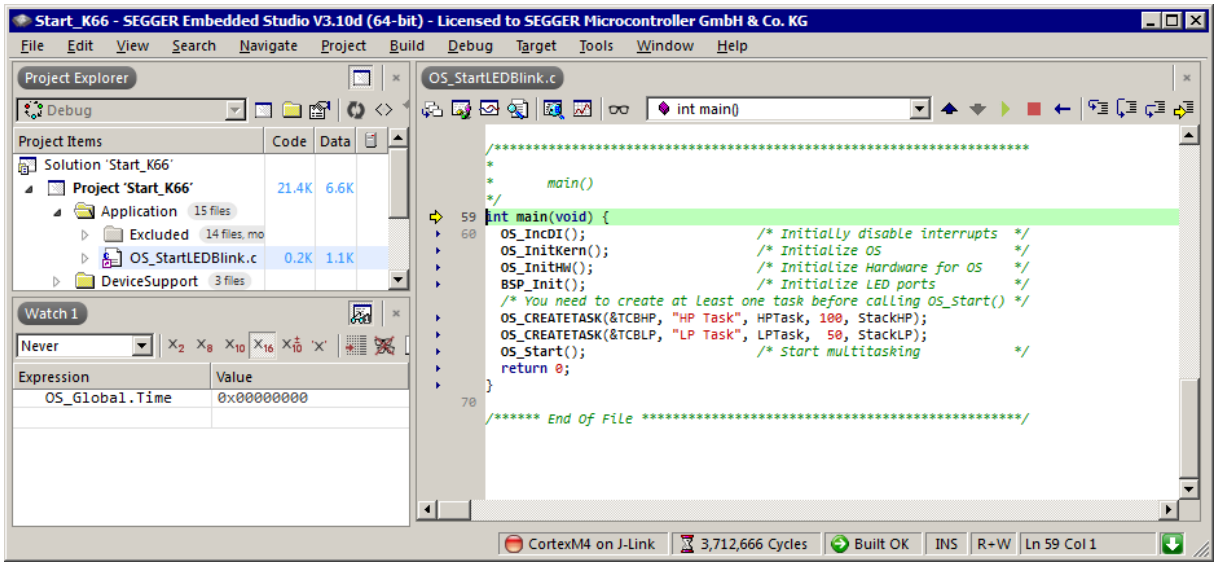
1.4 Stepping through the sample application

When starting the debugger, you will see the `main()` function (see example screenshot below). The `main()` function appears as long as project option `Run to main` is selected, which it is enabled by default. Now you can step through the program.

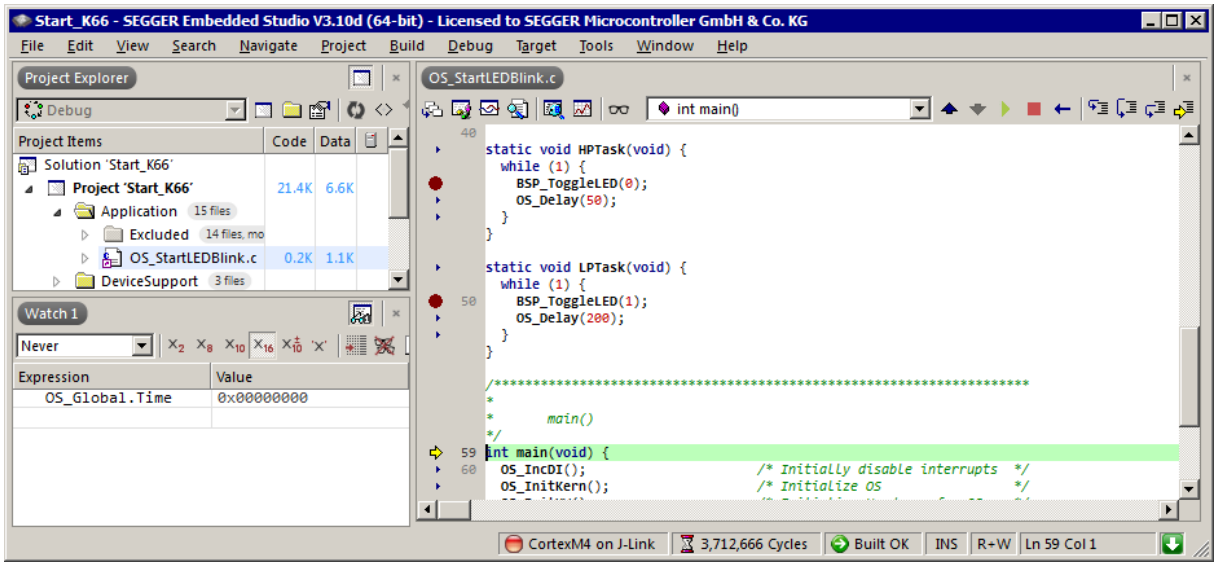
`OS_Init()` is part of the embOS library and written in assembler; you can therefore only step into it in disassembly mode. It initializes the relevant OS variables.

`OS_InitHW()` is part of `RTOSInit.c` and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the system tick interrupt for embOS. Step through it to see what is done.

`OS_Start()` should be the last line in `main()`, because it starts multitasking and does not return.



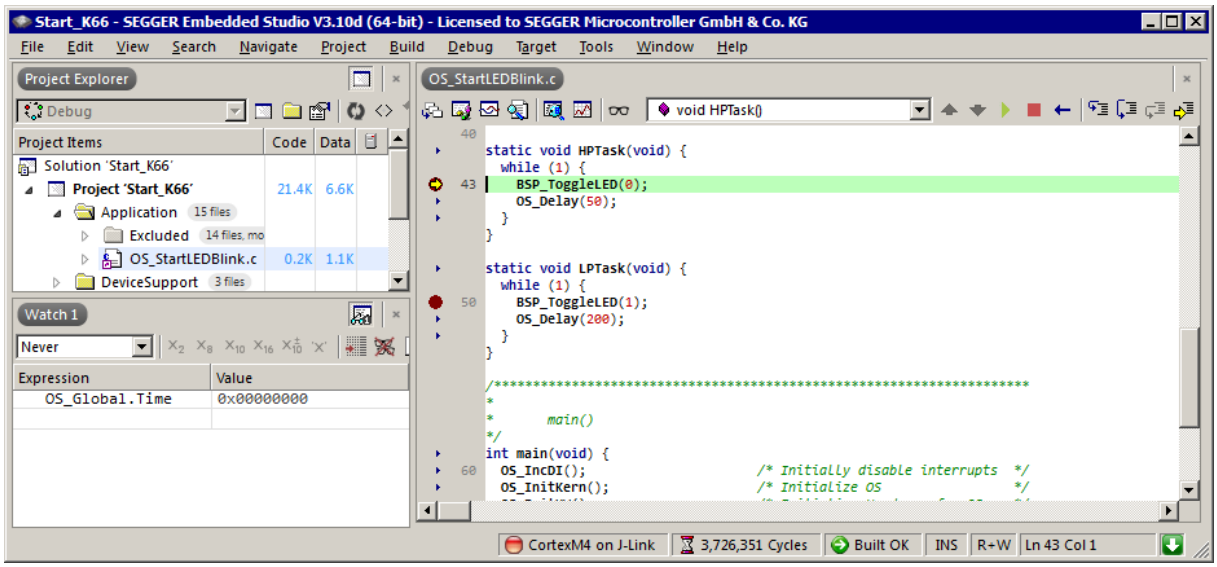
Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.



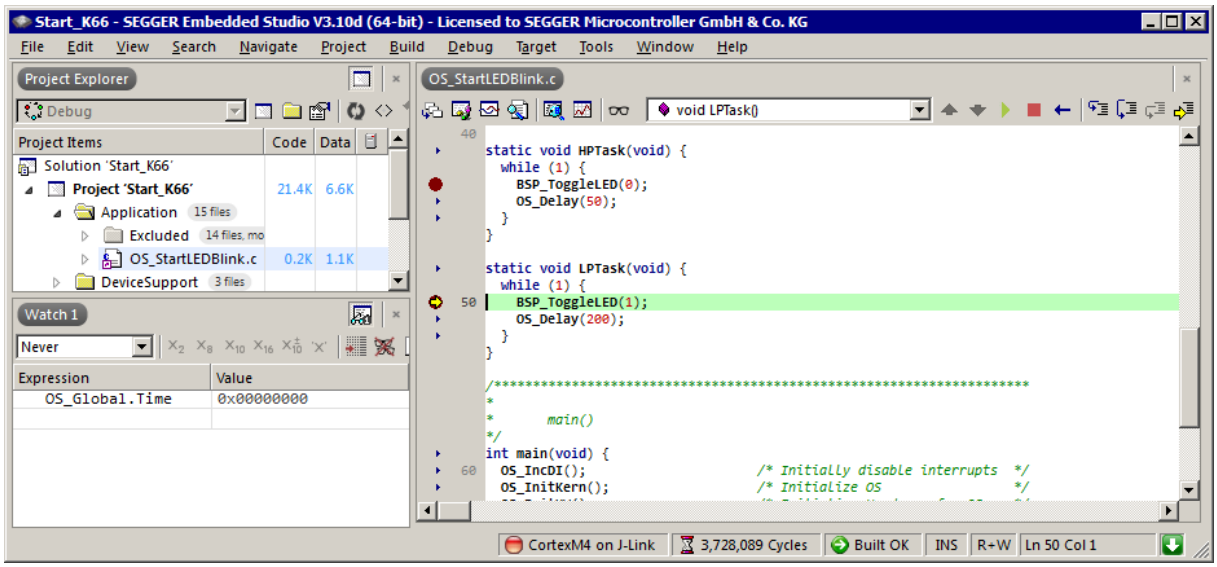
As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only.

 © 2010-2025 SEGGER Microcontroller GmbH

Click GO, step over OS_Start(), or step into OS_Start() in disassembly mode until you reach the highest priority task.



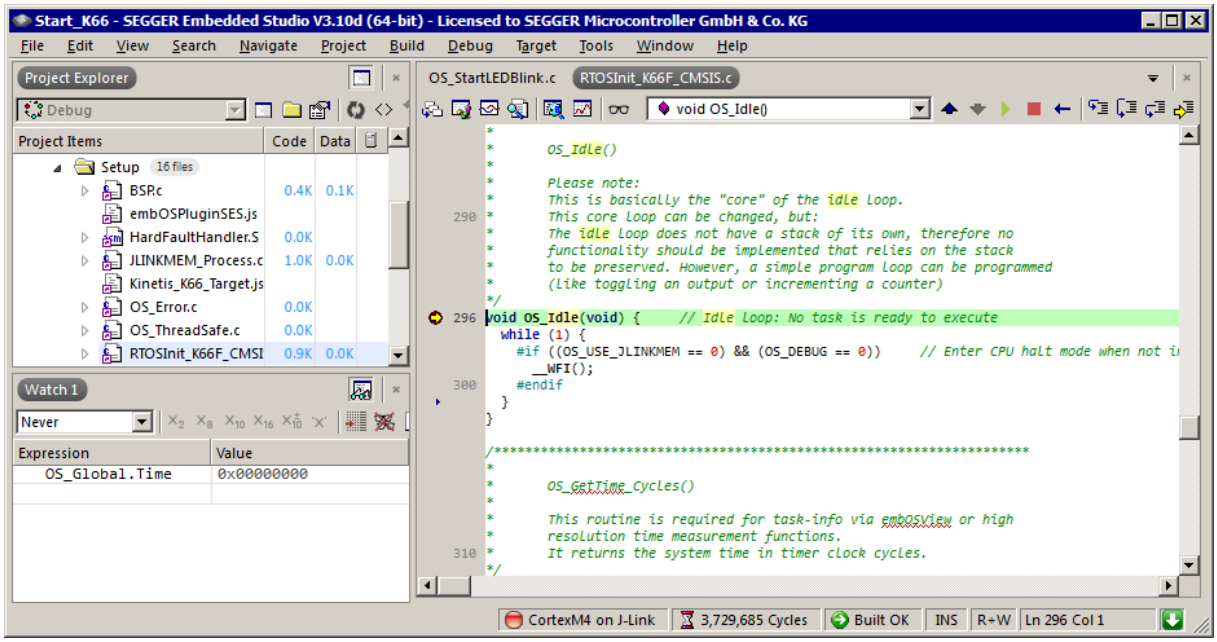
If you continue stepping, you will arrive at the task that has lower priority:



 © 2010-2025 SEGGER Microcontroller GmbH

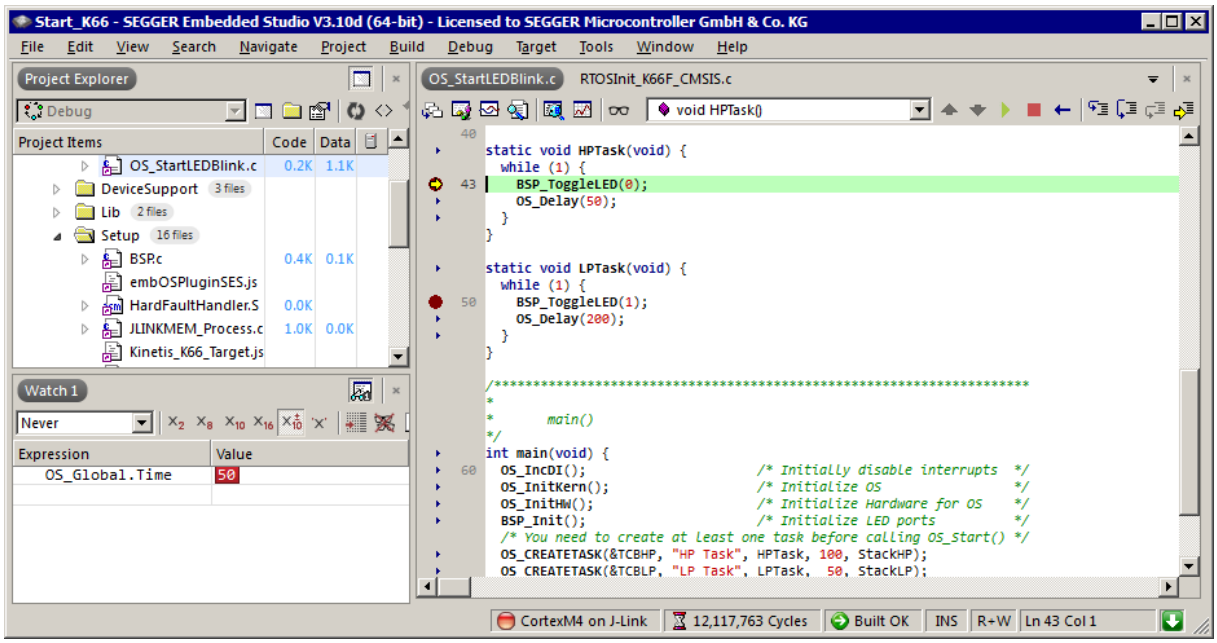
Continue to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

You will arrive there when you step into the `OS_TASK_Delay()` function in disassembly mode. `OS_Idle()` is part of `RTOSInit.c`. You may also set a breakpoint there before stepping over the delay in `LPTask()`.



If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay.

As can be seen by the value of embOS timer variable `OS_Global.Time`, shown in the Watch window, `HPTask()` continues operation after expiration of the delay.



 © 2010-2025 SEGGER Microcontroller GmbH

Chapter 2

Build your own application

© 2010-2025 SEGGER Microcontroller GmbH

2.1 Introduction

This chapter provides all information to set up your own embOS project. To build your own application, you should always start with one of the supplied sample workspaces and projects. Therefore, select an embOS workspace as described in chapter *First Steps* on page 11 and modify the project to fit your needs. Using an embOS start project as starting point has the advantage that all necessary files are included and all settings for the project are already done.

2.2 Required files for an embOS

To build an application using embOS, the following files from your embOS distribution are required and have to be included in your project:

- **RTOS.h** from the directory `.\Start\Inc`. This header file declares all embOS API functions and data types and has to be included in any source file using embOS functions.
- **RTOSInit*.c** from one target specific `.\Start\BoardSupport\<Manufacturer>\<MCU>` subfolder. It contains hardware-dependent initialization code for embOS. It initializes the system timer interrupt but can also initialize or set up the interrupt controller, clocks and PLLs, the memory protection unit and its translation table, caches and so on.
- **OS_Error.c** from one target specific subfolder `.\Start\BoardSupport\<Manufacturer>\<MCU>`. The error handler is used only if a debug library is used in your project.
- One **embOS library** from the subfolder `.\Start\Lib`.
- Additional CPU and compiler specific files may be required according to CPU.

When you decide to write your own startup code or use a low level `init()` function, ensure that non-initialized variables are initialized with zero, according to C standard. This is required for some embOS internal variables. Your `main()` function has to initialize embOS by calling `OS_Init()` and `OS_InitHW()` prior to any other embOS functions that are called.

2.3 Change library mode

For your application you might want to choose another library. For debugging and program development you should always use an embOS debug library. For your final application you may wish to use an embOS release library or a stack check library.

Therefore you have to select or replace the embOS library in your project or target:

- If your selected library is already available in your project, just select the appropriate project configuration.
- To add a library, you may add the library to the existing Lib group. Exclude all other libraries from your build, delete unused libraries or remove them from the configuration.
- Check and set the appropriate `OS_LIBMODE_*` define as preprocessor option and/or modify the `OS_Config.h` file accordingly.

2.4 Select another CPU

embOS contains CPU-specific code for various CPUs. Manufacturer- and CPU-specific sample start workspaces and projects are located in the subfolders of the `.\Start\BoardSupport` directory. To select a CPU which is already supported, just select the appropriate workspace from a CPU-specific folder.

If your CPU is currently not supported, examine all `RTOSInit.c` files in the CPU-specific subfolders and select one which almost fits your CPU. You may have to modify `OS_InitHW()`, the interrupt service routines for the embOS system tick timer and the low level initialization.

 © 2010-2025 SEGGER Microcontroller GmbH

Chapter 3

Libraries

© 2010-2025 SEGGER Microcontroller GmbH

3.1 Naming conventions for prebuilt libraries

embOS is shipped with different pre-built libraries with different combinations of features. Note that not all combinations are available (e.g., there is no VFP support for Armv6-M).

The libraries are named as follows:

`libos_v<Architecture><VFP><Endianness><LibMode><Errata><Trust-Zone><PACBTI><STOP>.a`

Parameter	Meaning	Values
`Architecture`	ARM architecture	`6m` : Cortex-M0/M0+/M1 `7m` : Cortex-M3/M4/M7 `8mbl` : Cortex-M23 `8mml` : Cortex-M33 `81mml`: Cortex-M55/M85
`VFP_support`	Floating point support	: No hardware VFP support `_vfpv4` : Softfp floating-point ABI `_vfpv4h`: Hardware floating-point ABI
`Endianness`	Byte order	`_tbe` : Thumb mode, big endian `_tle` : Thumb mode, little endian
`LibMode`	embOS library mode	`_xr` : Extreme Release `_r` : Release `_s` : Stack check `_sp` : Stack check + profiling `_d` : Debug `_dp` : Debug + Profiling + Stack check `_dt` : Debug + Profiling + Stack check + Trace
`Errata`	Considered Arm errata	`_837070` : Erratum 837070 : No erratum considered
`TrustZone`	Armv8-M TrustZone support	`_tz` : TrustZone support : No TrustZone support
`PACBTI`	Armv8.1-M PACBTI support	`_pacbti` : PACBTI support : No PACBTI support
`STOP`	SEGGER STOP support	`_stop` : Stack Overflow prevention : No Stack Overflow prevention

Example

`libos_v7m_t_le_dp.a` is the library for a project using Cortex-M3 core, thumb mode, little endian mode with debug and profiling support.

 © 2010-2025 SEGGER Microcontroller GmbH

Chapter 4

CPU and compiler specifics

© 2010-2025 SEGGER Microcontroller GmbH

4.1 Interrupt and thread safety

Using embOS with specific calls to standard library functions (e.g. heap management functions) may require thread-safe system libraries if these functions are called from several tasks or interrupts. Embedded Studio's system library provides functions, which can be overwritten to implement a locking mechanism making the system library functions thread-safe.

The Setup directory in each embOS BSP contains the file `OS_ThreadSafe.c` which overwrites these functions. By default they disable and restore embOS interrupts to ensure thread safety in tasks, embOS interrupts, `OS_Idle()` and software timers. Zero latency interrupts are not disabled and therefore unprotected. If you need to call e.g. malloc() also from within a zero latency interrupt additional handling needs to be added. If you don't call such functions from within embOS interrupts, `OS_Idle()` or software timers, you can instead use thread safety for tasks only. This reduces the interrupt latency because a mutex is used instead of disabling embOS interrupts.

You can choose the safety variant with the macro `OS_INTERRUPT_SAFE`.

- When defined to 1 thread safety is guaranteed in tasks, embOS interrupts, `OS_Idle()` and software timers.
- When defined to 0 thread safety is guaranteed only in tasks. In this case you must not call e.g. heap functions from within an ISR, `OS_Idle()` or embOS software timers.

Alternatively, embOS delivers its own thread-safe functions for heap management. These are described in the embOS generic manual.

 © 2010-2025 SEGGER Microcontroller GmbH

4.2 Thread-Local Storage TLS

Embedded Studio's standard library supports usage of thread-local storage. Several library objects and functions need local variables which have to be unique to a thread. Thread-local storage will be required when these functions are called from multiple threads.

embOS for Embedded Studio is prepared to support the tread-local storage, but does not use it per default. This has the advantage of no additional overhead as long as thread-local storage is not needed by the application. The embOS implementation of thread-local storage allows activation of TLS separately for each task.

Only tasks that are accessing TLS variables, for instance by calling functions from the system library, need to initialize their TLS by calling an initialization function when the task is started. For each task that uses TLS the memory for the thread-local storage is allocated on the heap. Therefore, thread-safe heap management should be used together with TLS. For information on thread safety, please refer to *Interrupt and thread safety* on page 21.

When the task terminates by a call of `OS_TASK_Terminate()`, the memory used for TLS is automatically freed and put back into the free heap memory.

Library objects that need thread-local storage when used in multiple tasks are for example:

- error functions - errno, strerror.
- locale functions - localeconv, setlocale.
- time functions - asctime, localtime, gmtime, mktime.
- multibyte functions - mbrlen, mbrtowc, mbsrtowc, mbtowc, wcrtomb, wcsrtomb, wctomb.
- rand functions - rand, srand.
- etc functions - atexit, strtok.
- C++ exception engine.

4.2.1 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
`OS_TLS_Set()`	Initializes the thread-local storage for the current task.		•			
`OS_TLS_SetTaskContextExtension()`	Initializes the thread-local storage and sets the TLS task context extension for the current task.		•			

 © 2010-2025 SEGGER Microcontroller GmbH

4.2.1.1 OS_TLS_Set()

Description

Initializes the thread-local storage for the current task.

Prototype

```
void OS_TLS_Set(void);
```

Additional information

`OS_TLS_Set()` shall be the first function called from a task when TLS should be used in this task.

The required memory for the thread-local storage is allocated from the heap. If `OS_TLS_Set()` was already called for this task or if there is not enough memory on the heap, then embOS will call `OS_Error()` with the error code `OS_ERR_TLS_INIT`.

This function has to be used only in combination with `OS_TASK_AddContextExtension()` or `OS_TASK_SetContextExtension()` and `OS_TLS_ContextExtension` as argument to these functions. When `OS_TLS_SetTaskContextExtension()` is used, `OS_TLS_Set()` will be called automatically.

Example

```
static void Task(void) {
  OS_TLS_Set();
  OS_TASK_SetContextExtension(&OS_TLS_ContextExtension);
  while (1) {
  }
}
```

 © 2010-2025 SEGGER Microcontroller GmbH

4.2.1.2 OS_TLS_SetTaskContextExtension()

Description

Initializes the thread-local storage and sets the TLS task context extension for the current task.

Prototype

```
void OS_TLS_SetTaskContextExtension(void);
```

Additional information

`OS_TLS_SetTaskContextExtension()` shall be the first function called from a task when TLS should be used in this task.

The required memory for the thread-local storage is allocated from the heap. If `OS_TLS_Set()` was already called for this task or if there is not enough memory on the heap, then embOS will call `OS_Error()` with the error code `OS_ERR_TLS_INIT`.

If the task already contains a task context extension, `OS_TLS_SetTaskContextExtension()` cannot be used. Instead, `OS_TASK_AddContextExtension()` needs to be called with `OS_TLS_ContextExtension` as argument. Furthermore, `OS_TLS_Set()` needs to be called to initialize TLS for this task.

Example

The following printout demonstrates the usage of task specific TLS in an application.

```
#include "RTOS.h"

static OS_STACKPTR int StackHP[128], StackLP[128];  // Task stacks
static OS_TASK         TCBHP, TCBLP;                // Task control blocks

static void HPTask(void) {
  OS_TLS_SetTaskContextExtension();
  while (1) {
    errno = 42;  // errno specific to HPTask
    OS_TASK_Delay(50);
  }
}

static void LPTask(void) {
  OS_TLS_SetTaskContextExtension();
  while (1) {
    errno = 1;  // errno specific to LPTask
    OS_TASK_Delay(200);
  }
}

int main(void) {
  errno = 0;      // errno not specific to any task
  OS_Init();      // Initialize embOS
  OS_InitHW();    // Initialize required hardware
  OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
  OS_TASK_CREATE(&TCBLP, "LP Task",  50, LPTask, StackLP);
  OS_Start();     // Start embOS
  return 0;
}
```

 © 2010-2025 SEGGER Microcontroller GmbH

4.3 SEGGER Stack Overflow Prevention

In order to detect stack overflows in a running embedded system the SEGGER Compiler is able to generate code to check for stack overflows in every function. This can be activated using the command line switch -mstack-overflow-check (cc1 interface). For secure systems, a stack overflow must be detected before any memory is destroyed by the overflowing stack, therefore a check is required on any change of the stack pointer and before any large stack growth.

Stack Overflow Prevention with Embedded Studio

Embedded Studio makes the use of Stack Overflow Prevention as easy as possible. To enable the feature, set the project option Code -> Code Generation -> Enable Stack Overflow Prevention to Yes.

If the error callback `__SEGGER_STOP_X_OnError()` is not implemented in the project, it defaults to staying in an endless loop. An example implementation for error handling is included in `SEGGER_STOP.c` in $(StudioDir)/samples.

embOS Cortex-M Embedded Studio comes with its own implementation of `__SEGGER_STOP_X_OnError()` in `SEGGER_STOP_Handler.c`.

Stack Limits

SEGGER Stack Overflow Prevention expects that `__SEGGER_STOP_Limit_MSP` and `__SEGGER_STOP_Limit_PSP` are set to the corresponding stack limits. If `__SEGGER_STOP_Limit_MSP` or `__SEGGER_STOP_Limit_PSP` are set to zero the corresponding stack overflow prevention is disabled.

The startup code initializes the main stack limit variable `__SEGGER_STOP_Limit_MSP`. `__SEGGER_STOP_Limit_MSP` is used when embOS runs on the main stack pointer. main(), the scheduler, software timer and interrupts are executed on the main stack pointer.

`__SEGGER_STOP_Limit_PSP` is used when embOS runs on the process stack pointer. embOS tasks are executed on the process stack pointer. At context switch `__SEGGER_STOP_Limit_PSP` needs to be set to the corresponding task stack limit (or to zero which disables the stack overflow prevention).

4.3.1 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
`OS_STOP_Set()`	Sets the Stack Overflow Prevention stack limit.		•			
`OS_STOP_SetTaskContextExtension()`	Extends the task context with the SEGGER Stack Overflow Prevention.		•			

 © 2010-2025 SEGGER Microcontroller GmbH

4.3.1.1 OS_STOP_Set()

Description

Sets the Stack Overflow Prevention stack limit.

Prototype

```
void OS_STOP_Set(OS_U32 StackLimit);
```

Parameters

Parameter	Description
StackLimit	Stack limit

Additional information

OS_STOP_Set() sets the STOP variable __SEGGER_STOP_Limit_PSP to the given StackLimit parameter plus space for one interrupt. An interrupt stores CPU (and FPU, if used) registers on the current stack. The task context must be extended with OS_TASK_AddContextExtension(), OS_TASK_SetContextExtension() or OS_TASK_SetDefaultContextExtension().

Example

```
#include "RTOS.h"

static OS_STACKPTR int StackHP[128];
static OS_TASK         TCBHP;

static void Task(void) {
  OS_STOP_Set((OS_U32)StackHP);
  OS_TASK_SetContextExtension(&OS_STOP_ContextExtension);
  while (1) {
    OS_TASK_Delay(100);
  }
}
```

 © 2010-2025 SEGGER Microcontroller GmbH

4.3.1.2 OS_STOP_SetTaskContextExtension()

Description

Extends the task context with the SEGGER Stack Overflow Prevention.

Prototype

```
void OS_STOP_SetTaskContextExtension(OS_U32 StackLimit);
```

Parameters

Parameter	Description
StackLimit	Stack limit

Additional information

OS_STOP_SetTaskContextExtension() initially sets the STOP variable __SEGGER_STOP_Limit_PSP to the given StackLimit parameter plus space for one interrupt. An interrupt stores CPU (and FPU, if used) registers on the current stack.

This is not done when the task context is extended with OS_TASK_AddContextExtension() or OS_TASK_SetContextExtension(). In that case the STOP variable __SEGGER_STOP_Limit_PSP must be set with OS_STOP_Set(). It can also be set manually by the application when another limit should be used. For example safety applications might want to call the STOP error handler when the stack is filled more than 70%.

After using this function, any further task context extensions cannot be added by calling OS_TASK_SetContextExtension(), but can be added using OS_TASK_AddContextExtension() instead.

If a task has already another task context extension set, the STOP task context extension can be added by passing the predefined OS_STOP_ContextExtension structure to OS_TASK_AddContextExtension().

Example

```
#include "RTOS.h"

static OS_STACKPTR int StackHP[128];
static OS_TASK         TCBHP;

static void Task(void) {
  OS_STOP_SetTaskContextExtension((OS_U32)StackHP);
  while (1) {
    OS_TASK_Delay(100);
  }
}
```

 © 2010-2025 SEGGER Microcontroller GmbH

4.4 Arm erratum 837070

Specific embOS ARMv7-M libraries use a workaround for the Cortex-M7 erratum 837070 (refer to *Naming conventions for prebuilt libraries* on page 19). When an embOS library without this workaround is used with a device that requires the workaround, debug builds of embOS will call `OS_Error()` with the error code `OS_ERR_LIB_INCOMPATIBLE`.

Cortex-M7 devices that implement the ARM core r0p0 or r0p1 are affected by the erratum, while later versions of these devices are unaffected. The workaround sets `PRIMASK` before writing to `BASEPRI` and restores `PRIMASK` afterwards; it therefore adds a minimal latency to Zero latency interrupts.

When working with an affected device, the define `USE_ERRATUM_837070` shall be set to 1 in the preprocessor settings or inside `OS_Config.h`, regardless of whether the embOS source code or an embOS library is being used.
Additionally, if working with the embOS source code, it also is possible for efficiency reasons to not restore any previous value of `PRIMASK` after modification of `BASEPRI`. To do so, the define `OS_PRESERVE_PRIMASK` shall be set to 0 in the preprocessor settings or inside `OS_Config.h`.

> **Note**
>
> With earlier versions of embOS for Cortex-M the define `OS_PRESERVE_PRIMASK` was set per default to 0.

 © 2010-2025 SEGGER Microcontroller GmbH

4.5 Arm Stack limit register PSPLIM

When the ARMv8-M Security Extension is included, there are two PSPLIM registers in the processor:

- PSPLIM_NS for the Non-secure state.
- PSPLIM_S for the Secure state.

The hardware continuously compares the process stack register (PSP) against this process stack limit register (PSPLIM). If the PSP is lower than the PSPLIM register value a stack overflow occurred and a fault exception is generated.

embOS Cortex-M comes with a task context extension for the PSPLIM register. Each task context can be extended by the call of OS_PSPLIM_SetTaskContextExtension(). The task context extension saves and restores the PSPLIM register on the according task stack. When a task gets deactivated the PSPLIM register is set to zero which deactivates the PSPLIM stack check for other tasks which do not use this extension.

4.5.1 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
OS_PSPLIM_Set()	Sets the PSPLIM register.		●			
OS_PSPLIM_SetTaskContextExtension()	Extends the task context with the stack check limit register PSPLIM.		●			

 © 2010-2025 SEGGER Microcontroller GmbH

4.5.1.1 OS_PSPLIM_Set()

Description

Sets the PSPLIM register.

Prototype

```
void OS_PSPLIM_Set(const void OS_STACKPTR *pStack);
```

Parameters

Parameter	Description
pStack	Pointer to the task stack.

Additional information

The PSPLIM register is banked between security states. `OS_PSPLIM_Set()` initially sets the `PSPLIM` register of the currently active security state to the parameter `pStack`.

If you like to use the PSPLIM register for more than one task the task context needs to be extended with e.g. `OS_TASK_AddContextExtension()` or `OS_PSPLIM_SetTaskContextExtension()`.

The PSPLIM register can only be written in privileged state. Unprivileged writes to PSPLIM are ignored.

Since the PSPLIM register may hold 8-byte aligned values only, debug builds of embOS check for correct alignment and call `OS_Error()` with error code `OS_ERR_INV_PARAMETER_VALUE` if `pStack` is not properly aligned.

Example

```
static OS_STACKPTR int StackHP[128];
static OS_TASK         TCBHP;

static void HPTask(void) {
  OS_EXTEND_TASK_CONTEXT_LINK PSPLIM_ContextExtensionLink;

  OS_TASK_AddContextExtension(&PSPLIM_ContextExtensionLink,
                              &OS_PSPLIM_ContextExtension);
  OS_PSPLIM_Set(StackHP);
  while (1) {
    BSP_ToggleLED(0);
    OS_TASK_Delay(50);
  }
}
```

 © 2010-2025 SEGGER Microcontroller GmbH

4.5.1.2 OS_PSPLIM_SetTaskContextExtension()

Description

Extends the task context with the stack check limit register PSPLIM.

Prototype

```
void OS_PSPLIM_SetTaskContextExtension(const void OS_STACKPTR *pStack);
```

Additional information

`OS_PSPLIM_SetTaskContextExtension()` initially sets the PSPLIM register to the parameter `pStack`. This is not done when the task context is extended with `OS_TASK_AddContextExtension()` or `OS_TASK_SetContextExtension()`. In that case the PSPLIM register should be set manually with `OS_PSPLIM_Set()`.

After using this function, any further task context extensions cannot be added by calling `OS_TASK_SetContextExtension()`, but can be added using `OS_TASK_AddContextExtension()` instead.

If a task has already another task context extension set, the PSPLIM task context extension can be added by passing the predefined `OS_PSPLIM_ContextExtension` structure to `OS_TASK_AddContextExtension()`.

`OS_PSPLIM_SetTaskContextExtension()` handles the PSPLIM register of the security state the embOS runs in. By default, embOS runs in the secure world, thus saving and restoring the `PSPLIM_s` register on context switches. If TrustZone is used, i.e. embOS and tasks run in the non-secure world, then the `PSPLIM_ns` register is saved and restored on context switches. However, non-secure tasks can also set a dedicated task context extension for TrustZone which additionally saves and restores `PSP_s`, `PSPLIM_s` and `CONTROL_s` of the non-secure task so that it can perform calls into the secure world.

For more information about TrustZone and the TrustZone context extension, please refer to *Arm TrustZone support* on page 32.

Since the PSPLIM register may hold 8-byte aligned values only, debug builds of embOS check for correct alignment and call `OS_Error()` with error code `OS_ERR_INV_PARAMETER_VALUE` if `pStack` is not properly aligned.

Example

```
static OS_STACKPTR int StackHP[128];
static OS_TASK         TCBHP;

static void HPTask(void) {
  OS_PSPLIM_SetTaskContextExtension(StackHP);
  while (1) {
    BSP_ToggleLED(0);
    OS_TASK_Delay(50);
  }
}
```

 © 2010-2025 SEGGER Microcontroller GmbH

4.6 Arm TrustZone support

embOS Cortex-M comes with libraries for Arm TrustZone support. With it embOS runs completely in the non-secure world but tasks can call functions from the secure world. When using the embOS sources the define `OS_SUPPORT_TRUSTZONE = 1` must be used.

If an embOS task wants to call secure functions the secure register `PSP_S`, `PSPLIM_S` and `CONTROL_S` must be set beforehand and the task context must be extended to save and restore these register at every context switch. An embOS task runs in secure state on a separate stack which is located in the secure memory.

You can use `OS_ARM_TZ_SetSecureStatePSP()` or `OS_ARM_TZ_SetTaskContextExtension()` to set the secure register. Additionally, `OS_ARM_TZ_SetTaskContextExtension()` extends the task context. `OS_ARM_TZ_SetSecureStatePSP()` sets the secure register only and the task context must be extended with `OS_TASK_AddContextExtension()` or `OS_TASK_SetContextExtension()` and the context extension `OS_ARM_TZ_ContextExtension`.

4.6.1 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
`OS_ARM_TZ_SetSecureStatePSP()`	Sets the secure `PSP_S`, `PSPLIM_S` and `CONTROL_S` registers.		•			
`OS_ARM_TZ_SetTaskContextExtension()`	Sets the secure `PSP_S`, `PSPLIM_S` and `CONTROL_S` registers and extends the task context to save and restore these registers.		•			

 © 2010-2025 SEGGER Microcontroller GmbH

4.6.1.1 OS_ARM_TZ_SetSecureStatePSP()

Description

Sets the secure PSP_S, PSPLIM_S and CONTROL_S registers.

Prototype

```
void OS_ARM_TZ_SetSecureStatePSP(      OS_ARM_TZ_SECURE_API_LIST* ApiList,
                                 const void*                      pStack,
                                       unsigned long              StackSize);
```

Parameters

Parameter	Description
ApiList	Pointer to a list of secure functions for accessing PSP_S, PSPLIM_S and CONTROL_S.
pStack	Pointer to the secure task stack.
StackSize	Size of the secure stack to which pStack points.

Additional information

The parameter ApiList must point to a function pointer list with secure functions for accessing the secure process stack pointer, secure process stack limit and the secure control registers. The parameter pStack must point to a stack which is located in the secure memory. This stack is used whenever the task calls a function in the secure world.

OS_ARM_TZ_SetSecureStatePSP() must be called before the task calls any functions from the secure world. The task context must be extended beforehand with the task context OS_ARM_TZ_ContextExtension (e.g. by using OS_TASK_SetContextExtension()).

Example:

```
//
// Locate secure task stack in secure memory.
//
static __no_init OS_STACKPTR int StackHP_s[256] @ "RAM_S";
//
// These functions must be placed in the secure memory.
//
static OS_ARM_TZ_SECURE_API_LIST Arm_TZ_ApiList = {
   Arm_TZ_GetCONTROL_s
  ,Arm_TZ_GetPSP_s
  ,Arm_TZ_GetPSPLIM_s
  ,Arm_TZ_SetCONTROL_s
  ,Arm_TZ_SetPSP_s
  ,Arm_TZ_SetPSPLIM_s
};

static void Task(void) {
  //
  // Extend the task context for the secure world and set the secure register
  //
  OS_TASK_SetContextExtension(&OS_ARM_TZ_ContextExtension);
  OS_ARM_TZ_SetSecureStatePSP(&Arm_TZ_ApiList, StackHP_s, sizeof(StackHP_s));
  while (1) {
    IncrementCounter_s();  // Call secure function and increment secure counter
    OS_TASK_Delay(10);
  }
}
```

 © 2010-2025 SEGGER Microcontroller GmbH

4.6.1.2 OS_ARM_TZ_SetTaskContextExtension()

Description

Sets the secure PSP_S, PSPLIM_S and CONTROL_S registers and extends the task context to save and restore these registers.

Prototype

```
void OS_ARM_TZ_SetTaskContextExtension(      OS_ARM_TZ_SECURE_API_LIST* ApiList,
                                       const void*                     pStack,
                                             unsigned long             StackSize);
```

Parameters

Parameter	Description
ApiList	Pointer to a list of secure functions for accessing PSP_S, PSPLIM_S and CONTROL_S.
pStack	Pointer to the secure task stack.
StackSize	Size of the secure stack to which pStack points.

Additional information

The parameter ApiList must point to a function pointer list with secure functions for accessing the secure process stack pointer, secure process stack limit and the secure control registers. The parameter pStack must point to a stack which is located in the secure memory. This stack is used whenever the task calls a function in the secure world.

OS_ARM_TZ_SetTaskContextExtension() must be called before the task calls any functions from the secure world.

Example:

```
//
// Locate secure task stack in secure memory.
//
static __no_init OS_STACKPTR int StackHP_s[256] @ "RAM_S";
//
// These functions must be placed in the secure memory.
//
static OS_ARM_TZ_SECURE_API_LIST Arm_TZ_ApiList = {
   Arm_TZ_GetCONTROL_s
  ,Arm_TZ_GetPSP_s
  ,Arm_TZ_GetPSPLIM_s
  ,Arm_TZ_SetCONTROL_s
  ,Arm_TZ_SetPSP_s
  ,Arm_TZ_SetPSPLIM_s
};

static void Task(void) {
  //
  // Extend the task context for the secure world.
  //
  OS_ARM_TZ_SetTaskContextExtension(&Arm_TZ_ApiList, StackHP_s, sizeof(StackHP_s));
  while (1) {
    IncrementCounter_s();  // Call secure function and increment secure counter
    OS_TASK_Delay(10);
  }
}
```

 © 2010-2025 SEGGER Microcontroller GmbH

4.7 Arm Pointer Authentication (PAC) and Branch target identification (BTI)

ARMv8.1-M PAC und BTI extensions can be used with embOS. BTI requires that the embOS sources are built with `OS_SUPPORT_ARMV81M_BTI` set to 1 or embOS libraries with PAC and BTI support are used. The application is responsible to enable PAC and BTI and to set the PAC key.

embOS libraries with BTI and PAC support use both features to protect embOS against security attacks. The attacks utilize existing and legitimate code fragments called gadgets. In a successful exploit the attacker gains control over the call stack, for example via stack smashing, and then the pointers stored on the stack are overwritten to point to selected gadgets. By branching from one gadget to another the attacker can escalate the operating privileges and take full control of the system.

Please refer to the ARMv8-M Architecture Reference Manual for more details.

 © 2010-2025 SEGGER Microcontroller GmbH

Chapter 5

Stacks

© 2010-2025 SEGGER Microcontroller GmbH

5.1 Task stack for Cortex-M

Each task uses its own individual stack. The stack pointer is initialized and set every time a task is activated by the scheduler. The stack-size required for a task is the sum of the stack-size of all routines plus a basic stack size plus the size used by exceptions. The basic stack size is the size of memory required to store the registers of the CPU plus the stack size required by calling embOS-routines.

For Cortex-M CPUs, this minimum basic task stack size is about 88 bytes. Because any function call uses some amount of stack and every exception also pushes at least 32 bytes onto the current stack, the task stack size has to be large enough to handle one exception too. For privileged tasks, we recommend at least 512 bytes stack as a start. Unprivileged tasks will require an additional 128 bytes of task stack.

> **Note**
>
> Stacks for Cortex-M devices need to be 8-byte aligned. embOS ensures that task stacks are properly aligned. However, since this can result in unused bytes, the application should ensure that task stacks are properly aligned. This can be achieved by defining an array using a 64-bit data type like `OS_U64`.

5.2 System stack for Cortex-M

The system stack is used in main(), the scheduler, software timers, `OS_Idle()`, and C-level interrupt handlers. The Cortex-M main stack pointer (MSP) is initialized to use the system stack.

The minimum system stack size required by embOS is about 160 bytes (stack check & profiling build). However, the actual stack requirements depend on the application. We recommend a minimum system stack size of 256 bytes.

The size of the system stack can be changed by modifying the project settings or linker file. Please consult your toolchain documentation for more details.

5.3 Interrupt stack for Cortex-M

If an interrupt occurs, the Cortex-M core switches to handler mode, which uses the main stack pointer. Cortex-M C-level interrupt handlers run on the system stack. Please take a look at the previous chapter for more details.

> **Note**
>
> When using an embOS Safe build, please note that the stack-check-limit is configurable through `OS_STACK_SetCheckLimit()` and by default is configured at 70 percent of the total stack size. This will impact the minimum size requirement for both task stacks and the system stack.

 © 2010-2025 SEGGER Microcontroller GmbH

Chapter 6

Interrupts

© 2010-2025 SEGGER Microcontroller GmbH

6.1 What happens when an interrupt occurs?

- The CPU-core receives an interrupt request from the interrupt controller.
- As soon as the interrupts are enabled, the interrupt is accepted and executed.
- The CPU pushes R0-R3, R12, LR, Return Address and xPSR onto the current stack.
- The CPU loads the according `EXC_RETURN` value into LR.
- The CPU switches to handler mode and main stack.
- The CPU jumps to the vector address delivered by the NVIC.
- The interrupt handler is processed.
- The interrupt handler ends with a return from interrupt.
- The CPU uses the `EXC_RETURN` value in LR to switch back to the mode and stack which was active before the exception was entered.
- The CPU restores R0-R3, R12, LR, Return Address and xPSR from the stack and continues execution of the interrupted application.

6.2 Defining interrupt handlers in C

Interrupt handlers for Cortex-M cores are written as normal C-functions which do not take parameters and do not return any value. Interrupt handlers which call an embOS function need a prologue and an epilogue function as described in the generic manual and in the examples below.

Example

Simple interrupt routine:

```
static void _Systick(void) {
  OS_INT_EnterNestable();  // Inform embOS that interrupt code is running
  OS_TICK_Handle();        // May be interrupted
  OS_INT_LeaveNestable();  // Inform embOS that interrupt handler is left
}
```

6.3 Interrupt vector table

After reset, ARM Cortex-M CPUs use an initial `interrupt vector table` located at address `0x00`. It typically contains the initial stack pointer as well as the addresses of all exception handlers required by the application, but at minimum must contain the initial stack pointer and the address of the `Reset_Handler()`.

After reset, the `Vector Table Offset Register` of the `Nested Vectored Interrupt Controller` can be used to indicate a vector table at any other address as well. In case the vector table is kept at address `0x00` or relocated to any other address in ROM, all interrupt handler function addresses must be present in the vector table at compile time. If the vector table is located in RAM, however, interrupt handlers can be installed dynamically at runtime.

6.3.1 Required embOS system interrupt handler

embOS for Cortex-M core needs two exception handlers which belong to the system itself, `PendSV_Handler()` and `SysTick_Handler()`. Both are delivered with embOS. When using your own interrupt vector table, ensure that they are referenced in the vector table.

> **Note**
>
> Some older BSPs used to name the PendSV ISR `OS_Exception()` and thus need to rename it to `PendSV_Handler()`.

 © 2010-2025 SEGGER Microcontroller GmbH

6.4 Interrupt-stack switching

Since Cortex-M core based controllers have two separate stack pointers and embOS utilizes the process stack pointer to execute tasks, there is no need to explicitly switch stacks inside interrupt routines, which utilize the main stack pointer. The routines `OS_INT_EnterIntStack()` and `OS_INT_LeaveIntStack()` are supplied for source code compatibility to other processors only and have no functionality.

6.5 Zero latency interrupts

ARM Cortex-M3, M4, M7 and M33 processors provide a mechanism to raise the interrupt priority level of the CPU in order to disable interrupts with a higher interrupt priority level (please note that lower priority numbers define a higher priority). When embOS needs to perform atomic operations, embos raises the interrupt priority level of the CPU to 128. All interrupt priorities from 0 to 127 are never disabled by embOS and thus named zero latency interrupts. To ensure that the operations are still atomic, embOS functions must not be called from within zero latency interrupts.

It is not possible to raise the interrupt priority level of the CPU for Cortex-M0, M0+, M1 and M23 processors. Thus, zero latency interrupts are not available on those processors.

> **Note**
>
> Please be aware with ARM Erratum 837070, embOS sets PRIMASK before writing to BASEPRI and unconditionally clears it afterwards. Therefore, zero latency interrupts are disabled for a few cycles when embOS dis- or enables embOS interrupts. Please refer to chapter ARM erratum 837070 for more details.

6.6 Interrupt priorities

The interrupt priority is any number between 0 and 255 as seen by the CPU core. With embOS and its own setup functions for the interrupt controller and priorities, there is no difference in the priority values regardless of the different preemption level of specific devices. Using the CMSIS functions to set up interrupt priorities requires different values for the priorities. These values depend on the number of preemption levels of the specific chip. A description is found in the chapter CMSIS.

6.6.1 Interrupt priorities with Cortex-M3, M4, M7 and M33 cores

Cortex-M3, M4, M7 and M33 support up to 256 levels of programmable priority with a maximum of 128 levels of preemption. Most Cortex-M chips have fewer supported levels, for example 8, 16, 32, and so on. The chip designer can customize the chip to obtain the levels required. There is a minimum of 8 preemption levels. Every interrupt with a higher preemption level may preempt any other interrupt handler running on a lower preemption level. Interrupts with equal preemption level may not preempt each other. The interrupt priority is split into group priority and subpriority. The group priority determines the preemption level.

With introduction of zero latency interrupts, interrupt priorities usable for interrupts using embOS API functions are limited.

- Any interrupt handler using embOS API functions has to run with interrupt priorities from 128 to 255. These embOS interrupt handlers have to start with `OS_INT_Enter()` or `OS_INT_EnterNestable()` and have to end with `OS_INT_Leave()` or `OS_INT_LeaveNestable()`.
- Any zero latency interrupt (running at priorities from 0 to 127) must not call any embOS API function. Even `OS_INT_Enter()` and `OS_INT_Leave()` must not be called.

 © 2010-2025 SEGGER Microcontroller GmbH

- Interrupt handlers running at low priorities (from 128 to 255) not calling any embOS API function are allowed, but must not re-enable interrupts! The priority limit between embOS interrupts and zero latency interrupts is fixed to 128 and can only be changed by defining `OS_IPL_THRESHOLD` and recompiling the embOS libraries (or using embOS sources in your project)! This is done for efficiency reasons. The macro `OS_IPL_THRESHOLD` can be modified in OSCHIP.h or overwritten by project specific preprocessor settings. In case of doubt, please contact the embOS support.

> **Note**
>
> If you do not set an interrupt priority with `NVIC_SetPriority()` or `OS_ARM_ISRSetPrio()`, the priority after reset is `0x00`, which is not a valid embOS interrupt priority but a zero latency interrupt.

6.6.2 Interrupt priorities with Cortex-M0, M0+, M1 and M23 cores

All Cortex-M0, M0+, M1 and M23 support 4 levels of programmable priority. Priority grouping is not available. Thus, the interrupt priority equals the preemption level. Every interrupt with a higher interrupt priority may preempt any other interrupt handler running with a lower interrupt priority. Interrupts with equal priority may not preempt each other.

All interrupt handlers may call embOS API irrespective of their priority. Any interrupt handler using embOS API functions has to start with `OS_INT_Enter()` or `OS_INT_EnterNestable()` and has to end with `OS_INT_Leave()` or `OS_INT_LeaveNestable()`.

6.6.3 Priority of the embOS scheduler

The embOS scheduler runs in the PendSV handler and on the lowest interrupt priority. The scheduler may be preempted by any other interrupt with higher preemption level. The application interrupts shall run on higher preemption levels to ensure short reaction time.

During initialization, the priority of the embOS scheduler is set to `0x03` for ARMv6-M and ARMv8-M Baseline and to `0xFF` for ARMv7-M and ARMv8-M Mainline, which is the lowest preemption level regardless of the number of preemption levels.

> **Note**
>
> Applications that prevent the execution of the PendSV interrupt (e.g. by disabling interrupts via PRIMASK or via FAULTMASK on ARMv7-M, or by disabling interrupts via FAULTMASK on ARMv6-M) must not call any embOS API that would initiate a task switch.

> **Note**
>
> Applications must not change the priority of the PendSV interrupt.

6.6.4 Priority of the embOS system timer

The embOS system timer runs on the second lowest preemption level. Thus, the embOS timer may preempt the scheduler. Application interrupts which require fast reaction should run on a higher preemption level.

 © 2010-2025 SEGGER Microcontroller GmbH

6.6.5 Priority of embOS software timers

The embOS software timer callback functions are called from the scheduler and run on the schedulers preemption level which is the lowest interrupt priority level. To ensure short reaction time of other interrupts, other interrupts should run on a higher preemption level and the software timer callback functions should be as short as possible.

6.6.6 Priority of application interrupts for Cortex-M0, M0+, M1 and M23 cores

Application interrupts using embOS functions may run on any priority. We recommend that application interrupts should run on a higher preemption level than the embOS scheduler, at least at the second lowest preemption level.

6.6.7 Priority of application interrupts for Cortex-M3, M4, M7 and M33 cores

Application interrupts using embOS functions may run on any priority level between 255 to 128. Interrupt handlers which require fast reaction may run on higher priorities than 128, but must not call any embOS function (zero latency interrupts). We recommend that application interrupts should run on a higher preemption level than the embOS scheduler, at least at the second lowest preemption level.

As the number of priority levels is chip specific, the second lowest preemption level varies depending on the chip. If the number of preemption levels is not documented, the second lowest preemption level can be set as follows, using embOS functions:

```
unsigned char Priority;
OS_ARM_ISRSetPrio(OS_ISR_ID_TICK, 0xFF);
  // Set to lowest level, ALL BITS set
Priority  = OS_ARM_ISRSetPrio(OS_ISR_ID_TICK, 0xFF);  // Read priority back
Priority -= 1;                                        // Lower preemption level
OS_ARM_ISRSetPrio(OS_ISR_ID_TICK, Priority);
```

6.7 Interrupt nesting

The Cortex-M CPU uses a priority controlled interrupt scheduling which allows nesting of interrupts. Any interrupt or exception with a higher preemption level may interrupt an interrupt handler running on a lower preemption level when interrupts are enabled during execution of the interrupt service routine. An interrupt handler calling embOS functions has to start with an embOS prologue function that informs embOS that an interrupt handler is running. For any interrupt handler, the user may decide individually whether this interrupt handler may be preempted or not by choosing the prologue function `OS_INT_Enter()` or `OS_INT_EnterNestable()`. When `OS_INT_Enter()` is called it disables embOS interrupts, while `OS_INT_EnterNestable()` keeps interrupts enabled allowing the interrupt to be preempted by other interrupts with higher priority.

 © 2010-2025 SEGGER Microcontroller GmbH

6.8 Interrupt enable behavior

6.8.1 Interrupt enable behavior with Cortex-M0, M0+, M1 and M23 cores

Calling `OS_INT_Disable()` on these architectures results in *PRIMASK* being set, effectively disabling embOS interrupts. An embOS API function is called after that call to `OS_INT_Disable()` may internally disable embOS interrupts by setting *PRIMASK* and, since the embOS interrupt disable counter is not set, will subsequently re-enable embOS interrupts by clearing *PRIMASK*. Therefore, upon returning from that embOS API function embOS interrupts are enabled although the application did not call `OS_INT_Enable()`.

Since zero latency interrupts are not supported with these architectures, calling `OS_INT_DisableAll()` results in an identical behavior as calling `OS_INT_Disable()`.

6.8.2 Interrupt enable behavior with Cortex-M3, M4, M7 and M33 cores

Calling `OS_INT_Disable()` on these architectures results in *BASEPRI* being set, effectively disabling embOS interrupts. An embOS API function may internally disable embOS interrupts by writing *`OS_IPL_THRESHOLD`* to *BASEPRI* and, since the embOS interrupt disable counter is not set, will subsequently re-enable embOS interrupts by clearing *BASEPRI*. Therefore, upon returning from that embOS API function embOS interrupts are enabled although the application did not call `OS_INT_Enable()`.

Calling `OS_INT_DisableAll()` on these architectures results in *PRIMASK* being set, effectively disabling both embOS interrupts and zero latency interrupts. An embOS API function may internally disable embOS interrupts by writing *`OS_IPL_THRESHOLD`* to *BASEPRI* and, since the embOS interrupt disable counter is not set, will subsequently re-enable embOS interrupts by clearing *BASEPRI*. The value of *BASEPRI*, however, does not have any effect while *PRIMASK* is still set. Therefore, upon returning from that embOS API function both embOS interrupts and zero latency interrupts are still disabled.

 © 2010-2025 SEGGER Microcontroller GmbH

6.9 Interrupt handling

For the Cortex-M core, which has a built-in vectored interrupt controller, embOS delivers additional functions to install and setup interrupt handler functions.

This API is not available in embOS library mode `OS_LIBMODE_SAFE`.

6.9.1 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
`OS_ARM_ISRInit()`	Initializes the interrupt handling.	•	•			
`OS_ARM_InstallISRHandler()`	Sets an interrupt handler in the RAM vector table.	•	•			
`OS_ARM_EnableISR()`	Enables acceptance of a specific interrupt source.	•	•		•	•
`OS_ARM_DisableISR()`	Disables acceptance of a specific interrupt source.	•	•		•	•
`OS_ARM_ISRSetPrio()`	Sets the priority of a specific interrupt source.	•	•		•	•

 © 2010-2025 SEGGER Microcontroller GmbH

6.9.1.1 OS_ARM_ISRInit()

Description

Initializes the interrupt handling.

Prototype

```
void OS_ARM_ISRInit(OS_U32          IsVectorTableInRAM,
                    OS_U32          NumInterrupts,
                    OS_ISR_HANDLER* VectorTableBaseAddr[],
                    OS_ISR_HANDLER* RAMVectorTableBaseAddr[]);
```

Parameters

Parameter	Description
IsVectorTableInRAM	Defines the vector table location. (0 = Flash, 1 = RAM)
NumInterrupts	Number of implemented interrupts.
VectorTableBaseAddr	Flash vector table base address.
RAMVectorTableBaseAddr	RAM vector table base address.

Additional information

This function must be called before `OS_ARM_InstallISRHandler()`, `OS_ARM_EnableISR()`, `OS_ARM_DisableISR()` and `OS_ARM_ISRSetPrio()` can be called.

> **Note**
>
> Please note a RAM vector table can be used only if the device has a configurable VTOR implemented.

Example

```
void OS_InitHW(void) {
  OS_ARM_ISRInit(1u, 82, (OS_ISR_HANDLER**)__Vectors, (OS_ISR_HANDLER**)pRAMVectTable);
  OS_ARM_InstallISRHandler(OS_ISR_ID_TICK, OS_Systick);
  OS_ARM_ISRSetPrio(OS_ISR_ID_TICK, 0xE0u);
  OS_ARM_EnableISR(OS_ISR_ID_TICK);
}
```

 © 2010-2025 SEGGER Microcontroller GmbH

6.9.1.2 OS_ARM_InstallISRHandler()

Description

Sets an interrupt handler in the RAM vector table. Does nothing when vector table is in Flash.

Prototype

```
OS_ISR_HANDLER *OS_ARM_InstallISRHandler(int             ISRIndex,
                                        OS_ISR_HANDLER* pISRHandler);
```

Parameters

Parameter	Description
ISRIndex	Index of the interrupt source which should be installed. Note that the index counts from 0 for the first entry in the vector table.
pISRHandler	Address of the interrupt handler.

Return value

Previous interrupt handler address in the vector table.

Additional information

`OS_ARM_InstallISRHandler()` copies the vector table from Flash to RAM when it is called for the first time and RAM vector table is enabled.

> **Note**
>
> Please note a RAM vector table can be used only if the device has a configurable VTOR implemented.

Note that the `ISRIndex` counts from 0 for the first entry in the vector table. The first peripheral index therefore has the `ISRIndex` 16, because the first peripheral interrupt vector is located after the 16 generic vectors in the vector table. This differs from index values used with CMSIS.

Example

```
void OS_InitHW(void) {
  OS_ARM_ISRInit(1u, 82, (OS_ISR_HANDLER**)__Vectors, (OS_ISR_HANDLER**)pRAMVectTable);
  OS_ARM_InstallISRHandler(OS_ISR_ID_TICK, OS_Systick);
  OS_ARM_ISRSetPrio(OS_ISR_ID_TICK, 0xE0u);
  OS_ARM_EnableISR(OS_ISR_ID_TICK);
}
```

 © 2010-2025 SEGGER Microcontroller GmbH

6.9.1.3 OS_ARM_EnableISR()

Description

Enables acceptance of a specific interrupt source.

Prototype

```
void OS_ARM_EnableISR(int ISRIndex);
```

Parameters

Parameter	Description
ISRIndex	Index of the interrupt source which should be enabled. Note that the index counts from 0 for the first entry in the vector table.

Additional information

This function just enables the interrupt inside the interrupt controller. It does not enable the interrupt of any peripherals. This has to be done elsewhere.

Note that the ISRIndex counts from 0 for the first entry in the vector table. The first peripheral index therefore has the ISRIndex 16, because the first peripheral interrupt vector is located after the 16 generic vectors in the vector table. This differs from index values used with CMSIS.

 © 2010-2025 SEGGER Microcontroller GmbH

6.9.1.4 OS_ARM_DisableISR()

Description

Disables acceptance of a specific interrupt source.

Prototype

```
void OS_ARM_DisableISR(int ISRIndex);
```

Parameters

Parameter	Description
ISRIndex	Index of the interrupt source which should be disabled. Note that the index counts from 0 for the first entry in the vector table.

Additional information

This function just disables the interrupt in the interrupt controller. It does not disable the interrupt of any peripherals. This has to be done elsewhere.

Note that the ISRIndex counts from 0 for the first entry in the vector table. The first peripheral index therefore has the ISRIndex 16, because the first peripheral interrupt vector is located after the 16 generic vectors in the vector table. This differs from index values used with CMSIS.

 © 2010-2025 SEGGER Microcontroller GmbH

6.9.1.5 OS_ARM_ISRSetPrio()

Description

Sets the priority of a specific interrupt source.

Prototype

```
int OS_ARM_ISRSetPrio(int ISRIndex,
                      int Prio);
```

Parameters

Parameter	Description
`ISRIndex`	Index of the interrupt source which should be modified. Note that the index counts from 0 for the first entry in the vector table.
`Prio`	The priority which should be set for the specified interrupt source. `Prio` ranges from 0 (highest priority) to 255 (lowest priority).

Additional information

The priority value is independent of the chip-specific preemption levels. Any value between 0 and 255 can be used, were 255 always is the lowest priority and 0 is the highest priority. The function can be called to set the priority for all interrupt sources, regardless of whether embOS is used or not in the specified interrupt handler. Note that interrupt handlers running on priorities from 127 or higher must not call any embOS function.

Note that the `ISRIndex` counts from 0 for the first entry in the vector table. The first peripheral index therefore has the `ISRIndex` 16, because the first peripheral interrupt vector is located after the 16 generic vectors in the vector table. This differs from index values used with CMSIS.

> **Note**
>
> Please note there are Arm core specific restrictions when you must not change the exception priority. For more information, please have a look in the according Arm Architecture Reference Manual.

 © 2010-2025 SEGGER Microcontroller GmbH

Chapter 7

CMSIS

© 2010-2025 SEGGER Microcontroller GmbH

7.1 Introduction

ARM introduced the Cortex Microcontroller Software Interface Standard (CMSIS) as a vendor independent hardware abstraction layer for simplifying software re-use. The standard enables consistent and simple software interfaces to the processor, for peripherals, for real time operating systems as embOS and other middleware. As SEGGER is one of the CMSIS partners, embOS for Cortex-M is fully CMSIS compliant. embOS comes with a generic CMSIS start project which should run on any Cortex-M3 CPU. All other start projects, even those not based on CMSIS, are also fully CMSIS compliant and can be used as starting points for CPU specific CMSIS projects. How to use the generic project and adding vendor specific files to this or other projects is explained in the following chapters.

 © 2010-2025 SEGGER Microcontroller GmbH

7.2 The generic CMSIS start project

The folder `Start\BoardSupport\CMSIS` contains a generic CMSIS start project that should run on any ARMv7-M core. The subfolder `DeviceSupport\` contains the device specific source and header files which have to be replaced by the device specific files of the vendor to make the CMSIS sample start project device specific.

7.3 Device specific files needed for embOS with CMSIS

- **`Device.h`**: Contains the device specific exception and interrupt numbers and names. embOS needs the Cortex-M generic exception numbers `PendSV_IRQn` and `SysTick_IRQn`, as well as the exception names `PendSV_Handler` and `SysTick_Handler`, which are vendor independent and common for all devices. The sample file delivered with embOS does not contain any peripheral interrupt vector numbers and names as those are not needed by embOS. To make the embOS CMSIS sample device specific and allow usage of peripheral interrupts, this file has to be replaced by the one which is delivered from the CPU vendor.
- **`System_Device.h`**: Declares at least the two required system timer functions which are used to initialize the CPU clock system and one variable which allows the application software to retrieve information about the current CPU clock speed. The names of the clock controlling functions and variables are defined by the CMSIS standard and are therefore identical in all vendor specific implementations.
- **`System_Device.c`**: Implements the core specific functions to initialize the CPU, at least to initialize the core clock. The sample file delivered with embOS contains empty dummy functions and has to be replaced by the vendor specific file which contains the initialization functions for the core.
- **`Startup_Device.s`**: The startup file which contains the initial reset sequence and contains exception handler and peripheral interrupt handler for all interrupts. The handler functions are declared weak, so they can be overwritten by the application which implements the application specific handler functionality. The sample which comes with embOS only contains the generic exception vectors and handler and has to be replaced by the vendor specific startup file.

Startup code requirements:

The reset handler must call the **`SystemInit()`** function which is delivered with the core specific system functions. When using an ARMv7 CPU which may have a VFP floating point unit equipped, please ensure that the reset handler activates the VFP and VFP support is selected in the project options. When VFP support is not selected, the VFP should not be switched on. Otherwise, the `SystemInit()` function delivered from the device vendor should also honor the project settings and enable the VFP or keep it disabled according the project settings. Using CMSIS compliant startup code from the chip vendors may require modification if it enables the VFP unconditionally.

7.4 Device specific functions/variables needed for embOS with CMSIS

The embOS system timer is triggered by the Cortex-M generic system timer. The correct core clock and pll system is device specific and has to be initialized by a low level init function called from the startup code. embOS calls the CMSIS function `SysTick_Config()` to set up the system timer. The function relies on the correct core clock initialization performed by the low level initialization function `SystemInit()` and the value of the core clock frequency which has to be written into the `SystemCoreClock` variable during initialization or after calling `SystemCoreClockUpdate()`.

- **`SystemInit()`**: The system init function is delivered by the vendor specific CMSIS library and is normally called from the reset handler in the startup code. The system init

 © 2010-2025 SEGGER Microcontroller GmbH

function has to initialize the core clock and has to write the CPU frequency into the global variable SystemCoreClock.

- **SystemCoreClock**: Contains the current system core clock frequency and is initialized by the low level initialization function `SystemInit()` during startup. embOS for CMSIS relies on the value in this variable to adjust its own timer and all time related functions. Any other files or functions delivered with the vendor specific CMSIS library may be used by the application, but are not required for embOS.

7.5 CMSIS generic functions needed for embOS with CMSIS

The embOS system timer is triggered by the Cortex-M generic system timer which has to be initialized to generate periodic interrupts in a specified interval. The configuration function `SysTick_Config()` for the system timer relies on correct initialization of the core clock system which is performed during startup.

- **SystemCoreClockUpdate()**: This CMSIS function has to update the `SystemCoreClock` variable according the current system timer initialization. The function is device specific and may be called before the SystemCoreClock variable is accessed or any function which relies on the correct setting of the system core clock variable is called. embOS calls this function during the hardware initialization function `OS_InitHW()` before the system timer is initialized.
- **SysTick_Config()**: This CMSIS generic function is declared an implemented in the `core_cm*.h` file. It initializes and starts the SysTick counter and enables the SysTick interrupt. For embOS it is recommended to run the SysTick interrupt at the second lowest preemption priority. Therefore, after calling the `SysTick_Config()` function from `OS_InitHW()`, the priority is set to the second lowest preemption priority ba a call of `NVIC_SetPriority()`. The embOS function `OS_InitHW()` has to be called after initialization of embOS during main and is implemented in the `RTOSInit*.c` file.
- **SysTick_Handler()**: The embOS timer interrupt handler, called periodically by the interrupt generated from the SysTick timer. The `SysTick_Handler` is declared weak in the CMSIS startup code and is replaced by the embOS `Systick_Handler` function implemented in `RTOSInit*.c` which comes with the embOS start project.
- **PendSV_Handler()**: The embOS scheduler entry function. It is declared weak in the CMSIS startup code and is replaced by the embOS internal function contained in the embOS library. The embOS initialization code enables the `PendSV` exception and initializes the priority. The application **MUST NOT** change the `PendSV` priority.

7.6 Customizing the embOS CMSIS generic start project

The embOS CMSIS generic start project should run on every ARMv7-M CPU. As the generic device specific functions delivered with embOS do not initialize the core clock system and the PLL, the timing is not correct, a real CPU will run very slow. To run the sample project on a specific CPU, replace all files in the `DeviceSupport\` folder by the versions delivered by the CPU vendor. The vendor and CPU specific files should be found in the CMSIS release package, or are available from the core vendor. No other changes are necessary on the start project or any other files.

To run the generic CMSIS start project on an ARMv6-M, you have to replace the embOS libraries with libraries for ARMv6-M and have to add the specific vendor files.

7.7 Adding CMSIS to other embOS start projects

All CPU specific start projects are fully CMSIS compatible. If required or wanted in the application, the CMSIS files for the specific CPU may be added to the project without any modification on existing files. Note that the `OS_InitHW()` function in the RTOSInit file ini-

 © 2010-2025 SEGGER Microcontroller GmbH

tialize the core clock system and pll of the specific CPU. The system clock frequency and core clock frequency are defined in the RTOSInit file. If the application needs access to the `SystemCoreClock`, the core specific CMSIS startup code and core specific initialization function `SystemInit` has to be included in the project. In this case, `OS_InitHW()` function in RTOSInit may be replaced, or the CMSIS generic `RTOSInit_CMSIS.c` file may be used in the project.

7.7.1 Differences between embOS projects and CMSIS

Several embOS start projects are not based on CMSIS but are fully CMSIS compliant and can be mixed with CMSIS libraries from the device vendors. Switching from embOS to CMSIS, or mixing embOS with CMSIS functions is possible without problems, but may require some modification when the interrupt controller setup functions from CMSIS shall be used instead of the embOS functions.

7.7.1.1 Different peripheral ID numbers

Using CMSIS, the peripheral IDs to setup the interrupt controller start from 0 for the first peripheral interrupt. With embOS, the first peripheral is addressed with ID number 16. embOS counts the first entry in the interrupt vector table from 0, so, the first peripheral interrupt following the 16 Cortex system interrupt entries, is 16. When the embOS functions should be replaced by the CMSIS functions, this correction has to be taken into account, or if available, the symbolic peripheral id numbers from the CPU specific CMSIS device header file may be used with CMSIS. Note that using these IDs with the embOS functions will work only, when 16 is added to the IDs from the CMSIS device header files.

7.7.1.2 Different interrupt priority values

Using embOS functions, the interrupt priority value ranges from 0 to 255 and is written into the NVIC control registers as is, regardless of the number of implemented priority bits. 255 is the lowest priority, 0 is the highest priority. Using CMSIS, the range of interrupt priority levels used to setup the interrupt controller depends on the number of priority bits implemented in the specific CPU. The number of priority bits for the specific device shall be defined in the device specific CMSIS header file as `__NVIC_PRIO_BITS`. If it is not defined in the device specific header files, a default of 4 is set in the generic CMSIS core header file. A CPU with 4 priority bits supports up to 16 preemption levels. With CMSIS, the range of interrupt priorities for this CPU would be 0 to 15, where 0 is the highest priority and 15 is the lowest. To convert an embOS priority value into a value for the CMSIS functions, the value has to be shifted to the right by (`8 - __NVIC_PRIO_BITS`). To convert an CMSIS value for the interrupt priority into the value used with the embOS functions, the value has to be shifted to the left by (`8 - __NVIC_PRIO_BITS`). In any case, half of the priorities with lower values (from zero) are high priorities which must not be used with any interrupt handler using embOS functions.

 © 2010-2025 SEGGER Microcontroller GmbH

7.8 Interrupt and exception handling with CMSIS

The embOS CPU specific projects come with CPU specific vector tables and empty exception and interrupt handlers for the specific CPU. All handlers are named according the names of the CMSIS device specific handlers and are declared weak and can be replaced by an implementation in the application source files. The CPU specific vector table and interrupt handler functions in the embOS start projects can be replaced by the CPU specific CMSIS startup file of the CPU vendor without any modification on other files in the project. embOS uses the two Cortex-M generic exceptions PendSV and SysTick and delivers its own handler functions to handle these exceptions. All peripheral interrupts are device specific and are not used with embOS except for profiling support and system analysis with embOSView using a UART.

7.8.1 Enable and disable interrupts

The generic CMSIS functions `NVIC_EnableIRQ()` and `NVIC_DisableIRQ()` can be used instead of the embOS functions `OS_ARM_EnableISR()` and `OS_ARM_DisableISR()` functions. Note that the CMSIS functions use different peripheral ID indices to address the specific interrupt number. embOS counts from 0 for the first entry in the interrupt vector table, CMSIS counts from 0 for the first peripheral interrupt vector, which is ID number 16 for the embOS functions. About these differences, please refer to *Different peripheral ID numbers* on page 54. To enable and disable interrupts in general, the embOS functions `OS_INT_IncDI()` and `OS_INT_DecRI()` or other embOS functions described in the generic embOS manual should be used instead of the intrinsic functions from the CMSIS library.

7.8.2 Setting the Interrupt priority

With CMSIS, the CMSIS generic function `NVIC_SetPriority()` can be used instead of the `OS_ARM_ISRSetPrio()` function. Note that with the CMSIS function, the range of valid interrupt priority values depends on the number of priority bits defined and implemented for the specific device. The number of priority bits for the specific device shall be defined in the device specific CMSIS header file as `__NVIC_PRIO_BITS`. If it is not defined in the device specific header files, a default of 4 is set in the generic CMSIS core header file. A CPU with 4 priority bits supports up to 16 preemption levels. With CMSIS, the range of interrupt priorities for this CPU would be 0 to 15, where 0 is the highest priority and 15 is the lowest. About interrupt priorities in an embOS project, please refer to *Interrupt priorities* on page 40 and *Interrupt nesting* on page 42, about the differences between interrupt priority and ID values used to setup the NVIC controller, please refer to *Different interrupt priority values* on page 54.

 © 2010-2025 SEGGER Microcontroller GmbH

Chapter 8

Floating Point (FP) support

© 2010-2025 SEGGER Microcontroller GmbH

8.1 ARM Floating-point Extension

Some Cortex-M4, Cortex-M7 and Cortex-M33 processors implement the ARMv7-M/ARMv8-M Floating-point Extension, providing a Floating Point Unit (FPU).

When selecting such CPU and activating floating-point support in the IDE's project options, the compiler and linker will generate efficient code that uses the FPU when floating-point calculations are performed in the application. With embOS, the FPU registers are automatically saved and restored during preemptive and cooperative task switches. For efficiency reasons, embOS does not save and restore the FPU registers for tasks that did not use the FPU.

8.2 Using embOS libraries with floating-point support

When floating-point support is selected as project option, an embOS library with floating-point support must be used in the project. embOS libraries with floating-point support require that the FPU is switched on during startup and remains switched on during program execution. When using a customized startup code, ensure that the FPU is switched on during startup and that the ASPEN and LSPEN bits of the Floating-point Context Control Register (FPCCR) are not cleared (their reset value is 1 and embOS expects them to remain set).

In OS_Init(), a debug build of embOS checks whether the FPU was switched on and the FPCCR.ASPEN and FPCCR.LSPEN bits are set: If any of these conditions is not met, embOS calls OS_Error() with error code OS_ERR_FPU_NOT_ENABLED.

8.3 Using the FPU in interrupt service routines

Using the FPU in interrupt service routines does not require any additional functions in order to save and restore the FPU registers, since these are automatically saved and restored by hardware.

8.4 FPU default behavior

The behavior of the ARM FPU is controlled by different flags in the Floating-point Status and Control Register (FPSCR). Each time a new floating-point context is generated, the FPSCR is loaded with default values stored in the Floating-point Default Status and Control Register (FPDSCR). The FPDSCR is initialized in OS_Init() using the value 0x02000000, thereby setting the Default NaN mode control bit to 1. If a different default FPU behavior is desired, FPDSCR may be modified after OS_Init() was executed.

 © 2010-2025 SEGGER Microcontroller GmbH

8.5 GCC FP compiler options

The GCC compiler uses the compiler option `-mfloat-abi=<name>` to specify which floating-point ABI to use. Permissible values are `soft`, `softfp`, and `hard`.

ABI	Description
`soft`	The compiler will not generate any hardware FPU instructions and instead call library functions for floating-point operations. Function calls are generated by passing floating-point arguments in integer registers.
`softfp`	The compiler will generate hardware FPU instructions for floating-point operations. Function calls are generated by passing floating-point arguments in integer registers.
`hard`	The compiler will generate hardware FPU instructions for floating-point operations. Function calls are generated by passing floating-point arguments in FPU registers.

With embOS object code, please ensure the library in use matches the configured ABI for your project.

 © 2010-2025 SEGGER Microcontroller GmbH

Chapter 9

RTT and SystemView

© 2010-2025 SEGGER Microcontroller GmbH

9.1 SEGGER Real Time Transfer

With SEGGER's Real Time Transfer (RTT) it is possible to output information from the target microcontroller as well as sending input to the application at a very high speed without affecting the target's real time behavior. SEGGER RTT can be used with any J-Link model and any supported target processor which allows background memory access.

RTT is included with many embOS start projects. These projects are by default configured to use RTT for debug output. Some IDEs, such as SEGGER Embedded Studio, support RTT and display RTT output directly within the IDE. In case the used IDE does not support RTT, SEGGER's J-Link RTT Viewer, J-Link RTT Client, and J-Link RTT Logger may be used instead to visualize your application's debug output.

For more information on SEGGER Real Time Transfer, refer to segger.com/jlink-rtt.

9.2 SEGGER SystemView

SEGGER SystemView is a real-time recording and visualization tool to gain a deep understanding of the runtime behavior of an application, going far beyond what debuggers are offering. The SystemView module collects and formats the monitor data and passes it to RTT.

SystemView is included with many embOS start projects. These projects are by default configured to use SystemView in debug builds. The associated PC visualization application, SystemView, is not shipped with embOS. Instead, the most recent version of that application is available for download from our website.

SystemView is initialized by calling `SEGGER_SYSVIEW_Conf()` on the target microcontroller. This call is performed within `OS_InitHW()` of the respective `RTOSInit*.c` file. As soon as this function was called, the connection of the SystemView desktop application to the target can be started. In order to remove SystemView from the target application, remove the `SEGGER_SYSVIEW_Conf()` call, the `SEGGER_SYSVIEW.h` include directive as well as any other reference to `SEGGER_SYSVIEW_*` like `SEGGER_SYSVIEW_TickCnt`.

For more information on SEGGER SystemView and the download of the SystemView desktop application, refer to segger.com/systemview.

> **Note**
>
> SystemView uses embOS timing API to get at start the current system time. This requires that `OS_TIME_ConfigSysTimer()` was called before `SEGGER_SYSVIEW_Start()` is called or the SystemView PC application is started.

 © 2010-2025 SEGGER Microcontroller GmbH

Chapter 10

embOS Thread Script

© 2010-2025 SEGGER Microcontroller GmbH

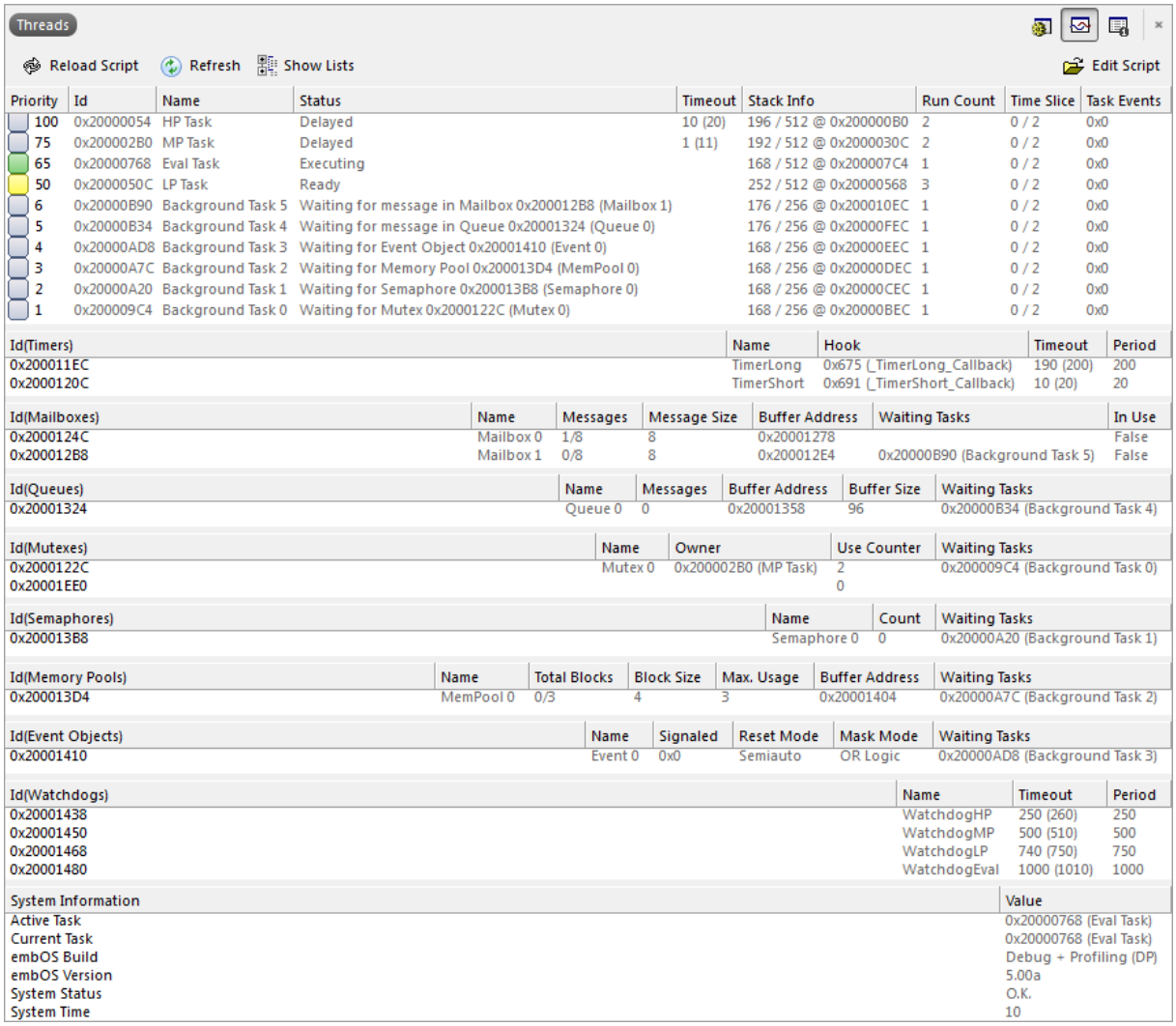
10.1 Introduction

A thread script is included with every board support package shipped with embOS. This script may be used to display various information about the system, the tasks and created embOS objects like timers, mailboxes, queues, semaphores, memory pools, events and watchdogs.

When creating a custom project, the thread script may be added to the respective project's options (`"Debug" -> "Debugger" -> "Threads Script File"`).

10.2 How to use it

To enable the threads window, click on View in the menu bar and choose the option Threads in the sub-menu `More Debug Windows`. Alternatively, the threads window may also be enabled by pressing `[Ctrl + Alt + H]`. The object lists and system information within the threads window can be enabled or disabled via the `Show Lists` dropdown menu. The threads window gets updated every time the application is halted. It should closely resemble the screenshot below:



Some of this information is available in debug builds of embOS only. Using other builds, the respective entries will show `"n.a."` to indicate this.

 © 2010-2025 SEGGER Microcontroller GmbH

10.2.1 Task List

Priority	Id	Name	Status	Timeout	Stack Info	Run Count	Time Slice	Task Events
100	0x20000054	HP Task	Delayed	10 (20)	196 / 512 @ 0x200000B0	2	0 / 2	0x0
75	0x200002B0	MP Task	Delayed	1 (11)	192 / 512 @ 0x2000030C	2	0 / 2	0x0
65	0x20000768	Eval Task	Executing		168 / 512 @ 0x200007C4	1	0 / 2	0x0
50	0x2000050C	LP Task	Ready		252 / 512 @ 0x20000568	3	0 / 2	0x0
6	0x20000B90	Background Task 5	Waiting for message in Mailbox 0x200012B8 (Mailbox 1)		176 / 256 @ 0x200010EC	1	0 / 2	0x0
5	0x20000B34	Background Task 4	Waiting for message in Queue 0x20001324 (Queue 0)		176 / 256 @ 0x20000FEC	1	0 / 2	0x0
4	0x20000AD8	Background Task 3	Waiting for Event Object 0x20001410 (Event 0)		168 / 256 @ 0x20000EEC	1	0 / 2	0x0
3	0x20000A7C	Background Task 2	Waiting for Memory Pool 0x200013D4 (MemPool 0)		168 / 256 @ 0x20000DEC	1	0 / 2	0x0
2	0x20000A20	Background Task 1	Waiting for Semaphore 0x200013B8 (Semaphore 0)		168 / 256 @ 0x20000CEC	1	0 / 2	0x0
1	0x200009C4	Background Task 0	Waiting for Mutex 0x2000122C (Mutex 0)		168 / 256 @ 0x20000BEC	1	0 / 2	0x0

The task list displays various information about the running tasks:

Column	Description
Priority	This is the priority of the task
Id	The address of a tasks task control block
Name	The name of the task
Status	The current status of the task
Timeout	Time in ms till the task gets called again
Stack Info	Shows the maximum usage (left) of the total stack for this task (right) in bytes
Run Count	Shows how many times the task has been started since the last reset
Time Slice	Show the number of remaining and maximum time slices if round robin scheduling is available
Task Events	Show the event mask of a task

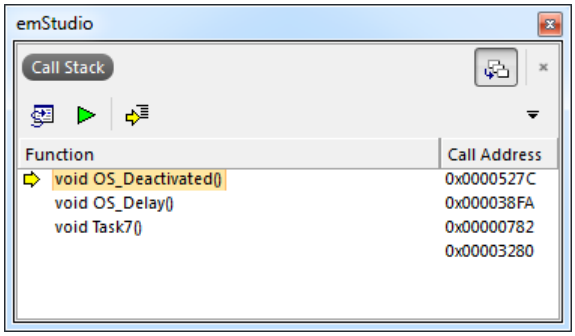
> **Note**
>
> By default the thread script is limited to display a total of 25 tasks only. This limit may be changed inside the respective project's options (`“Debug” -> “Debugger” -> “Thread Maximum”`).

 © 2010-2025 SEGGER Microcontroller GmbH

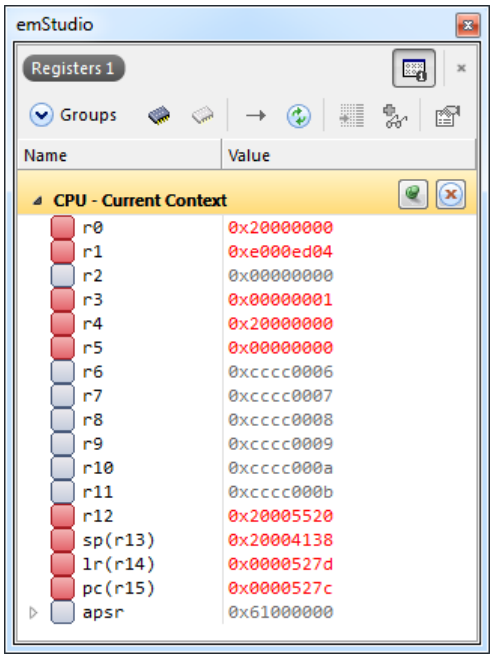
10.2.2 Task sensitivity

In addition to the information displayed in the threads list, the threads script furthermore allows for the investigation of the register contents and the call stack of inactive tasks. To display this information, double click the entry of the respective task in the threads window. The register window and the call stack window will subsequently be updated to display information about the chosen task's state. To view this information, the call stack and the register window have to be enabled.

After double clicking the inactive task, the call stack window shows the last function that has been called by this task:



Also, the register window gets updated and shows the register contents of the inactive task:



10.2.3 Timers

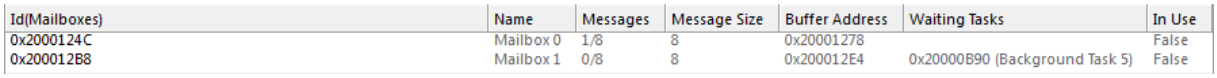
Id(Timers)	Name	Hook	Timeout	Period
0x200011EC	TimerLong	0x675 (_TimerLong_Callback)	190 (200)	200
0x2000120C	TimerShort	0x691 (_TimerShort_Callback)	10 (20)	20

The timers list displays various information about active timers:

Column	Description
Id(Timers)	The timer's address
Name	If available, the respective object identifier is shown here
Hook	The function address that is called after the timeout
Timeout	The time delay and the point in time, when the timer finishes waiting
Period	The time period the timer runs

 © 2010-2025 SEGGER Microcontroller GmbH

10.2.4 Mailboxes

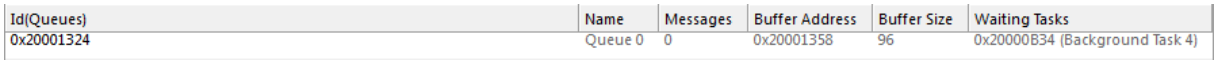


Id(Mailboxes)	Name	Messages	Message Size	Buffer Address	Waiting Tasks	In Use
0x2000124C	Mailbox 0	1/8	8	0x20001278		False
0x200012B8	Mailbox 1	0/8	8	0x200012E4	0x20000B90 (Background Task 5)	False

The mailboxes list displays various information about used mailboxes:

Column	Description
Id(Mailboxes)	The mailbox's address
Name	If available, the respective object identifier is shown here
Messages	The number of messages in a mailbox and the maximum number of messages the mailbox can hold
Message Size	The size of an individual message in bytes
Buffer Address	The message buffer address
Waiting Tasks	The list of tasks that are waiting for the mailbox (address and, if available, name)

10.2.5 Queues

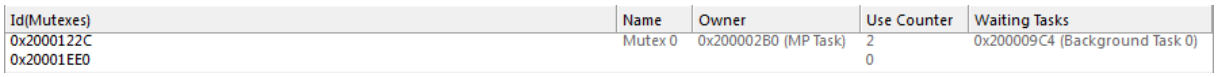


Id(Queues)	Name	Messages	Buffer Address	Buffer Size	Waiting Tasks
0x20001324	Queue 0	0	0x20001358	96	0x20000B34 (Background Task 4)

The queues list displays various information about used queues:

Column	Description
Id(Queues)	The queue's address
Name	If available, the respective object identifier is shown here
Messages	The number of messages in a queue
Buffer Address	The message buffer address
Buffer Size	The size of the message buffer in bytes
Waiting Tasks	The list of tasks that are waiting for the queue (address and, if available, name)

10.2.6 Mutexes



Id(Mutexes)	Name	Owner	Use Counter	Waiting Tasks
0x2000122C	Mutex 0	0x200002B0 (MP Task)	2	0x200009C4 (Background Task 0)
0x20001EE0			0	

The mutexes list displays various information about used mutexes:

Column	Description
Id(Mutexes)	The mutexes' address
Name	If available, the respective object identifier is shown here
Owner	The address and name of the owner task
Use Counter	Counts the number of times the mutex was claimed
Waiting Tasks	The list of tasks that are waiting for the mutex (address and, if available, name)

 © 2010-2025 SEGGER Microcontroller GmbH

10.2.7 Semaphores

Id(Semaphores)	Name	Count	Waiting Tasks
0x200013B8	Semaphore 0	0	0x20000A20 (Background Task 1)

The semaphores list displays various information about used semaphores:

Column	Description
Id(Semaphores)	The semaphores' address
Name	If available, the respective object identifier is shown here
Count	Counts how often this semaphore can be claimed
Waiting Tasks	The list of tasks that are waiting for the semaphore (address and, if available, name)

10.2.8 Readers-writer lock

RW Lock	Name	Status	Max. number of tokens	Tokens left
0x200009AC	RW Lock 0	Unlocked	2	1

The readers-writer lock list displays various information about used readers-writer locks:

Column	Description
Id(RW Lock)	The readers-writer locks address
Name	If available, the respective object identifier is shown here
Status	If all tokens are taken the readers-writer lock is locked. Otherwise it is unlocked.
Max. number of tokens	The maximum numbers of token which were defined when the readers-writer lock was created.
Tokens left	The number of available tokens.

10.2.9 Memory Pools

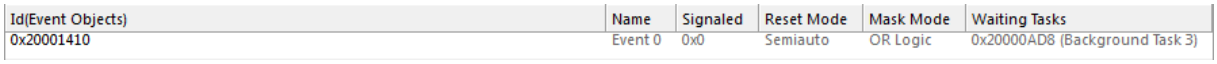
Id(Memory Pools)	Name	Total Blocks	Block Size	Max. Usage	Buffer Address	Waiting Tasks
0x200013D4	MemPool 0	0/3	4	3	0x20001404	0x20000A7C (Background Task 2)

The memory pools list displays various information about used memory pools:

Column	Description
Id(Memory Pools)	The memory pool's address
Name	If available, the respective object identifier is shown here
Total Blocks	Shows the available blocks and the maximal number of blocks
Block Size	Shows the size of a single memory block
Max. Usage	Shows the maximal count of blocks which were simultaneously allocated
Buffer Address	The address of the memory pool buffer
Waiting Tasks	The list of tasks that are waiting for free memory blocks (address and, if available, name)

 © 2010-2025 SEGGER Microcontroller GmbH

10.2.10 Event Objects



Id(Event Objects)	Name	Signaled	Reset Mode	Mask Mode	Waiting Tasks
0x20001410	Event 0	0x0	Semiauto	OR Logic	0x20000AD8 (Background Task 3)

The event objects list displays various information about used event objects:

Column	Description
Id(Event Objects)	The event object's address
Name	If available, the respective object identifier is shown here
Signaled	The hexadecimal value of the bit mask containing the signaled event bits
Reset Mode	The event object's reset mode
Mask Mode	The current mask mode indicating whether OR or AND logic is used to check if a task shall resume
Waiting Tasks	The list of tasks that are waiting for the event object (address and, if available, name)

10.2.11 Watchdogs

Id(Watchdogs)	Name	Timeout	Period
0x20001438	WatchdogHP	250 (260)	250
0x20001450	WatchdogMP	500 (510)	500
0x20001468	WatchdogLP	740 (750)	750
0x20001480	WatchdogEval	1000 (1010)	1000

The watchdogs list displays various information about used watchdogs:

Column	Description
Id(Watchdogs)	The watchdog's address
Name	If available, the respective object identifier is shown here
Timeout	The remaining time (and the system time in parentheses) until the watchdog has to be fed
Period	The period in which the watchdog has to be fed

10.2.12 System Information

The system information list displays various information about embOS.

System Information	Value
Active Task	0x20000768 (Eval Task)
Current Task	0x20000768 (Eval Task)
embOS Build	Debug + Profiling (DP)
embOS Version	5.00a
System Status	O.K.
System Time	10

 © 2010-2025 SEGGER Microcontroller GmbH

Chapter 11

Technical data

© 2010-2025 SEGGER Microcontroller GmbH

11.1 Resource Usage

The memory requirements of embOS (RAM and ROM) differs depending on the used features, CPU, compiler, and library model. The following values are measured using embOS library mode `OS_LIBMODE_XR`.

Module	Memory type	Memory requirements
embOS kernel	ROM	~1700 bytes
embOS kernel	RAM	~136 bytes
Task control block	RAM	36 bytes
Software timer	RAM	20 bytes
Task event	RAM	0 bytes
Event object	RAM	12 bytes
Mutex	RAM	16 bytes
Semaphore	RAM	8 bytes
RWLock	RAM	28 bytes
Mailbox	RAM	24 bytes
Queue	RAM	32 bytes
Watchdog	RAM	12 bytes
Fixed Block Size Memory Pool	RAM	32 bytes

 © 2010-2025 SEGGER Microcontroller GmbH