embOS-MPU

Real-Time Operating System

CPU & Compiler specifics
for Cortex-M using GCC

Document: UM01075
Software Version: 5.18.0.1
Revision: 0
Date: October 19, 2022

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

Trademarks

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: October 19, 2022

Software	Revision	Date	By	Description
5.18.0.1	0	221019	MM	New software version.
5.18.0.0	0	220923	MM	Chapter “MPU Support” updated.
5.16.0.0	0	211201	TS/MM	Initial version

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.

Table of contents

Chapter 1

Using embOS

1.1 Installation

This chapter describes how to start 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, proceed as follows:

Extract the zip-file to any folder of your choice, preserving the directory structure of this file. Keep all files in their respective sub directories. Make sure the files are not read only after copying.

> **Note**
>
> The BSP projects at `/Start/BoardSupport/<DeviceManufacturer>/<Device>` assume that the `/Start/Lib` and `Start//Inc` folders are located relative to the BSP folder. If you copy a BSP folder to another location, you will need to adjust these paths in the project.

Assuming that you are using an IDE to develop your application, no further installation steps are required. You will find many prepared sample start projects, which you should use and modify to write your application. So follow the instructions of section *First Steps* on page 11.

You should do this even if you do not intend to use the IDE for your application development to become familiar with embOS.

If you do not or do not want to work with the IDE, you should: Copy either all or only the library-file that you need to your work-directory. The advantage is that when switching to an updated version of embOS later in a project, you do not affect older projects that use embOS, too. embOS does in no way rely on an IDE, it may be used without the IDE using batch files or a make utility without any problem.

1.2 First Steps

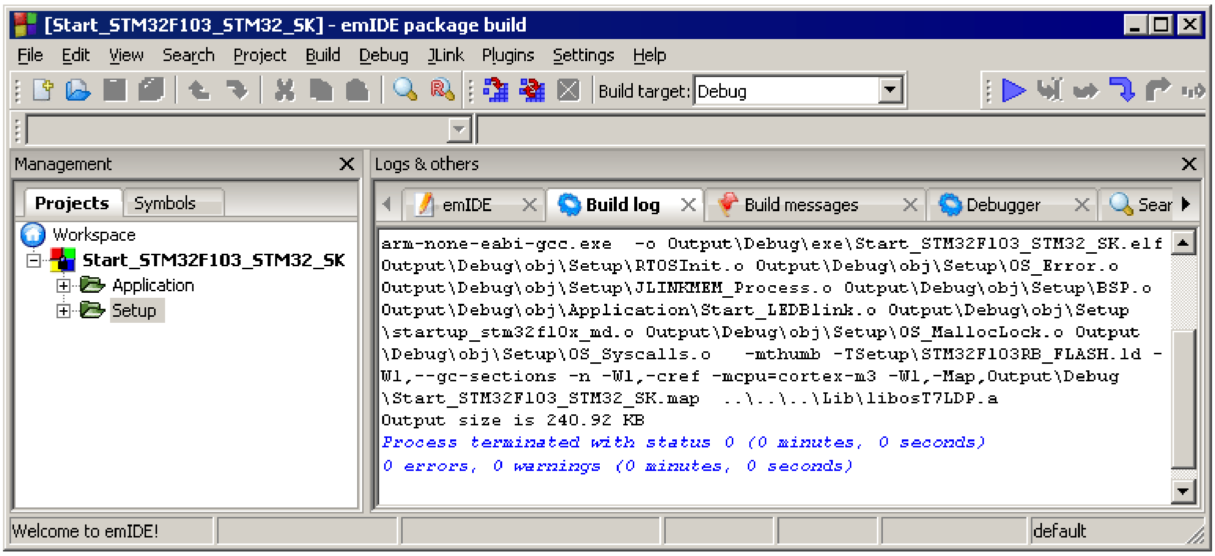
After installation of embOS you can create your first multitasking application. You have received several ready to go sample start workspaces and projects and every other files needed in the subfolder `Start`. It is a good idea to use one of them as a starting point for all of your applications. The subfolder `BoardSupport` contains the workspaces and projects which are located in manufacturer- and CPU-specific subfolders.

To start with, you may use any project from `BoardSupport` subfolder.

To get your new application running, you should proceed as follows:

- Create a work directory for your application, for example `c:\work`.
- Copy the whole folder `Start` which is part of your embOS distribution into your work directory.
- Clear the read-only attribute of all files in the new `Start` folder.
- Open one sample workspace/project in `Start\BoardSupport\<DeviceManufacturer>\<CPU>` with your IDE (for example, by double clicking it).
- Build the project. It should be built without any error or warning messages.

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



```
arm-none-eabi-gcc.exe  -o Output\Debug\exe\Start_STM32F103_STM32_SK.elf
Output\Debug\obj\Setup\RTOSInit.o Output\Debug\obj\Setup\OS_Error.o
Output\Debug\obj\Setup\JLINKMEM_Process.o Output\Debug\obj\Setup\BSP.o
Output\Debug\obj\Application\Start_LEDBlink.o Output\Debug\obj\Setup
\startup_stm32f10x_md.o Output\Debug\obj\Setup\OS_MallocLock.o Output
\Debug\obj\Setup\OS_Syscalls.o   -mthumb -TSetup\STM32F103RB_FLASH.ld -
Wl,--gc-sections -n -Wl,-cref -mcpu=cortex-m3 -Wl,-Map,Output\Debug
\Start_STM32F103_STM32_SK.map  ..\..\..\Lib\libosT7LDP.a
Output size is 240.92 KB
Process terminated with status 0 (0 minutes, 0 seconds)
0 errors, 0 warnings (0 minutes, 0 seconds)
```

For additional information you should open the ReadMe.txt file which is part of every specific project. The ReadMe file describes the different configurations of the project and gives additional information about specific hardware settings of the supported eval boards, if required.

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. (Note that the file actually shipped with your port of embOS may look slightly different from this one.)

What happens is easy to see:

After initialization of embOS two tasks are created and started. The two tasks are activated and execute until they run into the delay, then suspend for the specified time and 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
          a 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 ****************************/
```

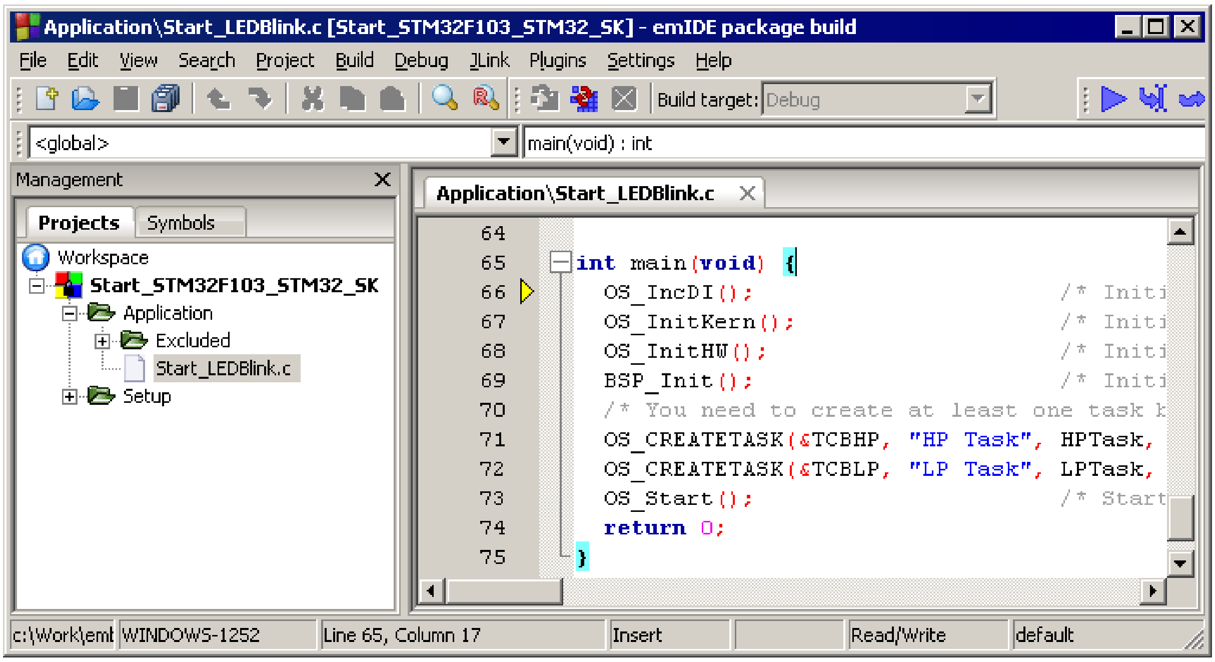
1.4 Stepping through the sample application

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.

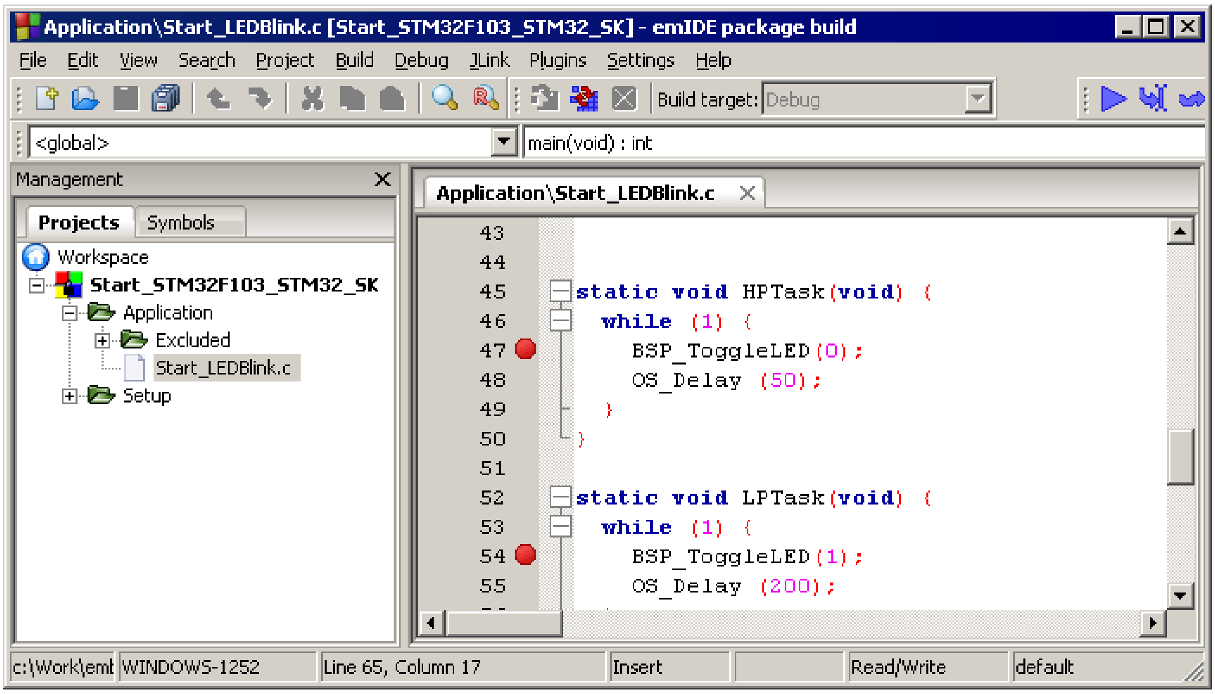
`OS_InitHW()` is part of `RTOSInit.c` and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the system tick interrupt for embOS. Step through it to see what is done.

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



```
int main(void) {
  OS_IncDI();                             /* Initi
  OS_InitKern();                          /* Initi
  OS_InitHW();                            /* Initi
  BSP_Init();                             /* Initi
  /* You need to create at least one task k
  OS_CREATETASK(&TCBHP, "HP Task", HPTask,
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,
  OS_Start();                             /* Start
  return 0;
}
```

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



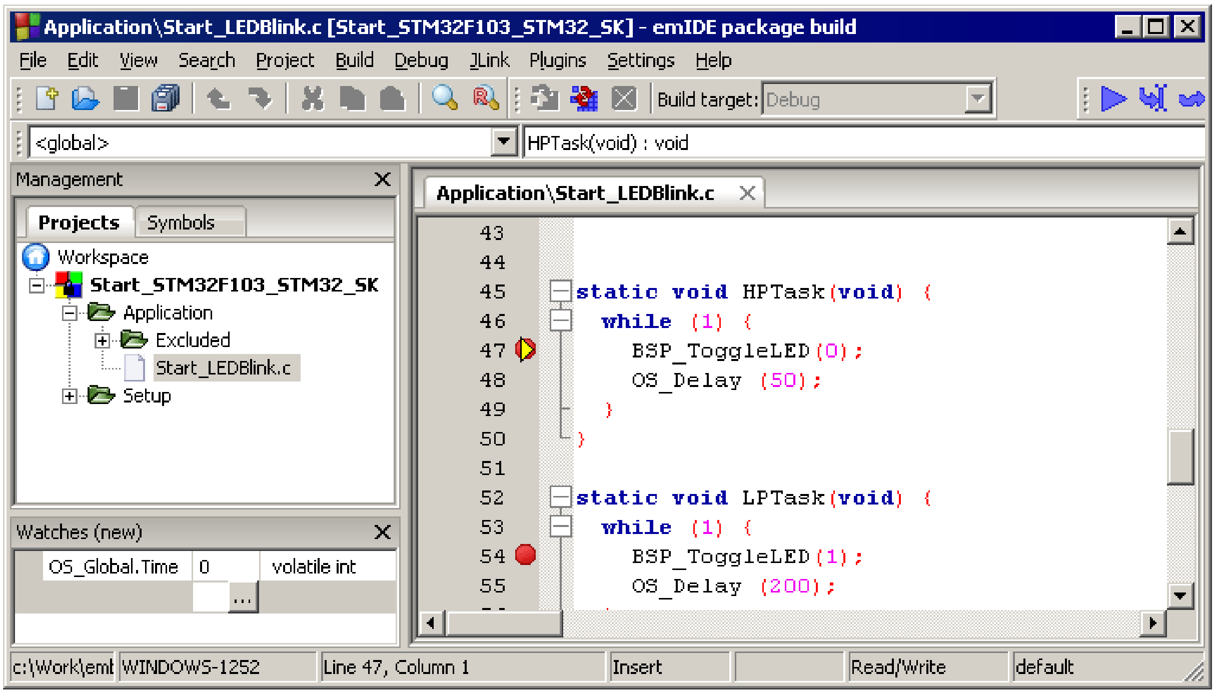
```


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

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
```

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

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

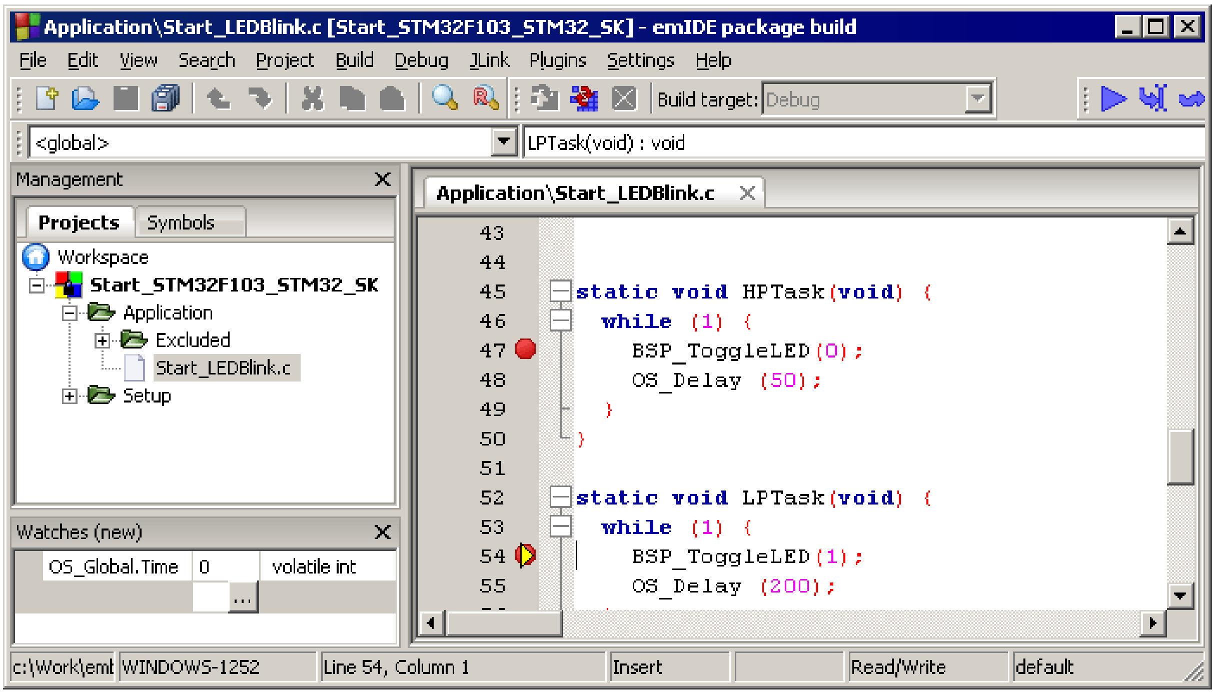

Application\Start_LEDBlink.c [Start_STM32F103_STM32_SK] - emIDE package build
File Edit View Search Project Build Debug JLink Plugins Settings Help
Build target: Debug
<global>
HPTask(void) : void
Management
Projects Symbols
Workspace
Start_STM32F103_STM32_SK
Application
Excluded
Start_LEDBlink.c
Setup
Application\Start_LEDBlink.c
Watches (new)
c:\Work\emt WINDOWS-1252 Line 47, Column 1 Insert Read/Write default

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

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
```

OS_Global.Time	0	volatile int

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


Application\Start_LEDBlink.c [Start_STM32F103_STM32_SK] - emIDE package build
File Edit View Search Project Build Debug JLink Plugins Settings Help
Build target: Debug
<global>
LPTask(void) : void
Management
Projects Symbols
Workspace
Start_STM32F103_STM32_SK
Application
Excluded
Start_LEDBlink.c
Setup
Application\Start_LEDBlink.c
Watches (new)
c:\Work\emt WINDOWS-1252 Line 54, Column 1 Insert Read/Write default

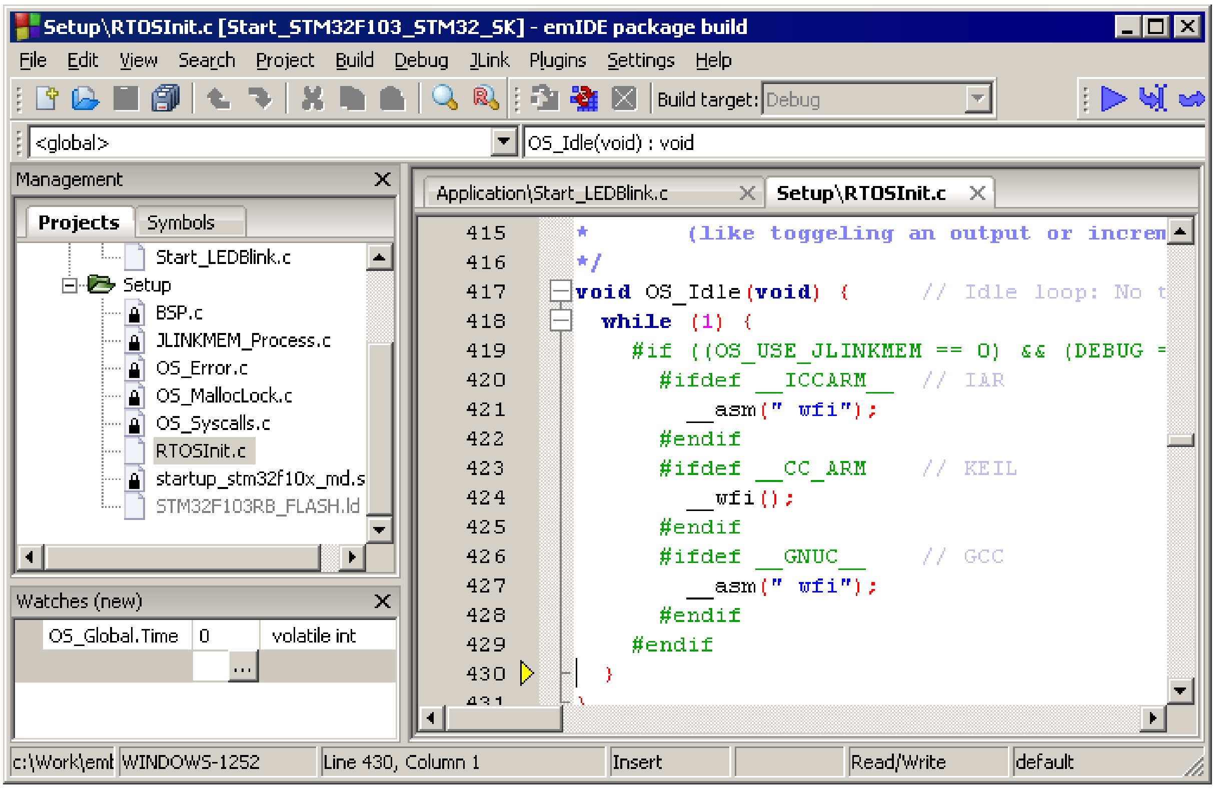
```
static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_Delay (50);
  }
}

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
```

OS_Global.Time	0	volatile int

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).

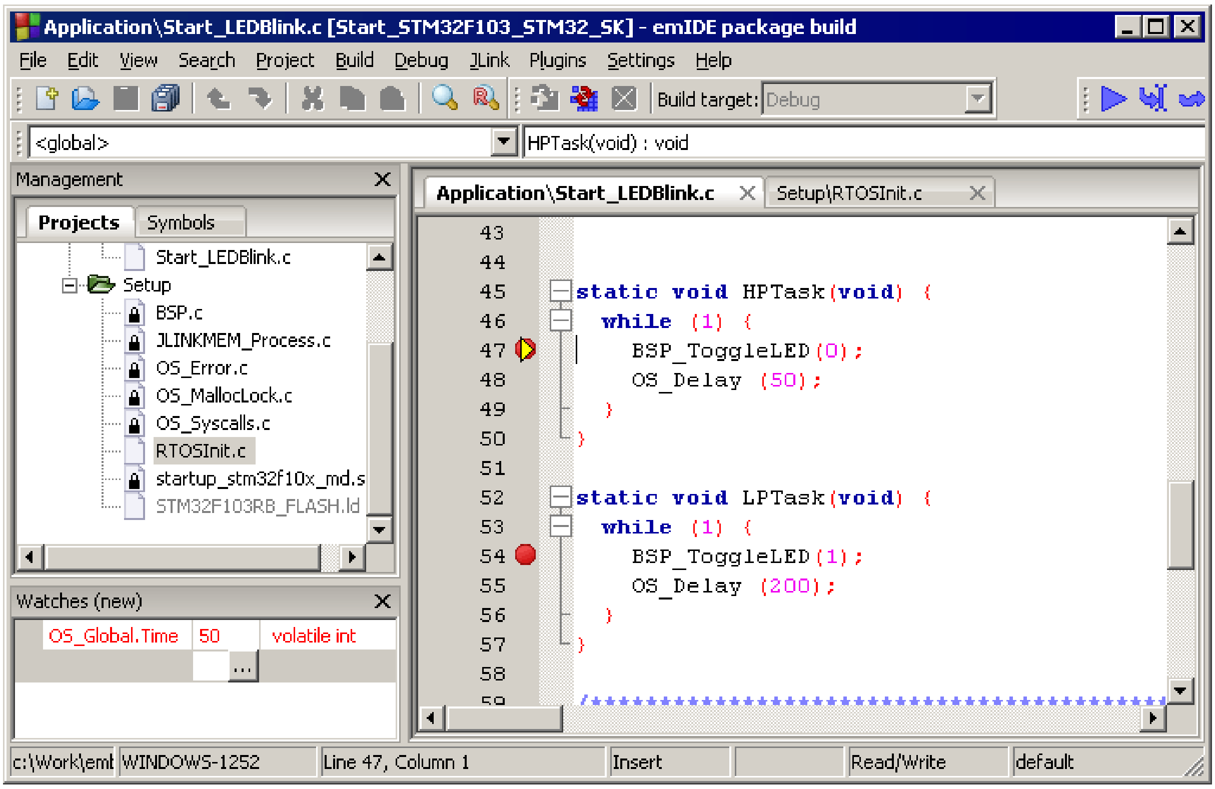
You will arrive there when you step into the `OS_TASK_Delay()` function in disassembly mode. `OS_Idle()` is part of `RTOSInit.c`. You may also set a breakpoint there before stepping over the delay in `LPTask()`.


Setup\RTOSInit.c [Start_STM32F103_STM32_SK] - emIDE package build
File Edit View Search Project Build Debug JLink Plugins Settings Help
Build target: Debug
<global>
OS_Idle(void) : void
Management
Projects Symbols
Start_LEDBlink.c
Setup
BSP.c
JLINKMEM_Process.c
OS_Error.c
OS_MallocLock.c
OS_Syscalls.c
RTOSInit.c
startup_stm32f10x_md.s
STM32F103RB_FLASH.ld
Watches (new)
OS_Global.Time 0 volatile int
Application\Start_LEDBlink.c
Setup\RTOSInit.c
c:\Work\emI WINDOWS-1252 Line 430, Column 1 Insert Read/Write default

```
*        (like toggeling an output or incren
*/
void OS_Idle(void) {      // Idle loop: No t
  while (1) {
    #if ((OS_USE_JLINKMEM == 0) && (DEBUG =
      #ifdef __ICCARM__   // IAR
        __asm(" wfi");
      #endif
      #ifdef __CC_ARM     // KEIL
        __wfi();
      #endif
      #ifdef __GNUC__     // GCC
        __asm(" wfi");
      #endif
    #endif
  }
```

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.


Application\Start_LEDBlink.c [Start_STM32F103_STM32_SK] - emIDE package build
File Edit View Search Project Build Debug JLink Plugins Settings Help
Build target: Debug
<global>
HPTask(void) : void
Management
Projects Symbols
Start_LEDBlink.c
Setup
BSP.c
JLINKMEM_Process.c
OS_Error.c
OS_MallocLock.c
OS_Syscalls.c
RTOSInit.c
startup_stm32f10x_md.s
STM32F103RB_FLASH.ld
Watches (new)
Application\Start_LEDBlink.c
Setup\RTOSInit.c
c:\Work\emI WINDOWS-1252 Line 47, Column 1 Insert Read/Write default

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

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
  }
}
/*****************************************************
```

OS_Global.Time	50	volatile int

Chapter 2

Build your own application

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.

Chapter 3

Libraries

3.1 Naming conventions for prebuilt libraries

embOS is shipped with different pre-built libraries with different combinations of features.

The libraries are named as follows:

`libosT<Architecture><VFP_support><Endianness><LibMode><Errata><TrustZone>.a`

Parameter	Meaning	Values
Architecture	Specifies the ARM architecture	6 : Cortex-M0/M0+/M1 7 : Cortex-M3/M4/M7 8BL : Cortex-M23 8ML : Cortex-M33
VFP_support	Floating point support	: No hardware VFP support V : VFP with softfp floating-point ABI VH : VFP with hard floating-point ABI
Endianness	Byte order	B : Big endian L : Little endian
LibMode	Specifies the 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	Specifies whether a workaround for ARM errata was applied.	_837070 : Erratum 837070 applied. : No workaround applied.
TrustZone	Specifies whether ARM TrustZone support is included.	_TZ: ARM TrustZone support applied. : No ARM TrustZone support.

Example

`libosT7LDP.a` is the library for a project using a Cortex-M3 or Cortex-M4 core without VFP, thumb mode, little endian mode with debug and profiling support.

`libosT7VLDP.a` is the library for a project using a Cortex-M4F core, thumb mode, little endian mode and VFPv4 softfp floating point unit with debug and profiling support.

> **Note**
>
> With earlier versions of embOS for Cortex-M the workaround for ARM erratum 837070 was applied by default for ARMv7-M devices. If libraries including the workaround are desired, a suitable set of libraries is provided, but projects would need to be updated accordingly. Please have a look in the chapter ARM erratum 837070 for more details.

Chapter 4

CPU and compiler specifics

4.1 Standard system libraries

embOS for Cortex-M and GCC may be used with the Red Hat newlib C libraries for most of all projects without any modification.

Since heap management with newlib depends on a working implementation of an `__sbrk()` function, that implementation is provided with embOS inside the source module `OS_Syscalls.c`, which itself is included in the "Setup" subdirectory of every embOS start project. Using that source file requires the symbols `__heap_start__` and `__heap_end__` to be appropriately defined in the respective project's linker file.

Heap management and file operation functions of standard system libraries are not reentrant and require a special initialization or additional modules when used with embOS when those non-thread-safe functions are used from different tasks (refer to *Interrupt and thread safety* on page 22).

4.2 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. Newlib 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.

4.2.1 __malloc_lock(), lock the heap against mutual access

`__malloc_lock()` is the locking function which is called by the system library whenever the heap management has to be locked against mutual access. The implementation delivered with embOS claims a mutex or disables interrupts to achieve this.

4.2.2 __malloc_unlock()

`__malloc_unlock()` is the counterpart to `__malloc_lock()`. It is called by the system library whenever the heap management locking can be released. The implementation delivered with embOS releases the mutex or restores the interrupt state.

None of these functions has to be called directly by the application. They are called from the system library functions when required. The functions are delivered in source form to allow replacement of the dummy functions in the system library.

4.3 Thread-Local Storage TLS

Newlib 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 GCC is prepared to support thread-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 22.

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.

4.3.1 OS_TLS_Set()

Description

OS_TLS_Set() is used by a task to initialize Thread-local storage for the current task.

Prototype

```
void OS_TLS_Set(struct _reent* pReentStruct);
```

Parameters

Parameter	Description
pReentStruct	Pointer to the thread local storage. It is the address of the variable of type struct _reent which holds the thread local data.

Additional information

OS_TLS_Set() shall be the first function called from a task when TLS should be used in the specific task. This function has to be only used 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.

Please ensure sufficient task stack if the _reent structure variable is placed on the task stack. For details on the _reent structure, _impure_ptr, and library functions which require precautions on reentrancy, refer to the GNU documentation.

Example

```
static void Task(void) {
  struct _reent TaskReentStruct;

  OS_TLS_Set(&TaskReentStruct);
  OS_TASK_SetContextExtension(&OS_TLS_ContextExtension);
  while (1) {
  }
}
```

4.3.2 OS_TLS_SetTaskContextExtension()

Description

OS_TLS_SetTaskContextExtension() may be called from a task to initialize thread-local storage for the current task and set the respective task context extension.

Prototype

```
void OS_TLS_SetTaskContextExtension(struct _reent* pReentStruct);
```

Parameters

Parameter	Description
pReentStruct	Pointer to the thread local storage. It is the address of the variable of type struct _reent which holds the thread local data.

Additional information

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

Please ensure sufficient task stack if the _reent structure variable is placed on the task stack. For details on the _reent structure, _impure_ptr, and library functions which require precautions on reentrancy, refer to the GNU documentation.

Example

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

```
#include "RTOS.h"

static OS_STACKPTR int StackHP[512], StackLP[512];  // Task stacks
static OS_TASK         TCBHP, TCBLP;                // Task control blocks

static void HPTask(void) {
  struct _reent TaskReentStruct;

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

static void LPTask(void) {
  struct _reent TaskReentStruct;

  OS_TLS_SetTaskContextExtension(&TaskReentStruct);
  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;
}
```

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 20). 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 unconditionally clears `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 to restore any previous value of `PRIMASK` after modification of `BASEPRI`. To do so, the define `OS_PRESERVE_PRIMASK` shall be set to 1 in the preprocessor settings or inside `OS_Config.h`.

4.5 ARMv8-M 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 OS_PSPLIM_Set()

Description

Sets the PSPLIM register.

Prototype

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

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.

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);
  }
}
```

4.5.2 OS_PSPLIM_SetTaskContextExtension()

Description

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

Prototype

```
void OS_PSPLIM_SetTaskContextExtension(const void* 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 31.

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);
  }
}
```

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 OS_ARM_TZ_SetSecureStatePSP()

Description

OS_ARM_TZ_SetSecureStatePSP() sets the secure PSP_s, PSPLIM_s and CONTROL_s register

Prototype

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

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 register and the secure control register. 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 before with the task context OS_ARM_TZ_ContextExtension (e.g. with 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);
  }
}
```

4.6.2 OS_ARM_TZ_SetTaskContextExtension()

Description

OS_ARM_TZ_SetTaskContextExtension() sets the secure PSP_s, PSPLIM_s and CONTROL_s register and extends the task context to save and restore these register.

Prototype

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

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 register and the secure control register. 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);
  }
}
```

Chapter 5

Stacks

5.1 Task stack for Cortex-M

Each task uses its 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 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 embOS system executes in thread mode, the scheduler executes in handler mode. The minimum system stack size required by embOS is about 160 bytes (stack check & profiling build). However, since the system stack is also used by the application before the start of multitasking (the call to `OS_Start()`), and because software timers and C-level interrupt handlers also use the system stack, the actual stack requirements depend on the application.

The size of the system stack can be changed by modifying the project settings or linker file. We recommend a minimum stack size of 256 bytes for the system stack.

In order to perform overflow checks on the system check and to provide stack usage information, embOS requires the symbols `__stack_start__` and `__stack_end__` to be appropriately defined in the respective project's linker file.

5.3 Interrupt stack for Cortex-M

If a normal hardware exception occurs, the Cortex-M core switches to handler mode which uses the main stack pointer. With embOS, the main stack pointer is initialized to use the `CSTACK` which is defined in the linker command file. The main stack is also used as stack by the embOS scheduler and during idle times, when no task is ready to run and `OS_Idle()` is executed.

> **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 `CSTACK`.

Chapter 6

Interrupts

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 in ROM at address `0x00`. It contains the initial stack pointer as well as the addresses of all exception handlers, which are defined in a C source or assembly file in the CPU specific subdirectory. All interrupt handler function addresses have to be present in that file at compile time as long as the table is kept in ROM.

If the vector table is copied to RAM, however, interrupt handlers can be installed dynamically at runtime. To do so, the vector table base register inside the NVIC controller has to be initialized to point to the vector table base address in RAM.

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()`.

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 M23 supports 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.

- 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 defined in `OS_Config.h` or 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.

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.

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 per default. Any interrupt or exception with a higher preemption level may interrupt an interrupt handler running on a lower preemption level. An interrupt handler calling embOS functions has to start with an embOS prologue function; it 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.

6.7.1 OS_INT_Enter()

Description

Disables nesting.

Prototype

```
void OS_INT_Enter (void);
```

Additional information

`OS_INT_Enter()` has to be used as prologue function, when the interrupt handler should not be preempted by any other interrupt handler that runs on a priority below the zero latency interrupt priority. An interrupt handler that starts with `OS_INT_Enter()` has to end with the epilogue function `OS_INT_Leave()`.

Example

Interrupt-routine that can not be preempted by other interrupts.

```
static void _Systick(void) {
  OS_INT_Enter();   // Inform embOS that interrupt code is running
  OS_HandleTick();  // Can not be interrupted by higher priority interrupts
  OS_INT_Leave();   // Inform embOS that interrupt handler is left
}
```

6.7.2 OS_INT_EnterNestable()

Description

Enables nesting.

Prototype

```
void OS_INT_EnterNestable (void);
```

Additional information

`OS_INT_EnterNestable()`, allow nesting. `OS_INT_EnterNestable()` may be used as prologue function, when the interrupt handler may be preempted by any other interrupt handler that runs on a higher interrupt priority. An interrupt handler that starts with `OS_INT_EnterNestable()` has to end with the epilogue function `OS_INT_LeaveNestable()`.

Example

Interrupt routine that can be preempted by higher priority interrupts.

```
static void _Systick(void) {
  OS_INT_EnterNestable();  // Inform embOS that interrupt code is running
  OS_HandleTick();         // Can be interrupted by higher priority interrupts
  OS_INT_LeaveNestable();  // Inform embOS that interrupt handler is left
}
```

6.8 Interrupt handling API

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`.

To handle interrupts with the vectored interrupt controller, embOS offers the following functions:

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
`OS_ARM_ISRInit()`	Used to initialize the interrupt handling.	●	●			
`OS_ARM_InstallISRHandler()`	Installs an interrupt handler.	●	●			
`OS_ARM_EnableISR()`	Enables a specific interrupt source.	●	●		●	●
`OS_ARM_DisableISR()`	Disables a specific interrupt source.	●	●		●	●
`OS_ARM_ISRSetPrio()`	Set or modify the priority of a specific interrupt source.	●	●			

6.8.1 OS_ARM_ISRInit()

Description

Used to initialize 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 whether a RAM vector table is used. 0: Vector table in Flash. 1: Vector table in RAM.
NumInterrupts	Number of implemented interrupts.
VectorTableBaseAddr	Flash vector table address.
RAMVectorTableBaseAddr	RAM vector table 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);
}
```

6.8.2 OS_ARM_InstallISRHandler()

Description

Installs an interrupt handler.

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

The previous interrupt handler.

Additional information

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

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);
}
```

6.8.3 OS_ARM_EnableISR()

Description

Used to enable interrupt acceptance of a specific interrupt source in a vectored interrupt controller.

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.

6.8.4 OS_ARM_DisableISR()

Description

Used to disable interrupt acceptance of a specific interrupt source in a vectored interrupt controller which is not of the VIC type.

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.

6.8.5 OS_ARM_ISRSetPrio()

Description

Used to set or modify the priority of a specific interrupt source by programming the interrupt controller.

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 specific interrupt. Prio ranges from 0 (highest priority) to 255 (lowest priority).

Additional information

This function sets the priority of an interrupt channel by programming the interrupt controller. Please refer to CPU-specific manuals about allowed priority levels. 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. 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**
>
> Please note there are Arm core specific restrictions when you must not change the exception priority. For example, you must not change the priority of an active exception. For more information, please have a look in the according Arm Architecture Reference Manual.

Chapter 7

CMSIS

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.

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 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 initialize 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.

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 52. 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 38 and *Interrupt nesting* on page 41, about the differences between interrupt priority and ID values used to setup the NVIC controller, please refer to *Different interrupt priority values* on page 52.

Chapter 8

Floating Point (FP) support

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 libraries 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.

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`.

- `soft` causes GCC to generate output containing library calls for floating-point operations.
- `softfp` allows the generation of code using hardware floating-point instructions, but still uses the soft-float calling conventions.
- `hard` allows generation of floating-point instructions and uses FPU-specific calling conventions.

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

Chapter 9

MPU support

9.1 Introduction

This section describes the optional Memory Protection Unit (MPU). An MPU divides the memory map into a number of regions and defines the location, size, access permissions, and memory attributes for each region. The access permissions affect the behavior of memory accesses to the region. If the CPU accesses a memory location that is prohibited by the MPU, the processor generates a fault.

While ARMv6-M and ARMv8-M Baseline processors can only generate HardFault exceptions, ARMv7[E]-M and ARMv8-M Mainline processors can generate dedicated MemManage exceptions for memory accesses which are prohibited by the MPU. ARMv7[E]-M and ARMv8-M Mainline processors can also cause other faults like BusFaults or UsageFaults. All faults are handled by embOS-MPU.

Not all Cortex-M CPUs implement the same MPU. ARM provides an individual MPU for each architecture, i.e. ARMv6-M, ARMv7[E]-M and ARMv8-M. However, the ARM MPUs are optional and device manufacturers may implement a different MPU instead. MPU specific rules and requirements that need to be met are explained in the respective subchapter of *MPU types* on page 60.

Note

The MPU hardware is managed by embOS-MPU. You must not enable or disable the MPU in your application.

9.2 Supervisor call

embOS-MPU needs a safe way to switch from an unprivileged task to the privileged OS. With Cortex-M this is done via the SVCall. An SVC exception handler is necessary to handle the SVCall. The file `SVCHandler.S` contains the SVC exception handler. When working with your own project please ensure that this file is part of your project.

9.3 Fault exceptions

If a task does an invalid operation an exception occurs. With Cortex-M this can be a HardFault on ARMv6/8-M Baseline processors, or a HardFault, MemManage fault, BusFault or a UsageFault on ARMv7[E]/8-M Mainline processors. These faults are handled in `HardFaultHandler.S`, `MemManageHandler.S`, `BusFaultHandler.S` and `UsageFaultHandler.S`. When working with your own project please ensure that these files are part of your project and the fault handlers are inserted in your vector table according to the used architecture. embOS-MPU for Cortex-M enables all fault exceptions on ARMv7[E]/8-M Mainline processors.

9.4 Alignment

Cortex-M MPUs require special memory alignment. embOS-MPU checks the alignment and calls `OS_Error()` in case of a wrong alignment. embOS setups MPU regions for newly created tasks. One MPU region is setup for the task stack. Thus, the task stack must also meet the requirements of MPU regions.

9.5 MPU memory attributes

With `OS_MPU_AddRegion()` it is possible to use the `Attributes` parameter to set additional attributes for a specific memory region. These attributes are core and MPU specific.

Prototype

```
void OS_MPU_AddRegion(OS_TASK* pTask,
                      OS_U32   BaseAddr,
                      OS_U32   Size,
                      OS_U32   Permissions,
                      OS_U32   Attributes);
```

9.6 Changing memory attributes for privileged tasks

While `OS_MPU_AddRegion()` can be used to change e.g. cache settings for unprivileged tasks, no embOS API exists to set cache settings for privileged tasks. Privileged tasks may program the MPU directly and must then use `OS_MPU_ExtendTaskContext()` to save and restore the MPU register contents during context switches. Please note the MPU is not enabled before this task was scheduled once since the MPU is disabled/enabled by the extended task context and must not be enabled by the user application.

Example:

```
void PrivTask(void) {
  OS_MPU_ExtendTaskContext();
  SetMPURegister();  // Program MPU/cache settings
  OS_TASK_Delay(1);  // Force a re-schedule which enables the MPU
  while (1) {
    OS_TASK_Delay(100);
  }
}
```

9.7 OS_MPU_ExtendTaskContext()

`OS_MPU_ExtendTaskContext()` extends the task context for the MPU registers which means it saves and restores the MPU settings.
When the task gets deactivated the MPU task context extension performs the following actions:

- Disable and clean/invalidate data and instruction cache (only when cache is available)
- Disable the MPU
- Save all MPU register values to the task stack
- Set MPU register values to zero, thereby disabling all MPU regions (only in embOS debug build)
- Enable data and instruction cache (only when cache is available)

When the task gets activated again the MPU task context extension performs the following actions:

- Disable and clean/invalidate data and instruction cache (only when cache is available)
- Restore the MPU register values from the task stack
- Enable the MPU
- Enable data and instruction cache (only when cache is available)

9.8 Cache maintenance

When a task context on a device with cache is extended with `OS_MPU_ExtendTaskContext()`, embOS saves and restores the MPU and cache settings for that task with every context switch. Since the RTOS itself or another task may run with different cache settings, the instruction and data caches need to be cleaned and invalidated. Please be aware that cache maintenance operations take some time, which also increases the context switch time. The actual context switch time depends on many factors, e.g. the number of cache lines that must be written back to the memory. Since embOS manages the cache settings during task switches, please note that caches may not be enabled or disabled during the application’s execution.

9.9 Buffer for MPU sanity check

Each task for which an MPU sanity check should be performed needs a buffer which holds a copy of the MPU register. The size of the buffer depends on the actual MPU hardware. The MPU sanity check is only available in `OS_LIBMODE_SAFE`. The following defines can be used for defining the buffer:

Core	Define	Value
ARMv6/7[E]-M	`OS_ARM_V7M_MPU_REGS_SIZE`	128
ARMv8-M	`OS_ARM_V8M_MPU_REGS_SIZE`	128
NXP S32K11x	`OS_S32K11X_MPU_REGS_SIZE`	128
NXP Kinetis K66	`OS_KINETIS_MPU_REGS_SIZE`	256

Example

```
OS_U8 Task1_Buffer[OS_ARM_V7M_MPU_REGS_SIZE];
OS_MPU_SetSanityCheckBuffer(&TCB_TASK1, Task1_Buffer);
```

9.10 MPU types

embOS-MPU supports different MPU implementations with and without cache. These predefined structures can be used with `OS_MPU_Init()`.

Core	Define
ARMv6/7[E]-M (Cortex-M0+/M3/M4)	OS_ARMv7M_MPU_API
ARMv7E-M with Cache (Cortex-M7)	OS_ARMv7M_CACHE_MPU_API
ARMv8-M (Cortex-M23/M33)	OS_ARMv8M_MPU_API
NXP S32K11x	OS_S32K11X_MPU_API
NXP Kinetis K66	OS_KINETIS_MPU_API

Example:

```
int main(void) {
  //
  // Initializes the MPU for Cortex-M0+/M3/M4
  //
  OS_MPU_Init(OS_ARMv7M_MPU_API);
}
```

```
int main(void) {
  //
  // Initializes the MPU for Cortex-M7 with cache
  //
  OS_MPU_Init(&OS_ARMv7M_CACHE_MPU_API);
}
```

9.10.1 ARMv6-M & ARMv7[E]-M MPUs

9.10.1.1 Alignment

The size must be a power of two and at least 256 bytes for ARMv6-M MPUs and 32 bytes for ARMv7[E]-M MPUs. The base address must be divisible by the size with zero remainder.

9.10.1.2 MPU memory attributes

The `Attributes` parameter of `OS_MPU_AddRegion()` specifies the memory type, shareability and cacheability of the memory region. The value of `Attributes` is written into `MPU_RASR[21:16]`. For more information, please refer to the *Armv6-M Architecture Reference Manual* or *Armv7[E]-M Architecture Reference Manual*.

The following defines can be used with the `Attributes` parameter of `OS_MPU_AddRegion()`:

Define	Explanation
OS_ARM_CACHEMODE_STRONGLY_ORDERED	Strongly ordered
OS_ARM_CACHEMODE_SHAREABLE_DEVICE	Shareable Device
OS_ARM_CACHEMODE_WRITE_THROUGH	Outer and Inner Write-Through, no Write-Allocate
OS_ARM_CACHEMODE_WRITE_BACK_NO_ALLOC	Outer and Inner Write-Back, no Write-Allocate
OS_ARM_CACHEMODE_NON_CACHEABLE	Outer and Inner Non-cacheable
OS_ARM_CACHEMODE_WRITE_BACK_ALLOC	Outer and Inner Write-Back, Write and Read-Allocate

Example:

```
void HPTask(void) {
  OS_MPU_AddRegion(&TCBHP, (OS_U32)MyQBuffer, 512,
                   OS_MPU_READWRITE, OS_ARM_CACHEMODE_WRITE_BACK_ALLOC);
  OS_MPU_SwitchToUnprivState();
  while (1) {
    DoSomething();
    OS_TASK_Delay(10);
  }
}
```

9.10.1.3 Modifying permissions and attributes of default task regions

The following tables show the predefined regions for unprivileged tasks as well as their permissions and attributes.

Region	Permissions	Attributes
ROM	OS_MPU_READONLY, OS_MPU_EXECUTION_ALLOWED	OS_ARM_CACHEMODE_STRONGLY_ORDERED
RAM	OS_MPU_READONLY, OS_MPU_EXECUTION_ALLOWED	OS_ARM_CACHEMODE_STRONGLY_ORDERED
Task stack	OS_MPU_READWRITE, OS_MPU_EXECUTION_ALLOWED	OS_ARM_CACHEMODE_STRONGLY_ORDERED

The default permissions and attributes can be changed by adding additional regions with different permissions and attributes. Memory regions with higher index have priority over regions which were defined earlier.

Execute permissions

With ARM MPUs it is possible to mark memory regions as non-executable. Unfortunately, this setting applies to both privileged and unprivileged states. Therefore, the XN bit also affects code which gets executed in an interrupt service routine that preempts an unprivileged task, or code which is executed when an unprivileged task calls a device driver.

9.10.1.4 ARMv7[E]-M cache

The data and instruction caches are disabled after reset. embOS-MPU invalidates and enables the instruction and data caches in `OS_MPU_Init()` when they are not yet enabled. embOS-MPU expects the caches do not need to be cleaned. You must not invalidate and enable the caches after `OS_MPU_Init()` again without cleaning the caches.

9.10.2 ARMv8-M MPUs

9.10.2.1 Alignment

The start and end address of regions must be 32-byte aligned.

9.10.2.2 MPU memory attributes

The `Attribues` parameter of `OS_MPU_AddRegion()` sets the attribute index field (bits [3:1]) of the `MPU_RLAR` register. The attribute index field associates the region with one of the 8 memory attribute encodings in the registers `MPU_MAIR0` and `MPU_MAIR1`. The attribute encoding for a specific index can be set by using the embOS API function `OS_ARMv8M_SetMPUAttribute()`. For more information about the attribute index and applicable attribute encodings, please refer to the *Armv8-M Architecture Reference Manual*.

9.10.2.3 Modifying permissions and attributes of default task regions

The following tables show the predefined regions for unprivileged tasks as well as their permissions and attributes.

Region	Permissions	Attributes
ROM	Read-only for privileged and unprivileged, `OS_MPU_EXECUTION_ALLOWED`	Attribute Index 0
RAM	No default region for RAM set. Behaves as `OS_MPU_NOACCESS`.	-

Task stack	`OS_MPU_READWRITE,`
	`OS_MPU_EXECUTION_ALLOWED`
		Attribute Index 0

ARMv8-M comes with some restrictions regarding the permissions that can be applied to regions, which is why default regions need to be handled differently. The following table shows the permissions that the architecture can assign to regions:

AP Encoding	Privileged Permissions	Unprivileged Permissions
00	Read/write	No access
01	Read/write	Read/write
10	Read-only	No access
11	Read-only	Read-only

With embOS, permissions that are applied to regions via `OS_MPU_AddRegion()` always require the embOS, which is executed in privileged state, to have full access. With that restriction only two of these four permissions are left to be assigned to regions. The following table shows the mapping of embOS MPU permissions to the actual architecture memory permissions assigned to regions:

embOS MPU permission	AP Encoding	Privileged Permissions	Unprivileged Permissions
`OS_MPU_NOACCESS`	00	Read/write	No access
`OS_MPU_READONLY`	00	Read/write	No access
`OS_MPU_READWRITE`	01	Read/write	Read/write

To make unprivileged tasks able to read ROM, a special memory permission is assigned to the ROM region which gives both, privileged and unprivileged code, the read-only permission. Now, privileged code cannot write to ROM, but this usually isn’t required anyway.

A region for RAM is not set. This is because setting `OS_MPU_READONLY` behaves the same as the permissions of the default memory map, which apply when no region is set for the accessed memory location. Also, since regions may not overlap with ARMv8-M, adding no default region for RAM allows the user to add custom regions located in RAM.

For modifying the attributes of the attribute index 0, please refer to *OS_ARMv8M_SetMPUAttribute* on page 64.

Execute permissions

With ARM MPUs it is possible to mark memory regions as non-executable. Unfortunately, this setting applies to both privileged and unprivileged states. Therefore, the XN bit also affects code which gets executed in an interrupt service routine that preempts an unprivileged task, or code which is executed when an unprivileged task calls a device driver.

9.10.2.4 Adding additional regions

When adding additional regions, it needs to be ensured that the regions of a task don’t overlap. This can be achieved by packing the data, which the task needs to access, into a dedicated structure or section that meets the size and alignment requirements. Structures and sections should be padded with unused bytes to obtain a size divisible by 32 and to avoid other data, to which the task shouldn’t have access to, to be placed in the remaining bytes of the region.

9.10.2.5 Task stacks for non-secure unprivileged tasks

If TrustZone is used and an unprivileged task calls secure functions from the non-secure world it needs an additional secure stack. The stack should be placed into secure memory, so that the non-secure world cannot access the secure stack.

```
// Ensure StackLP_s is placed in secure memory and properly aligned
static OS_STACKPTR int StackLP_s[256];

OS_ARM_TZ_SetTaskContextExtension(&Arm_TZ_ApiList, StackLP_s, sizeof(StackLP_s));
```

9.10.2.6 embOS-MPU variables

Unprivileged tasks need to have access to embOS MPU specific data located in a struct which is linked into a section called `.mpu_ram`. Since embOS-MPU for ARMv8-M does not support read-only access on RAM due to ARMv8-M restrictions, an additional MPU region needs to be added for the `.mpu_ram` section. The linker file needs to be also modified, so that `.mpu_ram` is linked into the memory with the required 32-byte alignment and without overlapping any other MPU region. The size of the structure contained in this section is 32 bytes to meet the size requirement for that region.

Example

```
extern char __MPU_RAM_segment_start__[];
extern char __MPU_RAM_segment_size__[];

OS_MPU_AddRegion(&TCBLP, __MPU_RAM_segment_start__,
                 (OS_U32)__MPU_RAM_segment_size__, OS_MPU_READWRITE, 0);
```

9.10.2.7 OS_ARMv8M_SetMPUAttribute()

Description

Programs the specified attribute within the Memory Attribute Indirection register.

Prototype

```
void OS_ARMv8M_SetMPUAttribute(OS_U32 Index,
                               OS_U32 Attribute);
```

Parameters

Parameter	Description
Index	Selects one of the 8 attributes that shall be configured.
Attribute	The actual attribute that shall be configured.

Additional information

Armv8-M supports two MPU Memory Attribute Indirection Register MPU_MAIR0 and MPU_MAIR1. Each register holds four sets of memory attribute settings which can used by MPU regions. Thus, a maximum of 8 sets of attributes can be configured. The actual set of attributes to be used by an MPU region is selected via the Attributes argument of OS_MPU_AddRegion() which specifies the index of the set.

Please refer to the ARMv8-M technical reference manual for valid Attribute values (MAIR_ATTR).

Example

```
extern char __RAM_segment_start__[];
extern char __RAM_segment_size__[];

//
// Set memory settings for attribute index 0
//
OS_ARMv8M_SetMPUAttribute(0, 0);
//
// Use attribute index 0
//
OS_MPU_AddRegion(&TCBLP, __RAM_segment_start__,
                 (OS_U32)__RAM_segment_size__, OS_MPU_READWRITE, 0);
```

9.10.3 NXP MPUs

9.10.3.1 Alignment

The start and end address of regions must be 32-byte aligned.

9.10.3.2 MPU memory attributes

With NXP MPUs the `Attributes` parameter of `OS_MPU_AddRegion()` is not used.

9.10.3.3 Modifying permissions and attributes of default task regions

The following tables show the predefined regions for unprivileged tasks as well as their permissions and attributes.

Region	Permissions	Attributes
ROM	OS_MPU_READONLY, OS_MPU_EXECUTION_ALLOWED	n/a
RAM	OS_MPU_READONLY, OS_MPU_EXECUTION_ALLOWED	n/a
Task stack	OS_MPU_READWRITE, OS_MPU_EXECUTION_ALLOWED	n/a

The default permissions can be changed by adding additional regions with different permissions. For overlapping memory regions, the least restrictive permissions are used for the memory access. This means that permissions granted for the default regions cannot be denied anymore using another region. For more information on permission priority, please refer to *NXP MPU permission priority* on page 66.

Example:

```
void HPTask(void) {
  //
  // Switch cache settings for RAM region to write back
  //
  OS_MPU_AddRegion(&TCBHP, RAM_START_ADDR, RAM_SIZE, OS_MPU_READONLY,
                   OS_ARM_CACHEMODE_WRITE_BACK_ALLOC);
  OS_MPU_SwitchToUnprivState();
  while (1) {
    DoSomething();
    OS_TASK_Delay(10);
  }
}
```

9.10.3.4 NXP MPU permission priority

The NXP MPU gives priority to granting permission over denying access for overlapping region descriptors. This is different to the ARM Cortex-M MPU which uses the access permissions from the higher region number for overlapping region descriptors. For example embOS-MPU grants read and execute access for the entire RAM for an unprivileged task per default. It is possible to extend the access permission to write access with an additional overlapping MPU region. But it is not possible to limit the access permissions to read-only.

With the NXP MPU separate access permissions can be programmed for the available bus masters. embOS-MPU programs access permissions for bus master 0 (Core) only. All other bus masters, like the DMA, have full access to the entire memory.

9.11 Further information

Please refer to the MPU chapter in the generic embOS manual. It describes the general embOS-MPU usage.

Chapter 10

RTT and SystemView

10.1 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.

10.2 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.

Chapter 11

Technical data

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	~2100 bytes
embOS kernel	RAM	~160 bytes
Task control block	RAM	308 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