embOS

Real-Time Operating System

CPU & Compiler specifics for ARM using Embedded Studio

Document: UM01062
Software Version: 5.18.0.0
Revision: 0
Date: November 15, 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

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

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

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

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

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

Manual versions

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

Print date: November 15, 2022

Software	Revision	Date	By	Description
5.18.0.0	0	221115	MM	Chapter “RTT and SystemView” added.
5.16.1.1	0	220707	MM	Chapter “Libraries” and “VFP and NEON support” updated.
5.16.1.0	0	220113	MM	Chapter “CPU and compiler specifics” updated.
5.14.0.0	0	210630	MM	New software version.
5.10.2.0	0	200908	MM	Chapter “VFP and NEON support” updated.
5.8.2.2	0	200228	TS	Chapter “MMU/MPU and cache support” updated.
5.8.2.1	0	200207	MM	New software version.
5.8.2.0	0	200106	MC	New software version.
5.06	0	190614	TS	Chapter “MMU/MPU and cache support updated regarding Cortex-R.
5.02a	0	180808	MM	New software version.
5.00	0	180530	TS	New software version.
4.40	0	180124	MC/TS	Chapter ”Thread-Local Storage TLS“ added.
4.34	0	170424	MC/MM	Chapter ”embOS Thread Script“ updated.
4.14.1	0	151124	TS	embOS library names updated.
4.14	0	151030	TS	First 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

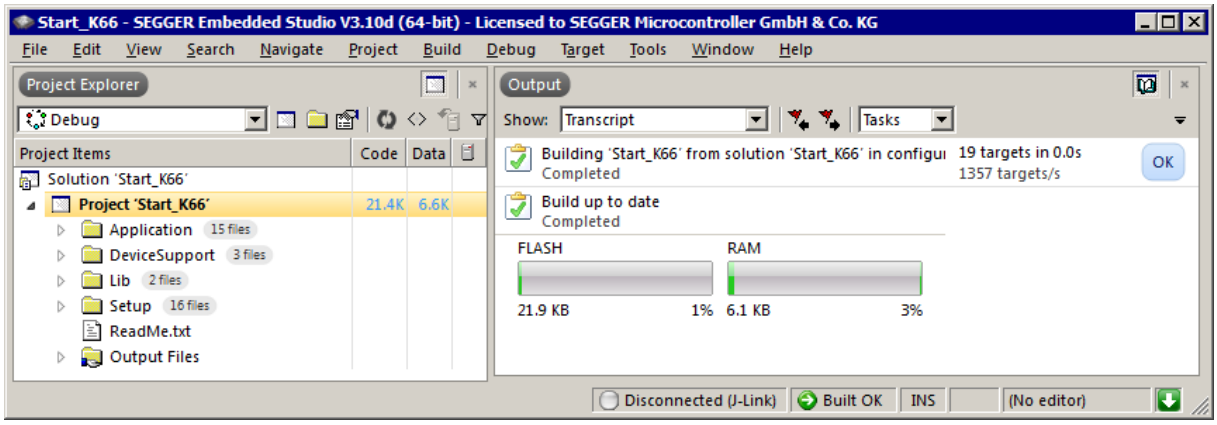
After installation of embOS you can create your first multitasking application. You have received several ready to go sample start workspaces and projects and every other files needed in the subfolder `Start`. It is a good idea to use one of them as a starting point for all of your applications. The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



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 ******************************/
```

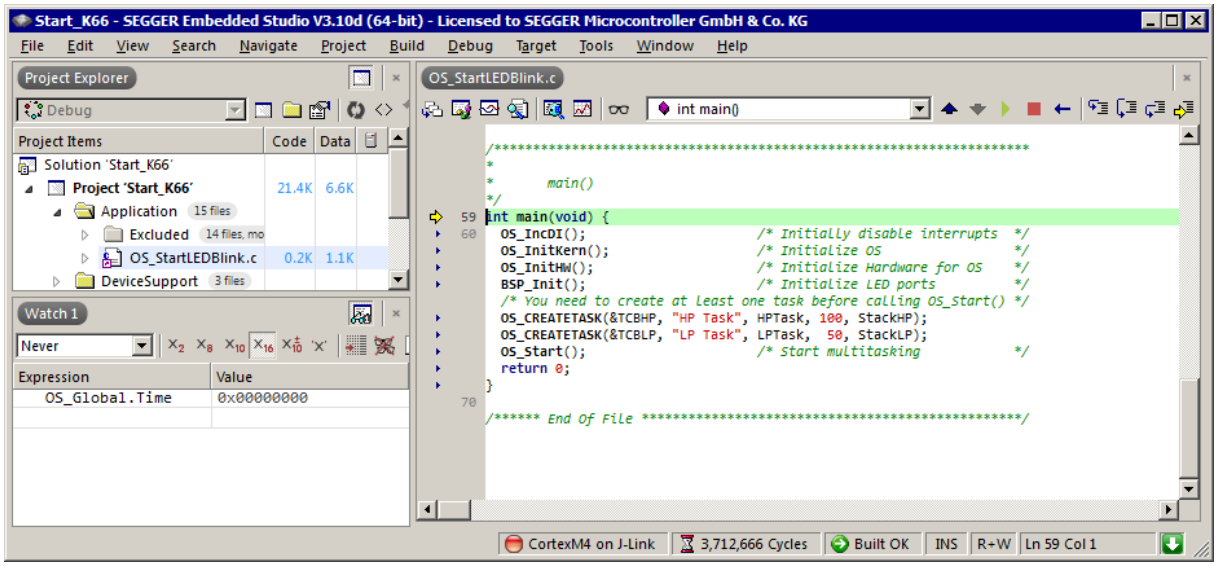

1.4 Stepping through the sample application

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.

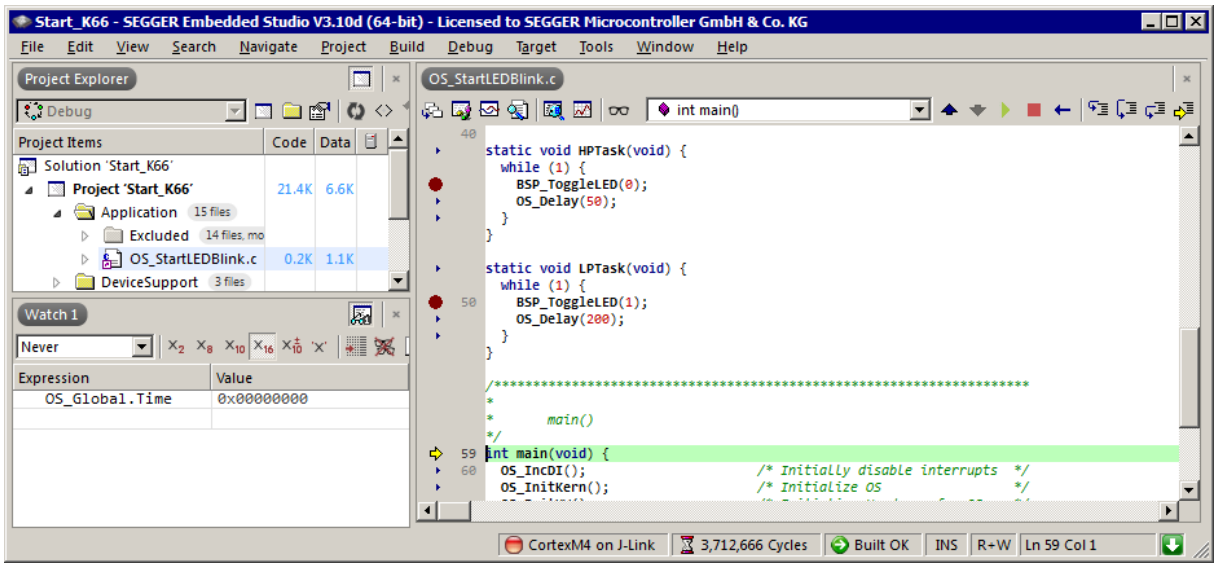
`OS_Init()` is part of the embOS library and written in assembler; you can therefore only step into it in disassembly mode. It initializes the relevant OS variables.

`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.

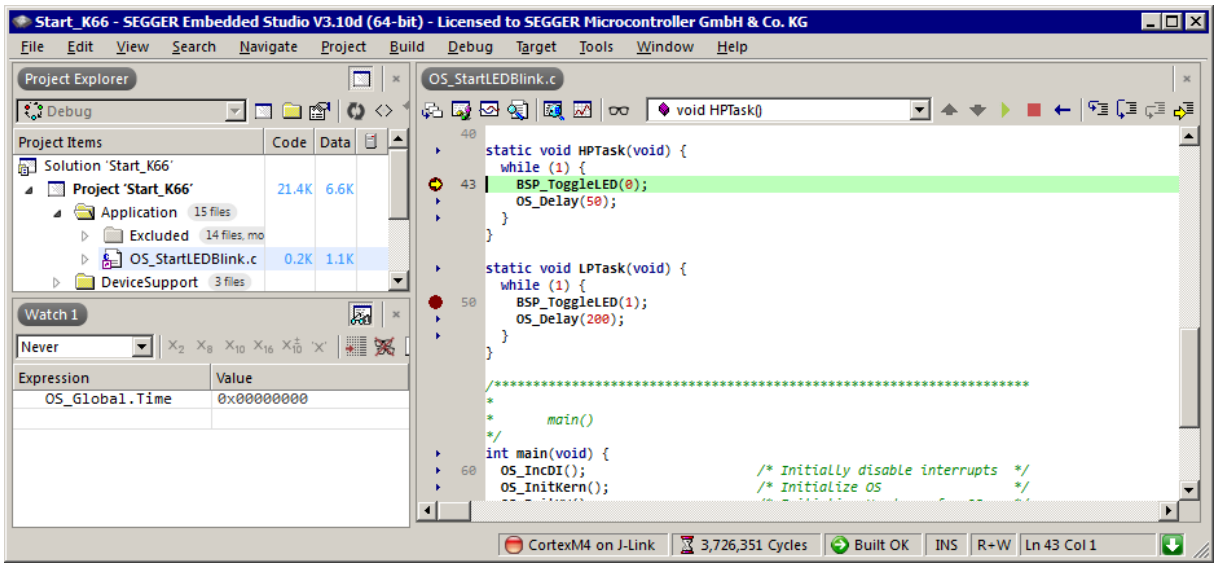


Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.

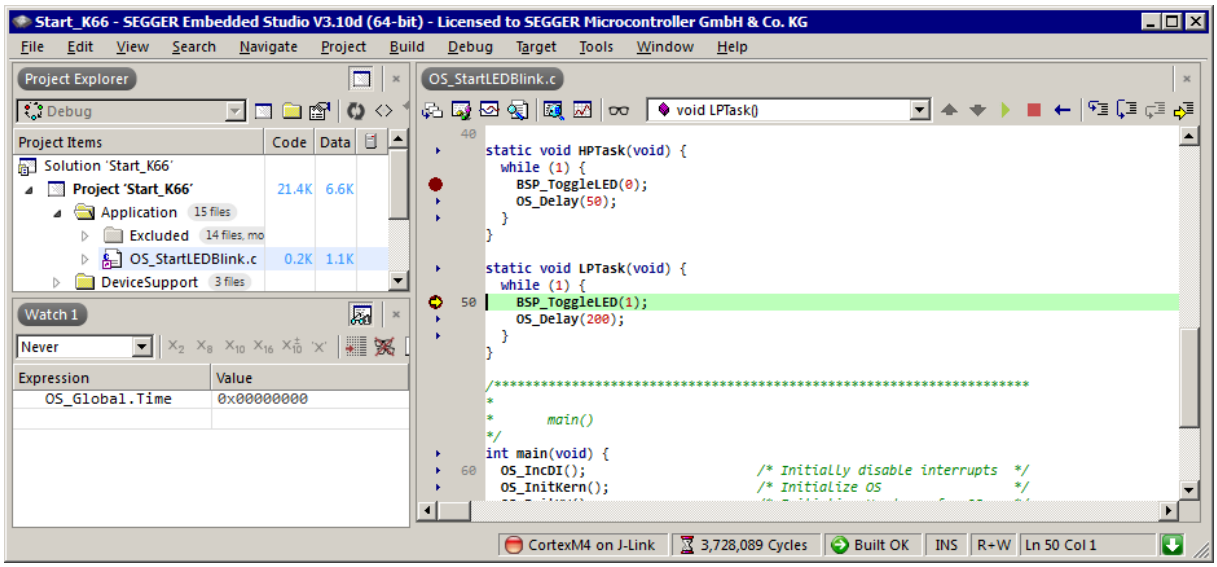


As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only.

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

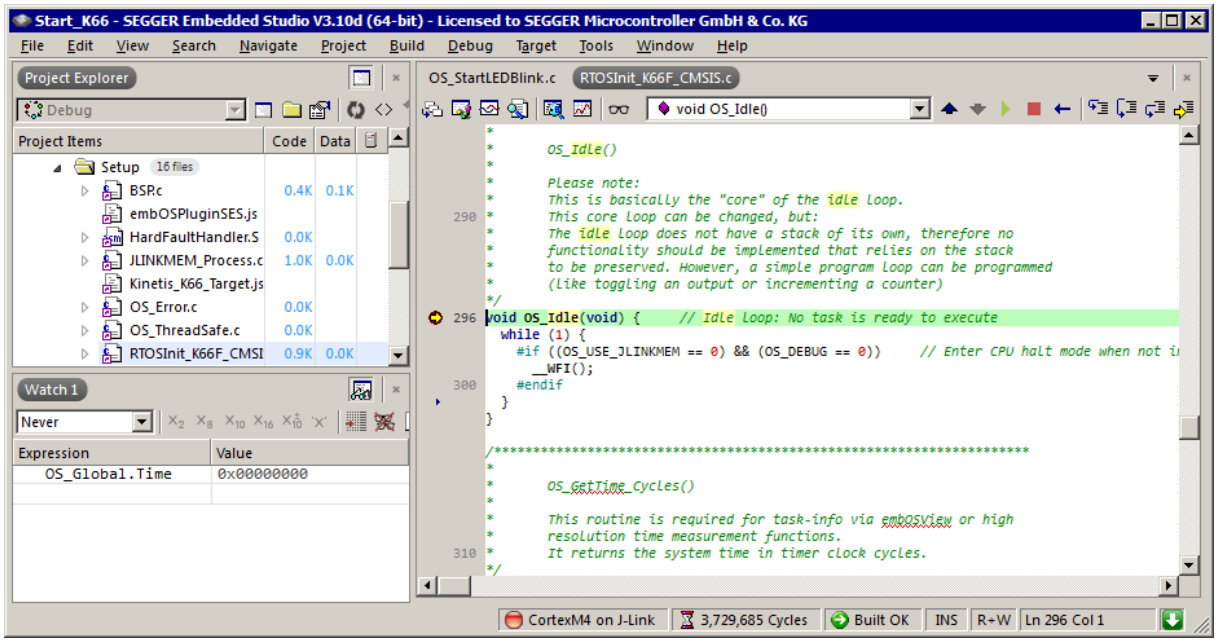


If you continue stepping, you will arrive at the task that has lower priority:



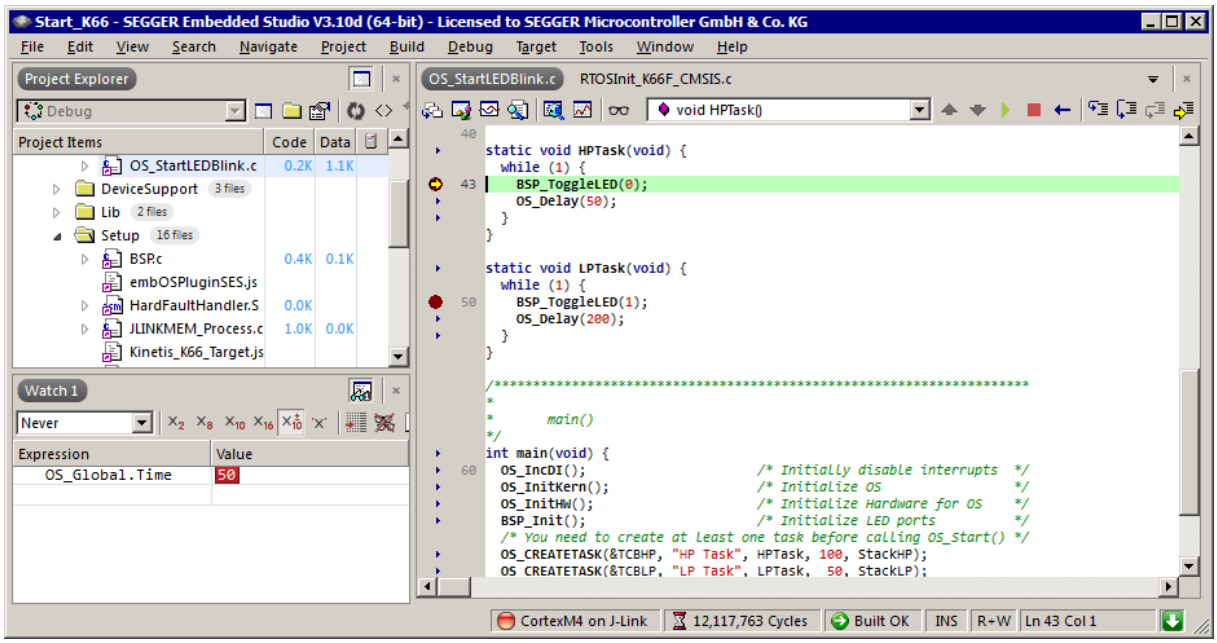
Continue to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

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



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

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



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:

`libos_<Architecture><VFP_NEON><CpuMode>_<Endianness>_<Interwork>_<Libmode>.a`

Parameter	Meaning	Values
`Architecture`	Specifies the ARM architecture	v4t : ARMv4 v5te: ARMv5 v7a : ARMv7-A v7r : ARMv7-R
`VFP_NEON`	Floating point / NEON support	_ : No VFP support _vfp_d16_ : VFP-D16 with softfp floating-point ABI _vfp_d16_hard_: VFP-D16 with hard floating-point ABI _vfp_d32_ : VFP-D32 with softfp floating-point ABI _vfp_d32_hard_: VFP-D32 with hard floating-point ABI
`CpuMode`	Specifies the CPU mode	a : ARM t : Thumb / Thumb2
`Endianness`	Byte order	be : Big endian le : Little endian
`Interwork`	Specifies if interworking is used	n : Non interwork (ARMv4, ARMv5) i : Interwork
`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

Example

`libos_v7a_vfp_d32_t_le_i_dp.a` is the library for an ARMv7A core, thumb2 mode, support for VFP/NEON D32, little endian mode, interwork, with debug and profiling support.

> **Note**
>
> When updating from an earlier embOS version you might need to update to an embOS library with VFP/NEON support. For example, if you use `libos_v7a_t_le_i_dp.a` for an ARMv7A CPU with VFP/NEON unit you will get a linker error message like "undefined symbol `OS_Init_VFPD32` referenced by symbol main".
> This check avoids that the project and the used embOS library use different VFP/NEON settings. If your project settings allow the compiler to generate VFP/NEON instructions, an embOS library with VFP/NEON support like `libos_v7a_vfp_d32_t_le_i_dp.a` must be used.

Chapter 4

CPU and compiler specifics

4.1 Interrupt and thread safety

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

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

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

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

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

4.2 Thread-Local Storage TLS

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

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

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

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

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

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

4.2.1 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(void);
```

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.

Example

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


4.2.2 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(void);
```

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.

Example

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

```
#include "RTOS.h"

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

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

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

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


Chapter 5

Stacks

5.1 Task stack

Each task uses its individual stack. The stack pointer is initialized and set every time a task is activated by the scheduler. The task stack needs to be able to accommodate the stack content of any (sub-)function plus the basic stack size.

The basic stack size is the size of memory required to store the context of the task on the stack. The minimum basic task stack size is 72 bytes for CPUs without a VFP/NEON unit and up to 332 bytes for CPUs with a VFP/NEON unit. We recommend at least 256 bytes stack as a start for CPUs without a VFP/NEON unit and 512 bytes for CPUs with a VFP/NEON unit.

> **Note**
>
> Stacks for ARM devices need to be 8-byte aligned. embOS ensures that task stacks are properly aligned. If an unaligned stack was aligned, the first few bytes up to the aligned address will not be used. Thus, 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

The embOS scheduler executes in supervisor (SVC) mode. However, embOS doesn't use the dedicated SVC stack symbol in order to initialize the SVC stack pointer. After `OS_Start()` was called embOS uses the stack symbol of the system stack which was used in the `main()` routine. This avoids the need to allocate dedicated SVC stack space.

The minimum system stack size required by embOS is about 160 bytes (stack check & profiling build, no VFP/NEON unit). 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 stack size definition in your linker file or within the project settings. We recommend a minimum stack size of 512 bytes for the system stack for CPUs without a VFP/NEON unit and 1024 bytes for CPUs with a VFP/NEON unit.

5.3 Interrupt stack

If a normal hardware exception occurs, the ARM core switches to IRQ mode, which has a separate stack pointer. After saving the scratch registers as well as `LR_irq` and `SPSR_irq` (and FPSCR, if VFP/NEON unit is present) onto the IRQ stack embOS switches to SVC mode. Only the previously mentioned registers are saved onto the IRQ stack. Thus, every interrupt requires 32 bytes on the IRQ stack. The maximum IRQ stack size required by the application can be calculated as "Maximum interrupt nesting level * 32 bytes". For the interrupt routine itself, the system stack is used, because they're executed in SVC mode.

The size of the IRQ stack can be changed by modifying the stack size definition in your linker file or within the project settings.

5.4 Stack specifics

There are two stacks which have to be declared in the linker script file or project settings:

- The system stack.
- The IRQ stack.

The system stack is used by the startup, the `main()` routine, embOS internal functions, and C-level interrupt handlers.

The IRQ stack is used when an interrupt exception is triggered. The exception handler saves some registers and then performs a mode switch which then uses the system stack for further execution.

When the CPU starts, it runs in supervisor mode. Then the startup code initializes the various stack pointer registers for each mode with their assigned stack and finally jumps into the `main()` routine. embOS expects the `main()` routine to use the system stack, no matter in which CPU mode.

When `OS_Init()` is called, embOS initializes the supervisor stack pointer to point to the system stack. After embOS is started with `OS_Start()`, the embOS scheduler runs in supervisor mode using the system stack while each task is running in system mode using its own dedicated stack.

> **Note**
>
> Stacks for ARM devices need to be 8-byte aligned.

Chapter 6

Interrupts

6.1 What happens when an interrupt occurs?

- The CPU-core receives an interrupt request.
- As soon as the interrupts are enabled, the interrupt is executed.
- The CPU switches to the IRQ mode which uses the IRQ stack.
- The CPU saves PC and flags in registers `LR_irq` and `SPSR_irq`.
- The CPU jumps to offset `0x18` in the exception vector table which contains an instruction to branch to the embOS low-level `IRQ_Handler()`.
- embOS `IRQ_Handler()`: Saves scratch registers as well as `LR_irq`, `SPSR_irq` and `FPSCR` on the IRQ stack.
- embOS `IRQ_Handler()`: Switches to supervisor mode and system stack.
- embOS `IRQ_Handler()`: Saves scratch VFP registers on the system stack.
- embOS `IRQ_Handler()`: Executes `OS_irq_handler()` (defined in `RTOSInit_*.c`).
- embOS `OS_irq_handler()`: Checks for interrupt source and executes the according ISR handler. The implementation of this functions depends on the implemented interrupt controller.
- embOS `IRQ_Handler()`: Restores scratch VFP registers from the system stack.
- embOS `IRQ_Handler()`: Switches to IRQ mode and IRQ stack.
- embOS `IRQ_Handler()`: Restores scratch registers as well as `LR_irq`, `SPSR_irq` and `FPSCR` from the IRQ stack.
- embOS `IRQ_Handler()`: Returns from interrupt.

> **Note**
>
> FPSCR and VFP registers are only preserved by embOS libraries with VFP support.

6.2 Defining interrupt handlers in C

Interrupt handlers called from the default C interrupt handler `OS_irq_handler()` located in `RTOSInit*.c` are just normal functions which do not take parameters and do not return any value. `OS_irq_handler()` first calls `OS_INT_Enter()` or `OS_INT_EnterNestable()` to inform embOS that interrupt code is running. Then this handler examines the source of interrupt and calls the related interrupt handler function. Finally, `OS_irq_handler()` calls `OS_INT_Leave()` or `OS_INT_LeaveNestable()` and returns to the primary low level interrupt handler `IRQ_Handler()`.

Depending on the interrupting source, it may be required to reset the interrupt pending condition of the related peripherals.

Example

Simple interrupt routine:

```
void Timer_IRQHandler(void) {
  static unsigned long Time = 0;

  //
  // Handle timer IRQ
  //
  Time++;
}
```

6.3 Interrupt handling without vectored interrupt controller

When using an ARM CPU without implementation of a vectored interrupt controller, the application is responsible to examine which interrupting source triggered the IRQ.

The reaction to an interrupt is as follows:

- `IRQ_Handler()` calls `OS_irq_handler()`.
- `OS_irq_handler()` informs embOS that interrupt code is running by calling `OS_INT_Enter()`.
- `OS_irq_handler()` determines the interrupt sources and handles all pending IRQs.
- `OS_irq_handler()` informs embOS that interrupt handling ended by calling `OS_INT_Leave()`.
- `IRQ_Handler()` returns to `IRQ_Handler()`.

Example

Simple interrupt routine:

```
void OS_irq_handler(void) {
  OS_INT_Enter();
  if (Timer_IsPending()) {  // Interrupt pending?
    Timer_IRQHandler();     // Handle interrupt
  }
  if (UART_IsPending()) {   // Interrupt pending ?
    UUART_IRQHandler();     // Handle interrupt
  }
  OS_INT_Leave();
}
```

During interrupt processing, you should not re-enable interrupts, as this would lead in recursion.

6.4 Interrupt handling with vectored interrupt controller

For ARM derivatives with built in vectored interrupt controller, embOS uses a different interrupt handling procedure and delivers additional functions to install and setup interrupt handler functions. You should not program the interrupt controller for IRQ handling directly. You should use the functions delivered with embOS.

The reaction to an interrupt with vectored interrupt controller is as follows:

- `IRQ_Handler()` calls `OS_irq_handler()`.
- `OS_irq_handler()` examines the interrupting source by reading the interrupt vector from the interrupt controller.
- `OS_irq_handler()` informs embOS that interrupt code is running by calling `OS_INT_Enter()`.
- `OS_irq_handler()` calls the interrupt handler function which is addressed by the interrupt vector.
- `OS_irq_handler()` resets the interrupt controller to re-enable acceptance of new interrupts.
- `OS_irq_handler()` informs embOS that interrupt handling ended by calling `OS_INT_Leave()`.
- `OS_irq_handler()` returns to `IRQ_Handler()`.

> **Note**
>
> Different ARM CPUs may have different versions of vectored interrupt controller hardware, and usage of embOS supplied functions varies depending on the type of interrupt controller. Refer to the samples delivered with embOS which are used in the CPU specific RTOSInit module.

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

Function	Description
`OS_ARM_InstallISRHandler()`	Installs an interrupt handler
`OS_ARM_EnableISR()`	Enables an interrupt
`OS_ARM_DisableISR()`	Disables an interrupt
`OS_ARM_ISRSetPrio()`	Sets the priority of an interrupt
`OS_ARM_ClearPendingFlag()`	Clears an interrupt pending flag
`OS_ARM_IsPending()`	Checks if an interrupt is pending
`OS_ARM_AssignISRSource()`	Assigns a hardware interrupt channel to an interrupt vector
`OS_ARM_EnableISRSource()`	Enables an interrupt channel of a VIC type interrupt controller
`OS_ARM_DisableISRSource()`	Disables an interrupt channel of a VIC type interrupt controller
`OS_ARM_SetISRCfg()`	Sets the interrupt configuration.
`OS_ARM_SetVBAR()`	Writes the vector table address register.

6.4.1 OS_ARM_InstallISRHandler()

Description

`OS_ARM_InstallISRHandler()` is used to install a specific interrupt vector when ARM CPUs with vectored interrupt controller are used.

Prototype

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

Parameters

Parameter	Description
`ISRIndex`	Index of the interrupt source, usually the interrupt vector number.
`pISRHandler`	Address of the interrupt handler function.

Return Value

`OS_ISR_HANDLER*`: The address of the interrupt handler that was previously installed with the addressed interrupt source.

Additional Information

This function just installs the interrupt vector but does not modify the priority and does not automatically enable the interrupt.

6.4.2 OS_ARM_EnableISR()

Description

`OS_ARM_EnableISR()` is 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.

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**
>
> For ARM CPUs with VIC type interrupt controller, this function just enables the interrupt vector itself. To enable the hardware assigned to that vector, you have to call `OS_ARM_EnableISRSource()` also.

6.4.3 OS_ARM_DisableISR()

Description

`OS_ARM_DisableISR()` is 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.

Additional Information

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

> **Note**
>
> When using an ARM CPU with built in interrupt controller of VIC type, use `OS_ARM_DisableISRSource()` to disable a specific interrupt.

6.4.4 OS_ARM_ISRSetPrio()

Description

OS_ARM_ISRSetPrio() is 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.
Prio	The priority which should be set for the specific interrupt.

Return Value

Previous priority which was assigned before the call of OS_ARM_ISRSetPrio().

Additional Information

This function sets the priority of an interrupt channel by programming the interrupt controller. Refer to CPU-specific manuals about allowed priority levels.

6.4.5 OS_ARM_ClearPendingFlag()

Description

`OS_ARM_ClearPendingFlag()` is used to clear an interrupt pending flag

Prototype

```
void OS_ARM_ClearPendingFlag(int ISRIndex);
```

Parameters

Parameter	Description
`ISRIndex`	Index of the interrupt source which should be cleared

Additional Information

This function just clears the interrupt pending flag inside the interrupt controller. It does not clear the interrupt pending flag in any peripheral.

6.4.6 OS_ARM_IsPending()

Description

`OS_ARM_IsPending()` is used to check if an interrupt is pending

Prototype

```
unsigned int OS_ARM_IsPending(int ISRIndex);
```

Parameters

Parameter	Description
`ISRIndex`	Index of the interrupt source which should be checked

Return value

= 0 Interrupt is not pending.
= 1 Interrupt is pending.

Additional Information

This function just checks the interrupt pending flag inside the interrupt controller. It does not check the interrupt pending flag in any peripheral.

6.4.7 OS_ARM_AssignISRSource()

Description

`OS_ARM_AssignISRSource()` is used to assign a hardware interrupt channel to an interrupt vector in an interrupt controller of VIC type.

Prototype

```
void OS_ARM_AssignISRSource(int ISRIndex,
                            int Source);
```

Parameters

Parameter	Description
`ISRIndex`	Index of the interrupt source which should be modified.
`Source`	The source channel number which should be assigned to the specified interrupt vector.

Additional Information

This function assigns a hardware interrupt line to an interrupt vector of VIC type only. It cannot be used for other types of vectored interrupt controllers. The hardware interrupt channel number of specific peripherals depends on specific CPU derivatives and has to be taken from the hardware manual of the CPU.

Example

```
/* Install UART interrupt handler */
OS_ARM_InstallISRHandler(UART_ID, &COM_ISR);         // UART interrupt vector
OS_ARM_ISRSetPrio(UART_ID, UART_PRIO);               // UART interrupt priority
OS_ARM_EnableISR(UART_ID);                           // Enable UART interrupt
/* Install UART interrupt handler with VIC type interrupt controller*/
OS_ARM_InstallISRHandler(UART_INT_INDEX, &COM_ISR);  // UART interrupt vector
OS_ARM_AssignISRSource(UART_INT_INDEX, UART_INT_SOURCE);
OS_ARM_EnableISR(UART_INT_INDEX);         // Enable UART interrupt vector
OS_ARM_EnableISRSource(UART_INT_SOURCE);  // Enable UART interrupt source
```


6.4.8 OS_ARM_EnableISRSource()

Description

OS_ARM_EnableISRSource() is used to enable an interrupt input channel of an interrupt controller of VIC type.

Prototype

```
void OS_ARM_EnableISRSource(int SourceIndex);
```

Parameters

Parameter	Description
SourceIndex	Index of the interrupt channel which should be enabled.

Additional Information

This function enables a hardware interrupt input of a VIC-type interrupt controller. It cannot be used for other types of vectored interrupt controllers. The hardware interrupt channel number of specific peripherals depends on specific CPU derivatives and has to be taken from the hardware manual of the CPU.

Example

```
/* Install UART interrupt handler */
OS_ARM_InstallISRHandler(UART_ID, &COM_ISR);         // UART interrupt vector
OS_ARM_ISRSetPrio(UART_ID, UART_PRIO);               // UART interrupt priority
OS_ARM_EnableISR(UART_ID);                           // Enable UART interrupt
/* Install UART interrupt handler with VIC type interrupt controller*/
OS_ARM_InstallISRHandler(UART_INT_INDEX, &COM_ISR);  // UART interrupt vector
OS_ARM_AssignISRSource(UART_INT_INDEX, UART_INT_SOURCE);
OS_ARM_EnableISR(UART_INT_INDEX);          // Enable UART interrupt vector
OS_ARM_EnableISRSource(UART_INT_SOURCE);  // Enable UART interrupt source
```


6.4.9 OS_ARM_DisableISRSource()

Description

`OS_ARM_DisableISRSource()` is used to disable an interrupt input channel of an interrupt controller of VIC type.

Prototype

```
void OS_ARM_DisableISRSource(int SourceIndex);
```

Parameters

Parameter	Description
`SourceIndex`	Index of the interrupt channel which should be disabled.

Additional Information

This function disables a hardware interrupt input of a VIC-type interrupt controller. It cannot be used for other types of vectored interrupt controllers. The hardware interrupt channel number of specific peripherals depends on specific CPU derivatives and has to be taken from the hardware manual of the CPU.

6.4.10 OS_ARM_SetISRCfg()

Description

`OS_ARM_SetISRCfg()` sets the interrupt configuration.

Prototype

```
void OS_ARM_SetISRCfg(int    ISRIndex,
                      OS_U32 Cfg);
```

Parameters

Parameter	Description
`ISRIndex`	Index of the interrupt source.
`Cfg`	0: Corresponding interrupt is level-sensitive. 1: Corresponding interrupt is edge-triggered.

6.4.11 OS_ARM_SetVBAR()

Description

`OS_ARM_SetVBAR()` writes the vector table address register.

Prototype

```
void OS_ARM_SetVBAR(OS_U32 Addr);
```

Parameters

Parameter	Description
`Addr`	Address of the vector table.

6.5 Interrupt-stack switching

Because ARM core based controllers have a separate stack pointer for interrupts, there is no need for explicit stack-switching in an interrupt routine. The routines `OS_INT_EnterIntStack()` and `OS_INT_LeaveIntStack()` are supplied for source compatibility to other processors only and have no functionality.

The ARM interrupt stack is used for the low-level interrupt handler `IRQ_Handler()` in the embOS library only.

6.6 Fast Interrupt (FIQ)

The FIQ interrupt cannot be used with embOS functions, it is reserved for high speed user functions.

Note the following:

- FIQ is never disabled by embOS.
- Never call any embOS function from an FIQ handler.
- Do not assign any embOS interrupt handler to FIQ.

> **Note**
>
> When you decide to use FIQ, ensure the FIQ stack is initialized during startup and that an interrupt vector for FIQ handling is included in your application.

Chapter 7

MMU/MPU and cache support

7.1 Introduction

This chapter describes the MMU/MPU and cache support for ARM CPUs. With the ARM core the MMU is part of the Virtual Memory System Architecture (VMSA) and the MPU is part of the Protected Memory System Architecture (PMSA). embOS comes with functions to support the MMU/MPU and cache of ARMv4, ARMv5 and ARMv7-A/ARMv7-R CPUs.

7.2 MMU handling for ARMv5/ARMv7A CPUs

The MMU allows virtual-to-physical address mapping with sections of one MByte and cache control. The MMU requires a translation table which can be located in any data area, RAM or ROM, but has to be aligned at a 16Kbyte boundary. A translation table in RAM has to be set up during run time. embOS delivers API functions to set up this table.

Function	Description
`OS_ARM_MMU_InitTT()`	Initialize the MMU translation table.
`OS_ARM_MMU_AddTTEntries()`	Add address entries to the table.
`OS_ARM_MMU_Enable()`	Enable the MMU.
`OS_ARM_MMU_GetVirtualAddr()`	Translates a physical address into a virtual address
`OS_ARM_MMU_v2p()`	Translates a virtual address into a physical address.

7.2.1 OS_ARM_MMU_InitTT()

Description

`OS_ARM_MMU_InitTT()` is used to initialize an MMU translation table which is located in RAM. The table is filled with zeroes, thus all entries are marked as `OS_ARM_MMU_UNMAPPED` initially.

Prototype

```
void OS_ARM_MMU_InitTT(unsigned int* pTranslationTable);
```

Parameters

Parameter	Description
`pTranslationTable`	Points to the base address of the translation table.

7.2.2 OS_ARM_MMU_AddTTEntries()

Description

OS_ARM_MMU_AddTTEntries() is used to add entries to the MMU address translation table. The start address of the virtual address, physical address, area size and cache modes are passed as parameter.

Prototype

```
void OS_ARM_MMU_AddTTEntries(unsigned int* pTranslationTable,
                             unsigned int  CacheMode,
                             unsigned int  VIndex,
                             unsigned int  PIndex,
                             unsigned int  NumEntries);
```

Parameters

Parameter	Description
pTranslationTable	Points to the base address of the translation table.
CacheMode	Specifies the cache operating mode and memory access permissions which should be used for the selected area. May be one of the following modes: ARMv4/ARMv5: OS_ARM_MMU_UNMAPPED: The associated MVA is unmapped, and attempting to access it generates a translation fault OS_ARM_CACHEMODE_NC_NB: non-cacheable, non-bufferable OS_ARM_CACHEMODE_C_NB: cacheable, non-bufferable OS_ARM_CACHEMODE_NC_B: non-cacheable, bufferable OS_ARM_CACHEMODE_C_B: cacheable, bufferable ARMv7A: OS_ARM_MMU_UNMAPPED: The associated MVA is unmapped, and attempting to access it generates a translation fault 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-Allocate OS_ARM_CACHEMODE_NON_SHAREABLE_DEVICE: Non-shareable Device OS_ARM_MMU_NOACCESS: All accesses generate Permission faults OS_ARM_MMU_READWRITE: Full access OS_ARM_MMU_READONLY: Read-only access OS_ARM_MMU_EXECUTE_NEVER: Determines whether the memory region is executable

Parameter	Description
VIndex	Virtual address index, which is the start address of the virtual memory address range with MBytes resolution. VIndex = (virtual address >> 20)
PIndex	Physical address index, which is the start address of the physical memory area range with MBytes resolution. PIndex = (physical address >> 20)
NumEntries	Specifies the size of the memory area in MBytes.

Additional information

The function adds entries for every section of one MegaByte size into the translation table for the specified memory area.

The macros for normal memory, i.e. OS_ARM_CACHEMODE_WRITE_THROUGH, OS_ARM_CACHEMODE_WRITE_BACK_NO_ALLOC, OS_ARM_CACHEMODE_NON_CACHEABLE and OS_ARM_CACHEMODE_WRITE_BACK_ALLOC, can be OR-red with OS_ARM_MMU_SHAREABLE to mark normal memory as shareable.

OS_ARM_MMU_NOACCESS, OS_ARM_MMU_READWRITE, OS_ARM_MMU_READONLY and OS_ARM_MMU_EXECUTE_NEVER can be used in combination with the cache attribute defines. If no memory access permissions are set full memory access is allowed per default.
OS_ARM_MMU_InitTT() sets all entries to OS_ARM_MMU_UNMAPPED. The MMU table does not need to define such entries.

7.2.3 OS_ARM_MMU_Enable()

Description

OS_ARM_MMU_Enable() is used to enable the MMU which will then perform the address mapping.

Prototype

```
void OS_ARM_MMU_Enable(unsigned int* pTranslationTable);
```

Parameters

Parameter	Description
pTranslationTable	Points to the base address of the translation table.

Additional information

As soon as the function was called, the address translation is active. The MMU table has to be setup before calling OS_ARM_MMU_Enable().

OS_ARM_MMU_Enable() also enables the branch prediction unit of Cortex-A CPUs.

7.2.4 OS_ARM_MMU_GetVirtualAddr()

Description

OS_ARM_MMU_GetVirtualAddr() is used to translate a physical address into a virtual address with specified cache mode.

Prototype

```
void* OS_ARM_MMU_GetVirtualAddr(unsigned long PAddr,
                                unsigned int  NumEntries);
```

Parameters

Parameter	Description
PAddr	The physical address as unsigned long.
CacheMode	The cache mode of the requested virtual address May be one of the defined cache modes: ARMv4/ARMv5: OS_ARM_CACHEMODE_NC_NB OS_ARM_CACHEMODE_C_NB OS_ARM_CACHEMODE_NC_B OS_ARM_CACHEMODE_C_B OS_ARM_CACHEMODE_ANY ARMv7A: OS_ARM_CACHEMODE_STRONGLY_ORDERED OS_ARM_CACHEMODE_SHAREABLE_DEVICE OS_ARM_CACHEMODE_WRITE_THROUGH OS_ARM_CACHEMODE_WRITE_BACK_NO_ALLOC OS_ARM_CACHEMODE_NON_CACHEABLE OS_ARM_CACHEMODE_WRITE_BACK_ALLOC OS_ARM_CACHEMODE_NON_SHAREABLE_DEVICE OS_ARM_CACHEMODE_ANY

Return value

void* to the first virtual address found. A value of 0xFFFFFFFF indicates that no entry was found.

Additional information

The function may be useful to examine an address of memory mapped to a virtual address with specific cache mode. For the CPU it may be necessary to write into a specific memory in uncached mode. This can be done by setting up the MMU table with different virtual address for the same physical memory with different cache modes. For efficiency reasons, the CPU should access the memory fully cached for normal operation. When a peripheral or DMA accesses the same memory for reading, for exaplme an LCD controller accesses the diplay buffer, or an Ethernet MAC access a transferbuffer, the CPU has to write the data uncached into this memory, or has to clean the cache after writing. The function OS_ARM_MMU_GetVirtualAddress() can be used to find the address for uncached access. The MMU table has to be setup before the function is called.

7.2.5 OS_ARM_MMU_v2p()

Description

OS_ARM_MMU_v2p() is used to translate a virtual address into a physical address.

Prototype

```
unsigned long OS_ARM_MMU_v2p(void* pVAddr);
```

Parameters

Parameter	Description
pVAddr	Pointer which represents the virtual address.

Return value

The physical address which is mapped to the virtual address passed as parameter.

Additional information

The function can be used to examine the physical addresss of memory. The CPU normally operates with virtual addresses which may differ from the physical address of the memory. When a peripheral or DMA has to be programmed to access the same memory, the peripheral has to be programmed to access the physical memory. The function OS_ARM_MMU_v2p() can be used to find the physical address of a memory area. The MMU table has to be setup before the function is called.

7.3 MPU handling for ARMv7-R CPUs

The ARMv7-R MPU is used to set cache and access settings for memory regions.

Function	Description
OS_ARM_MPU_AddEntry()	Sets an ARMv7-R PMSA MPU memory region.
OS_ARM_MPU_Enable()	Enables the ARMv7-R PMSA MPU.
OS_ARM_MPU_GetMinRegionSize()	Returns the ARMv7-R PMSA minimum memory region size.
OS_ARM_MPU_GetNumRegions()	Returns the number of available memory regions.
OS_ARM_MPU_Init()	Initializes the ARMv7-R PMSA MPU.

7.3.1 OS_ARM_MPU_AddEntry()

Description

OS_ARM_MPU_AddEntry() sets an ARMv7-R PMSA MPU memory region.

Prototype

```
void OS_ARM_MPU_AddEntry(OS_U32 Region,
                         void*  BaseAddr,
                         OS_U32 Size,
                         OS_U32 Permissions,
                         OS_U32 Attributes);
```

Parameters

Parameter	Description
Region	Region index
BaseAddr	Memory region address
Size	Memory region size in bytes
Permissions	OS_ARM_MPU_NOACCESS: No read or write access OS_ARM_MPU_READWRITE: Read and write access OS_ARM_MPU_READONLY: Read access only OS_ARM_MPU_EXECUTE_NEVER: No code execution allowed
Attributes	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-Allocate OS_ARM_CACHEMODE_NON_SHAREABLE_DEVICE: Non-shareable Device

Additional information

The region index starts at zero for the first region. The number of available regions can be read with OS_ARM_MPU_GetNumRegions(). The regions size must be aligned to the PMSA regions size which can be read with OS_ARM_MPU_GetMinRegionSize().

The macros for normal memory, i.e. OS_ARM_CACHEMODE_WRITE_THROUGH, OS_ARM_CACHEMODE_WRITE_BACK_NO_ALLOC, OS_ARM_CACHEMODE_NON_CACHEABLE and OS_ARM_CACHEMODE_WRITE_BACK_ALLOC, can be OR-red with OS_ARM_MPU_SHAREABLE to mark normal memory as shareable.

Example

```
int __low_level_init(void) {
  //
  // Sets access and cache settings for 256 Bytes at address 0x00
  //
  OS_ARM_MPU_AddEntry(0u,(void*)0x00000000, 256,
```



```
                        OS_ARM_MPU_READWRITE,
                        OS_ARM_CACHEMODE_WRITE_BACK_ALLOC);
  return 1;
}
```


7.3.2 OS_ARM_MPU_Enable()

Description

`OS_ARM_MPU_Enable()` enables the ARMv7-R PMSA MPU.

Prototype

```
void OS_ARM_MPU_Enable(void);
```

Additional information

`OS_ARM_MPU_Enable()` has to be called after the the MPU was initialized and configured.

7.3.3 OS_ARM_MPU_GetMinRegionSize()

Description

`OS_ARM_MPU_GetMinRegionSize()` returns the ARMv7-R PMSA minimum memory region size.

Prototype

```
OS_U32 OS_ARM_MPU_GetMinRegionSize(void);
```

Return value

Minimum memory region size which can be used with this PMSA implementation.

7.3.4 OS_ARM_MPU_GetNumRegions()

Description

`OS_ARM_MPU_GetNumRegions()` returns the number of available memory regions.

Prototype

```
OS_U32 OS_ARM_MPU_GetNumRegions(void);
```

Return value

Number of available memory regions.

7.3.5 OS_ARM_MPU_Init()

Description

`OS_ARM_MPU_Init()` initializes the ARMv7-R PMSA MPU.

Prototype

```
void OS_ARM_MPU_Init(void);
```


7.4 Cache handling for ARMv5/ARMv7 CPUs

ARM CPUs with MMU/MPU and cache have separate data and instruction caches. embOS delivers the following functions to setup and handle the MMU and caches.

Function	Description
`OS_ARM_ICACHE_Enable()`	Enable the instruction cache.
`OS_ARM_ICACHE_Invalidate()`	Invalidates the complete instruction cache.
`OS_ARM_DCACHE_Enable()`	Enable the data cache.
`OS_ARM_DCACHE_Invalidate()`	Invalidates the complete data cache.
`OS_ARM_DCACHE_Clean()`	Clean data cache.
`OS_ARM_DCACHE_CleanRange()`	Clean data cache range.
`OS_ARM_DCACHE_InvalidateRange()`	Invalidate the data cache.
`OS_ARM_CACHE_Sync()`	Syncs data and instruction cache.
`OS_ARM_AddL2Cache()`	Sets 2nd level cache API table.
`OS_ARM_CACHE_GetLineSize()`	Returns cache line size of the specified cache level.

7.4.1 OS_ARM_ICACHE_Enable()

Description

`OS_ARM_ICACHE_Enable()` is used to enable the instruction cache of the CPU.

Prototype

```
void OS_ARM_ICACHE_Enable(void);
```

Additional information

As soon as the function was called, the instruction cache is active. It is CPU implementation defined whether the instruction cache works without MMU. Normally, the MMU should be setup before activating instruction cache.

7.4.2 OS_ARM_ICACHE_Invalidate()

Description

`OS_ARM_ICACHE_Invalidate()` invalidates the complete instruction cache. Invalidating means, mark all entries in the specified area as invalid. Invalidation forces re-reading the code from memory into the cache, when the specified area is accessed again.

Prototype

```
void OS_ARM_ICACHE_Invalidate(void);
```


7.4.3 OS_ARM_DCACHE_Enable()

Description

`OS_ARM_DCACHE_Enable()` is used to enable the data cache of the CPU.

Prototype

```
void OS_ARM_DCACHE_Enable(void);
```

Additional information

The function must not be called before the MMU translation table was set up correctly and the MMU was enabled. As soon as the function was called, the data cache is active, according to the cache mode settings which are defined in the MMU translation table. It is CPU implementation defined whether the data cache is a write through, a write back, or a write through/write back cache. Most modern CPUs will have implemented a write through/write back cache.

7.4.4 OS_ARM_DCACHE_Invalidate()

Description

`OS_ARM_DCACHE_Invalidate()` invalidates the complete data cache. Invalidating means, mark all entries in the specified area as invalid. Invalidation forces re-reading the data from memory into the cache, when the specified area is accessed again.

Prototype

```
void OS_ARM_DCACHE_Invalidate(void);
```


7.4.5 OS_ARM_DCACHE_Clean()

Description

`OS_ARM_DCACHE_Clean()` is used to clean the data cache memory without invalidating the instruction cache.

Prototype

```
void OS_ARM_DCACHE_Clean(void);
```

Additional information

Cleaning the data cache is needed, when data should be transferred by a DMA or other BUS master that does not use the data cache. When the CPU writes data into a cacheable area, the data might not be written into the memory immediately. When then a DMA cycle is started to transfer the data from memory to any other location or peripheral, the wrong data will be written.

The cache is cleaned line by line. Cleaning one cache line takes approximately 10 CPU cycles. The total time to invalidate a range may be calculated as:

t = (NumBytes / Cache line size) * (10 [CPU clock cycles] + Memory write time).

The real time depends on the content of the cache. If data in the cache is marked as dirty, the cache line has to be written to memory. The memory write time depends on the memory BUS clock and memory speed. If data has to be written to memory, the required cycles for this memory operation has to be added to the 10 CPU clock cycles for every cache line to be cleaned.

7.4.6 OS_ARM_DCACHE_CleanRange()

Description

OS_ARM_DCACHE_CleanRange() is used to clean a range in the data cache memory to ensure that the data is written from the data cache into the memory.

Prototype

```
void OS_ARM_DCACHE_CleanRange(void*        p,
                              unsigned int NumEntries);
```

Parameters

Parameter	Description
p	Points to the base address of the memory area that should be updated.
NumBytes	Number of bytes which have to be written from cache to memory.

Additional information

Cleaning the data cache is needed, when data should be transferred by a DMA or other BUS master that does not use the data cache. When the CPU writes data into a cacheable area, the data might not be written into the memory immediately. When then a DMA cycle is started to transfer the data from memory to any other location or peripheral, the wrong data will be written.

Before starting a DMA transfer, a call of OS_ARM_DCACHE_CleanRange() ensures, that the data is transferred from the data cache into the memory and the write buffers are drained.

The cache is cleaned line by line. Cleaning one cache line takes approximately 10 CPU cycles. The total time to invalidate a range may be calculated as:

t = (NumBytes / Cache line size) * (10 [CPU clock cycles] + Memory write time).

The real time depends on the content of the cache. If data in the cache is marked as dirty, the cache line has to be written to memory. The memory write time depends on the memory BUS clock and memory speed. If data has to be written to memory, the required cycles for this memory operation has to be added to the 10 CPU clock cycles for every cache line to be cleaned.

> **Note**
>
> Unfortunately, only complete cache lines can be cleaned. Therefore, it is required, that the base address of the memory area has to be located at a cache line size byte boundary and the number of bytes to be cleaned has to be a multiple of the cache line size. The debug version of embOS will call OS_Error() with error code OS_ERR_NON_ALIGNED_INVALIDATE, if one of these restrictions is violated.

7.4.7 OS_ARM_DCACHE_InvalidateRange()

Description

OS_ARM_DCACHE_InvalidateRange() is used to invalidate a memory area in the data cache. Invalidating means, mark all entries in the specified area as invalid. Invalida- tion forces re-reading the data from memory into the cache, when the specified area is accessed again.

Prototype

```
void OS_ARM_DCACHE_InvalidateRange(void*        p,
                                   unsigned int NumBytes);
```

Parameters

Parameter	Description
p	Points to the base address of the memory area that should be updated.
NumBytes	Number of bytes which have to be written from cache to memory.

Additional information

This function is needed, when a DMA or other BUS master is used to transfer data into the main memory and the CPU has to process the data after the transfer.

To ensure, that the CPU processes the updated data from the memory, the cache has to be invalidated. Otherwise the CPU might read invalid data from the cache instead of the memory.

Special care has to be taken, before the data cache is invalidated. Invalidating a data area marks all entries in the data cache as invalid. If the cache contained data which was not written into the memory before, the data gets lost. The cache is invalidated line by line. Invalidating one cache line takes approximately 10 CPU cycles. The total time to invalidate a range may be calculated as: t = (NumBytes / Cache line size) * 10 [CPU clock cycles]. Notes

> **Note**
>
> Unfortunately, only complete cache lines can be invalidated. Therefore, it is required, that the base address of the memory area has to be located at a cache line size byte boundary and the number of bytes to be invalidated has to be a multiple of the cache line size. The debug version of embOS will call OS_Error() with error code OS_ERR_NON_ALIGNED_INVALIDATE, if one of these restrictions is violated.

7.4.8 OS_ARM_CACHE_Sync()

Description

OS_ARM_CACHE_Sync() cleans the data cache and invalidates the instruction cache to to ensure cache coherency.

Prototype

```
void OS_ARM_CACHE_Sync(void);
```

Additional information

This function is for example needed, when code is copied into RAM and code is then executed from RAM.

7.4.9 OS_ARM_AddL2Cache()

Description

OS_ARM_AddL2Cache() is used to add.

Prototype

```
void OS_ARM_v7_AddL2Cache(const OS_ARM_L2CACHE_API* pCacheAPI,
                          void*                     pParam);
```

Parameters

Parameter	Description
pCacheAPI	Pointer to 2nd level Cache API table.
pParam	Additional parameter (e.g. base address or cache registers).

Additional information

This function s needed to enable the L2 cache. Nothing else is necessary to do since the actual L2 cache routines are automatically called by the L1 cache routines. For example OS_ARM_DCACHE_InvalidateRange() calls also internally the according L2 cache routine.

Example

```
#define L2CACHE_BASE_ADDR 0x3FFFF000u

//
// Set API functions and base address for L2 Cache
//
OS_ARM_AddL2Cache(&OS_L2CACHE_L2C310, (void*)L2CACHE_BASE_ADDR);
```


7.4.10 OS_ARM_CACHE_GetLineSize()

Description

`OS_ARM_CACHE_GetLineSize()` returns the cache line size of the specified cache level.

Prototype

```
OS_U32 OS_ARM_CACHE_GetLineSize(OS_U32 CIndex);
```

Parameters

Parameter	Description
`CIndex`	Index of the cache level of which the cache line size shall be returned.

Additional information

The returned cache line size can be used to calculate the alignment and number of bytes passed to the `OS_ARM_DCACHE_InvalidateRange()` and `OS_ARM_DCACHE_CleanRange()` functions.

7.5 MMU and cache handling program sample

The MMU and cache handling has to be set up before the data segments are initial- ized. Otherwise a virtual address mapping would not work. The startup code must call a `__low_level_init()` function before sections are initialized.

It is a good idea to initialize memory access, the MMU table and the cache control during `__low_level_init()` . The following sample is an excerpt from one `__low_level_init()` function which is part of an `RTOSInit.c` file:

```
/*********************************************************************
*
*  MMU and cache configuration
*
*  The MMU translation table has to be aligned to 16KB boundary
*  and has to be located in uninitialized data area
*/
#pragma data_alignment=16384
__no_init static unsigned int _TranslationTable [0x1000]; // OS_INTERWORK int
int __low_level_init(void) {
  //
  // Init MMU and caches
  //
  OS_ARM_MMU_InitTT (&_TranslationTable[0]);
  //
  // Internal SRAM, the first MB remapped to 0,
  // cacheable, bufferable, region not executable
  //
  OS_ARM_MMU_AddTTEntries ( &_TranslationTable[0],
                            OS_ARM_CACHEMODE_C_B | OS_ARM_MMU_EXECUTE_NEVER,
                            0x000, 0x200, 0x001);
  //
  // Internal SRAM, original address, NON cachable, NON bufferable
  //
  OS_ARM_MMU_AddTTEntries ( &_TranslationTable[0],
                            OS_ARM_CACHEMODE_NC_NB,
                            0x200, 0x200, 0x001);
  OS_ARM_MMU_Enable (&_TranslationTable[0]);
  OS_ARM_ICACHE_Enable();
  OS_ARM_DCACHE_Enable();
  return 1;
}
```

Other samples are included in the CPU specific `RTOSInit*.c` files delivered with embOS.

7.6 MPU and cache handling program sample

```
int __low_level_init(void) {
  //
  // Enable MPU, Caches and branch prediction unit
  //
  OS_ARM_DCACHE_Enable();
  OS_ARM_ICACHE_Enable();
  OS_ARM_MPU_Init();
  //
  // Add MPU regions to set cache settings for different memory sections
  //
  OS_ARM_MPU_AddEntry(0u, (void*)0x00000000, 0x00140000,
    OS_ARM_MPU_READONLY,  OS_ARM_CACHEMODE_WRITE_BACK_ALLOC);  // FLASH
  OS_ARM_MPU_AddEntry(1u, (void*)0x08000000, 0x00030000,
    OS_ARM_MPU_READWRITE, OS_ARM_CACHEMODE_WRITE_BACK_ALLOC);  // RAM
  OS_ARM_MPU_Enable();

  return 1;
}
```


Chapter 8

VFP and NEON support

8.1 Introduction

Some ARM MCUs come with integrated Arm VFP and NEON units.

When activating the VFP or NEON support in the project options, the compiler and linker will add efficient code which uses the VFP/NEON register bank and VFP/NEON instructions where possible in the application.

With embOS, the VFP/NEON registers are automatically saved and restored when preemptive or cooperative task switches are performed. embOS also automatically saves and restores VFP/NEON registers for all embOS interrupt routines.

The VFP register bank consists of either 16 or 32 double-precision registers. The VFP register bank is also shared between the VFP and NEON units. If a NEON unit is implemented the VFP register bank consists of 32 64-bit double-precision registers. For a VFP unit with 32 double-precision registers or NEON unit all 32 double-precision registers are preserved, while for a VFP unit with 16 double-precision registers only the 16 double-precision registers need to be preserved.

embOS comes with libraries which preserve 16 double-precision registers D0-D15, 32 double-precision registers D0-D31 or none. Please have a look in the chapter *Libraries* on page 18 for more details.

> **Note**
>
> embOS ARM until V5.16.1.0 used task context extensions and ISR macros to preserve VFP/NEON registers. These API functions and macros are kept for compatibility but have no functionality anymore.

8.2 Using embOS libraries with VFP/NEON support

When VFP/NEON support is selected as project option, one of the embOS libraries with VFP/NEON support have to be used in the project. The embOS libraries with VFP/NEON support require that the VFP/NEON unit is switched on during startup and remains switched on during program execution. When the VFP/NEON unit is not switched on, the embOS scheduler will fail. Using your own startup code, ensure that the VFP/NEON unit is switched on during startup.

The debug version of embOS checks in `OS_Init()` whether the VFP/NEON unit is switched on. If not, embOS calls `OS_Error()` with the error code `OS_ERR_HW_NOT_AVAILABLE`.

8.3 Using the VFP/NEON unit in interrupt service routines

Using the VFP/NEON unit in embOS interrupt service routines does not require any additional functions to save and restore the VFP/NEON registers. embOS automatically saves and restores these registers.

VFP/NEON registers are not automatically saved and restored in zero latency interrupts. If the VFP/NEON unit is used in zero latency interrupts, it is the user’s responsibility to preserve these registers.

Chapter 9

RTT and SystemView

9.1 SEGGER Real Time Transfer

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

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

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

9.2 SEGGER SystemView

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

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

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

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

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

Chapter 10

embOS Thread Script

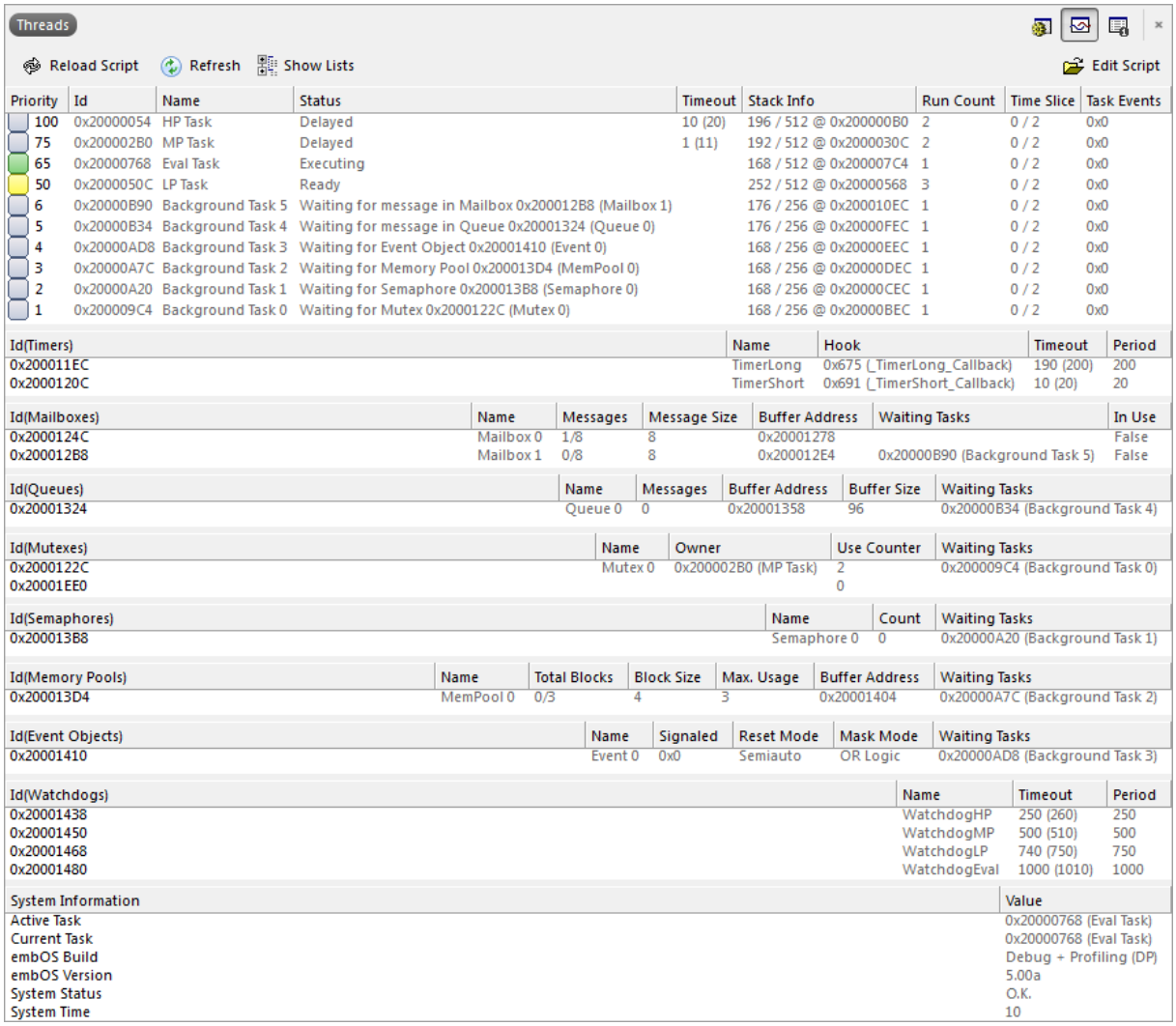
10.1 Introduction

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

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

10.2 How to use it

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



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

10.2.1 Task List

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

The task list displays various information about the running tasks:

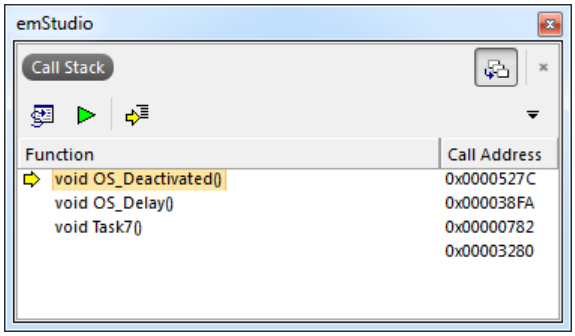
Column	Description
Priority	This is the priority of the task
Id	The address of a tasks task control block
Name	The name of the task
Status	The current status of the task
Timeout	Time in ms till the task gets called again
Stack Info	Shows the maximum usage (left) of the total stack for this task (right)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

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

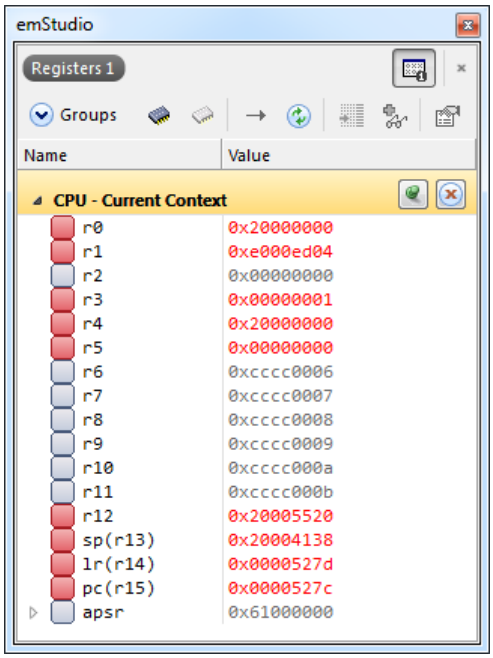
10.2.2 Task sensitivity

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

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



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



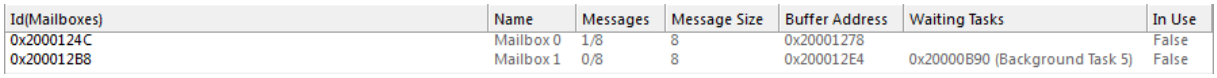
10.2.3 Timers

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

The timers list displays various information about active timers:

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

10.2.4 Mailboxes

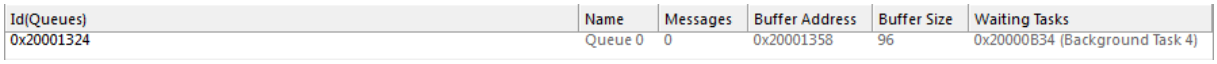


Id(Mailboxes)	Name	Messages	Message Size	Buffer Address	Waiting Tasks	In Use
0x2000124C	Mailbox 0	1/8	8	0x20001278		False
0x200012B8	Mailbox 1	0/8	8	0x200012E4	0x20000B90 (Background Task 5)	False

The mailboxes list displays various information about used mailboxes:

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

10.2.5 Queues

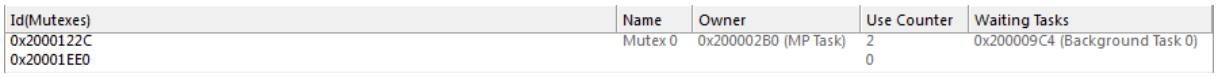


Id(Queues)	Name	Messages	Buffer Address	Buffer Size	Waiting Tasks
0x20001324	Queue 0	0	0x20001358	96	0x20000B34 (Background Task 4)

The queues list displays various information about used queues:

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

10.2.6 Mutexes



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

The mutexes list displays various information about used mutexes:

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

10.2.7 Semaphores

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

The semaphores list displays various information about used semaphores:

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

10.2.8 Readers-writer lock

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

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

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

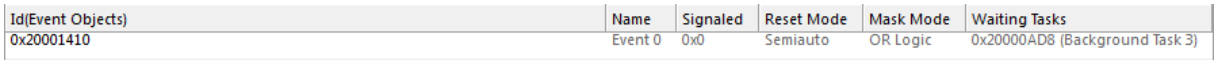
10.2.9 Memory Pools

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

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

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

10.2.10 Event Objects



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

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

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

10.2.11 Watchdogs

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

The watchdogs list displays various information about used watchdogs:

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

10.2.12 System Information

The system information list displays various information about embOS.

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

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

