

# embOS

Real-Time Operating System

CPU & Compiler specifics for ARM  
using Segger Embedded Studio

Document: UM01062  
Software Version: 5.02a  
Revision: 0  
Date: August 8, 2018



A product of SEGGER Microcontroller GmbH

[www.segger.com](http://www.segger.com)

## Disclaimer

Specifications written in this document are believed to be accurate, but are not guaranteed to be entirely free of error. The information in this manual is subject to change for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions. SEGGER makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. SEGGER specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

## Copyright notice

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

© 2010-2018 SEGGER Microcontroller GmbH, Hilden / Germany

## Trademarks

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

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

## Contact address

SEGGER Microcontroller GmbH

In den Weiden 11  
D-40721 Hilden

Germany

Tel.           +49 2103-2878-0  
Fax.           +49 2103-2878-28  
E-mail:       support@segger.com  
Internet:     [www.segger.com](http://www.segger.com)

## 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: August 8, 2018

Software	Revision	Date	By	Description
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.
Parameter	Parameters in API functions.
Sample	Sample code in program examples.
Sample comment	Comments in program examples.
User Input	Text entered at the keyboard by a user in a session transcript.
Secret Input	Text entered at the keyboard by a user, but not echoed (e.g. password entry), in a session transcript.
Reference	Reference to chapters, sections, tables and figures or other documents.
Emphasis	Very important sections.



# Table of contents

---

1	Using embOS .....	9
1.1	Installation .....	10
1.2	First Steps .....	11
1.3	The example application OS_StartLEDBlink.c .....	12
1.4	Stepping through the sample application .....	13
2	Build your own application .....	16
2.1	Introduction .....	17
2.2	Required files for an embOS .....	17
2.3	Change library mode .....	17
2.4	Select another CPU .....	17
3	Libraries .....	18
3.1	Naming conventions for prebuilt libraries .....	19
4	CPU and compiler specifics .....	20
4.1	Thread-safe system libraries .....	21
4.2	Thread-Local Storage TLS .....	21
5	Stacks .....	23
5.1	Task stack .....	24
5.2	System stack .....	24
5.3	Interrupt stack .....	24
5.4	Stack specifics .....	24
6	Interrupts .....	26
6.1	What happens when an interrupt occurs? .....	27
6.2	Defining interrupt handlers in C .....	27
6.3	Interrupt handling without vectored interrupt controller .....	27
6.4	Interrupt handling with vectored interrupt controller .....	29
6.5	Interrupt-stack switching .....	37
6.6	Fast Interrupt (FIQ) .....	38
7	MMU and cache support .....	39
7.1	MMU and cache support with embOS .....	40
7.2	MMU and cache handling for ARM CPUs .....	41
7.3	MMU and cache handling program sample .....	56

8	VFP and NEON support .....	57
8.1	Vector Floating Point and NEON support .....	58
9	embOS Thread Script .....	60
9.1	Introduction .....	61
9.2	How to use it .....	61
10	Technical data .....	67
10.1	Memory requirements .....	68



# Chapter 1

## Using embOS

---

This chapter describes how to start with and use embOS. You should follow these steps to become familiar with embOS.

## 1.1 Installation

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.

Assuming that you are using an IDE to develop your application, no further installation steps are required. You will find a lot of 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

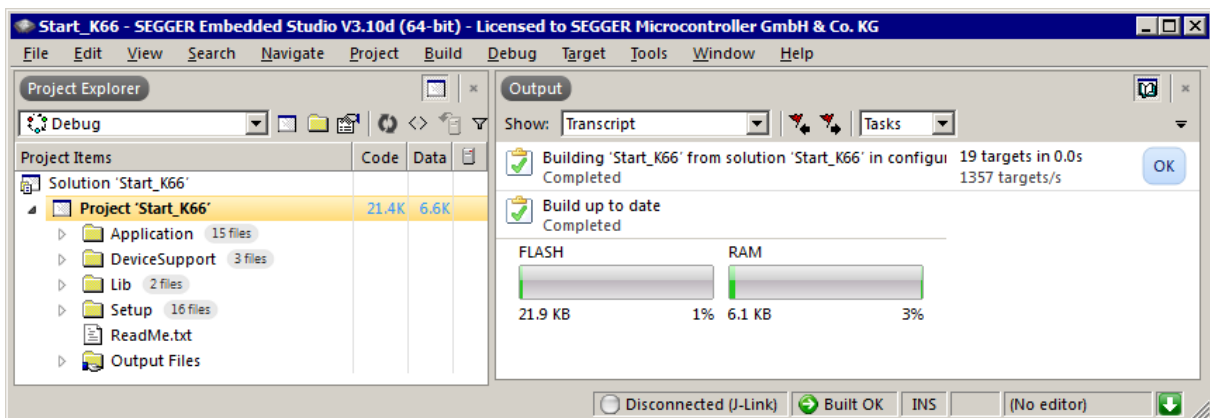
After installation of embOS you can create your first multitasking application. You have received several ready to go sample start workspaces and projects and every other files needed in the subfolder `Start`. It is a good idea to use one of them as a starting point for all of your applications. The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

```

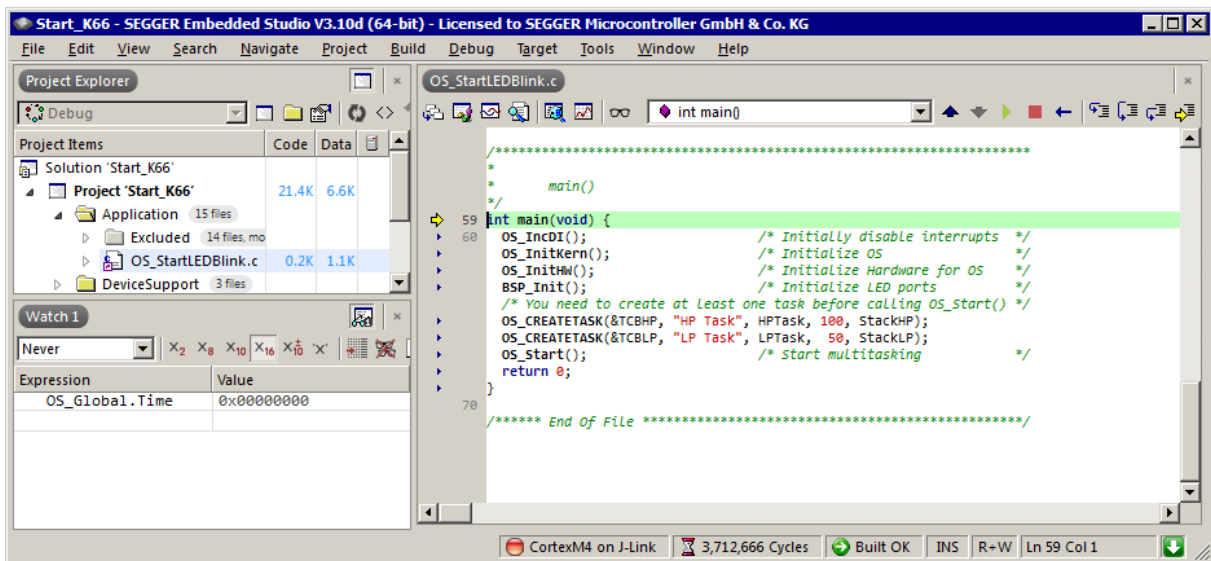
# 1.4 Stepping through the sample application

When starting the debugger, you will see the `main()` function (see example screen shot 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.

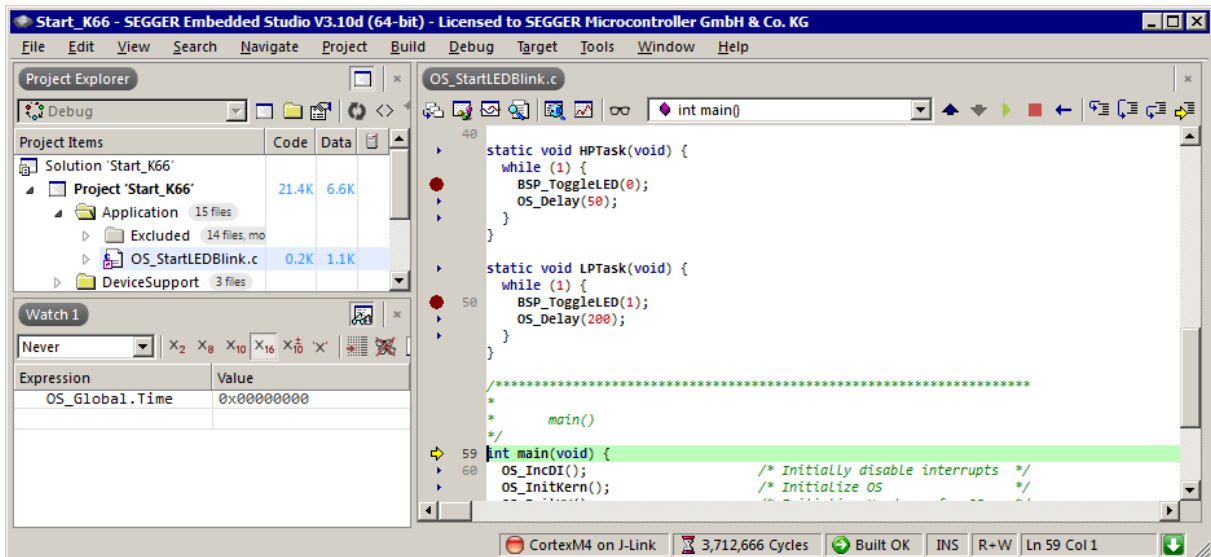
`OS_Init()` is part of the embOS library and written in assembler; you can there fore only step into it in disassembly mode. It initializes the relevant OS variables.

`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.

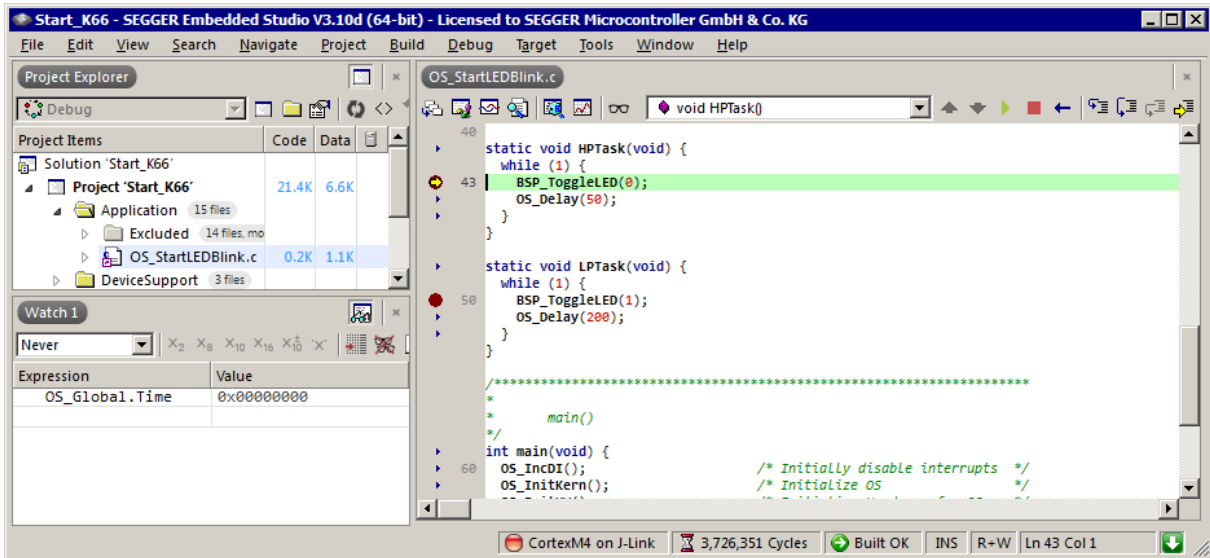


Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.

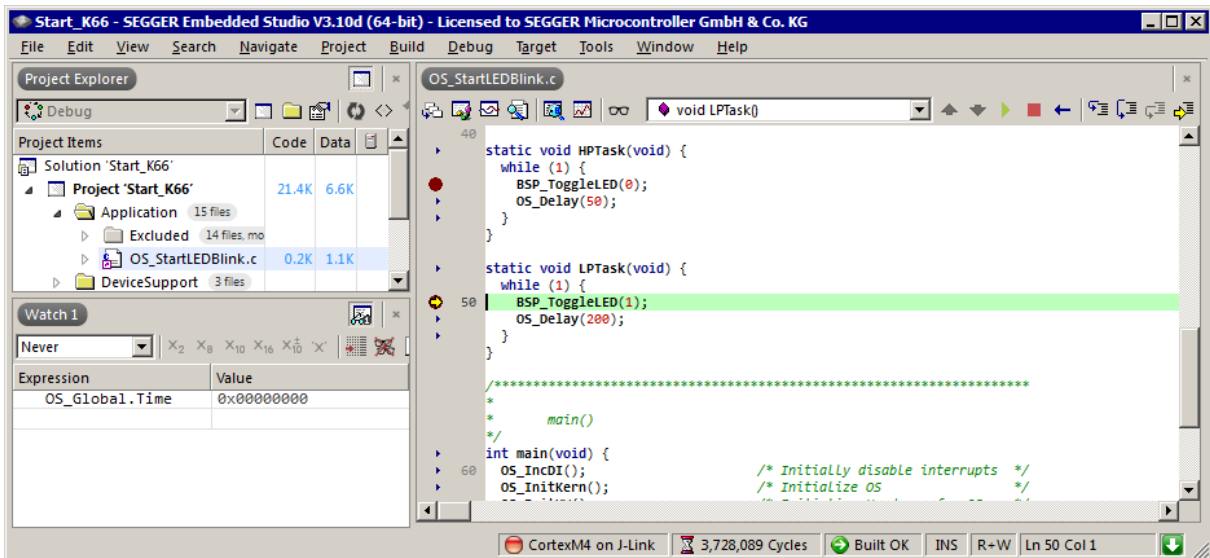


As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only.

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

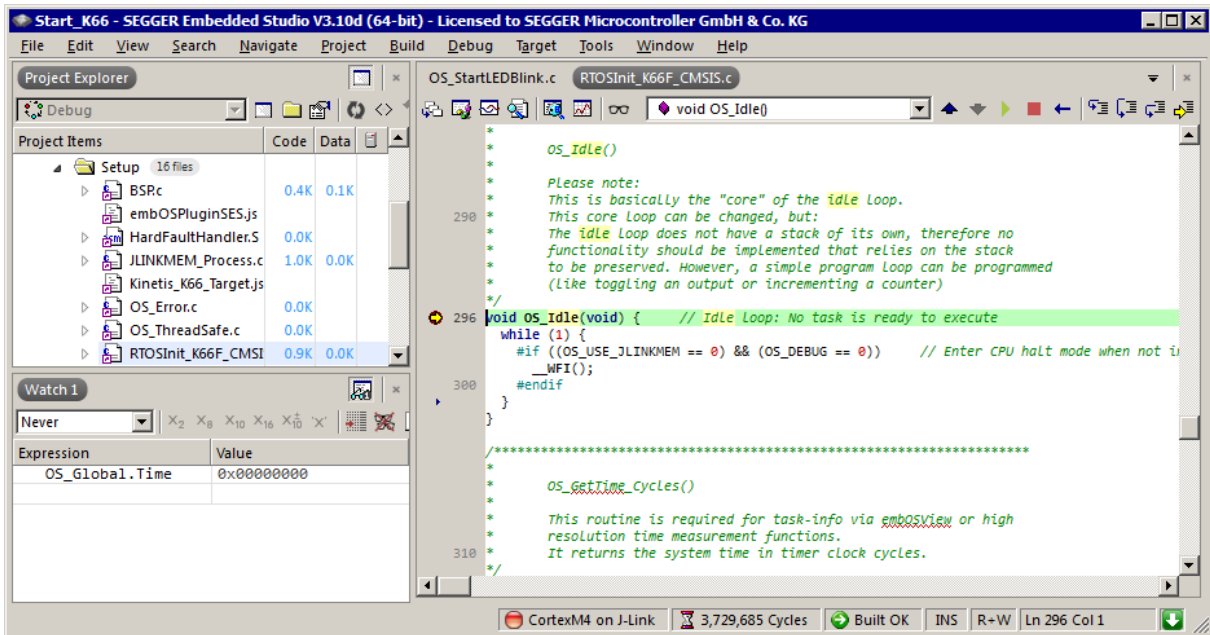


If you continue stepping, you will arrive at the task that has lower priority:



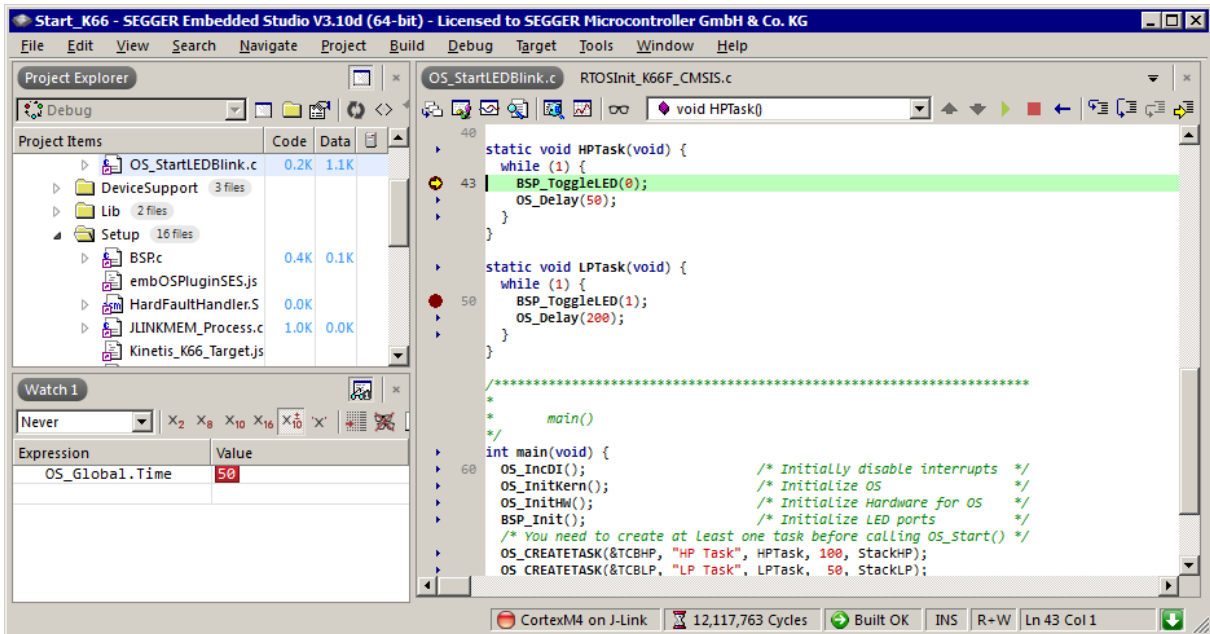
Continue to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

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 50 system tick delay.



# Chapter 2

## Build your own application

---

This chapter provides all information to set up your own embOS project.



## 2.1 Introduction

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 subfolder `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 `BoardSupport\<Manufacturer>\<MCU>` subfolder. It contains hardware-dependent initialization code for embOS. It initializes the system timer interrupt and optional communication for embOSView via UART or JTAG.
- `OS_Error.c` from one target specific subfolder `BoardSupport\<Manufacturer>\<MCU>`. The error handler is used if any debug library is used in your project.
- One embOS library from the subfolder `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 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 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 `BoardSupport\` folder. 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()`, `OS_COM_Init()`, the interrupt service routines for embOS system timer tick and communication to embOSView and the low level initialization.

# Chapter 3

## Libraries

---

This chapter includes CPU-specific information such as CPU-modes and available 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>_<CpuMode>_<Endianess>_<Interwork>_<Libmode>.a
```

Parameter	Meaning	Values
<a href="#">Architecture</a>	Specifies the ARM architecture	v4t : ARMv4 v5te: ARMv5 v7a : ARMv7A
<a href="#">CpuMode</a>	Specifies the CPU mode	a : ARM t : Thumb / Thumb2
<a href="#">Endianess</a>	Endianess	be : Big endian le : Little endian
<a href="#">Interwork</a>	Specifies if interworking is used	n : Non interwork i : Interwork
<a href="#">Libmode</a>	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_t_le_i_dp.a` is the library for an ARMv7A core, thumb2 mode, little endian mode, interwork, with debug and profiling support.

# Chapter 4

## CPU and compiler specifics

---

## 4.1 Thread-safe system libraries

embOS ARM for Segger Embedded Studio may be used with standard system libraries. embOS delivers the file `OS_ThreadSafe.c` which includes hook functions to make e.g. the heap management functions thread safe.

## 4.2 Thread-Local Storage TLS

The SEGGER Embedded Studio 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 SEGGER Embedded Studio is prepared to support the 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 every task. Only tasks that call functions using TLS need to activate the TLS by calling an initialization function when the task is started.

Library objects that need thread-local storage when used in multiple tasks are e.g.:

- 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\_TASK\_SetContextExtensionTLS()

#### Description

`OS_TASK_SetContextExtensionTLS()` may be called from a task to initialize and use Thread-local storage.

#### Prototype

```
void OS_TASK_SetContextExtensionTLS(void);
```

#### Additional information

`OS_TASK_SetContextExtensionTLS()` shall be the first function called from a task when TLS should be used in the specific task. The function must not be called multiple times from one task. The thread-local storage is allocated on the heap. To ensure thread safe heap management, the thread safe system library functionality shall also be enabled when using TLS.

Thread safe system library calls are automatically enabled when the source module `OS_ThreadSafe.c` which is delivered with embOS in the BSP Setup folders is included in the project.

#### 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_TASK_SetContextExtensionTLS();
}
```

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

static void LPTask(void) {
    OS_TASK_SetContextExtensionTLS();
    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

---

This chapter describes how embOS uses the different stacks of the ARM CPU.

## 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 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 the ARM7/ARM9 and Cortex-A/R cores, the minimum basic task stack size is about 68 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. We recommend at least 256 bytes stack as a start.

## 5.2 System stack

The embOS system executes in supervisor mode. The minimum system stack size required by embOS is about 136 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 stack size define in your linker file. We recommend a minimum stack size of 256 bytes for the `CSTACK`.

## 5.3 Interrupt stack

If a normal hardware exception occurs, the ARM core switches to IRQ mode, which has a separate stack pointer. To enable support for nested interrupts, execution of the ISR itself in a different CPU mode than IRQ mode is necessary. embOS switches to supervisor mode after saving scratch registers, `LR_irq` and `SPSR_irq` onto the IRQ stack.

As a result, only registers mentioned above are saved onto the IRQ stack. For the interrupt routine itself, the supervisor stack is used. The size of the interrupt stack can be changed by modifying the `irq_stack` define in your linker file. We recommend at least 128 bytes.

Every interrupt requires 28 bytes on the interrupt stack. The maximum interrupt stack size required by the application can be calculated as is "Maximum interrupt nesting level \* 28 bytes". For task switching from within an interrupt handler, it is required, that the end address of the interrupt stack is aligned to an 8 byte boundary. This alignment is forced during stack pointer initialization in the startup routine. Therefore, an additional margin of about 8 bytes should be added to the calculated maximum interrupt stack size. For standard applications, we recommend at least 92 to 128 bytes of IRQ stack.

## 5.4 Stack specifics

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

- `CSTACK` is the system stack.
- `IRQ_STACK` is the interrupt stack.

The `CSTACK` is used during startup, during `main()`, or embOS internal functions, and for C-level interrupt handler.

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 `CSTACK` as stack for further execution.

The startup code initializes the system stack pointer and the IRQ stack pointer. When the CPU starts, it runs in Supervisor mode.



The start up code switches to IRQ mode and sets the stack pointer to the stack which was defined as `IRQ_STACK`. The startup code switches to System mode and sets the stack pointer to the stack which was defined as `CSTACK`.

The `main()` function therefore is called in system mode and uses the `CSTACK`. When embOS is initialized, the supervisor stack pointer is initialized. The supervisor stack and system stack are the same, both stack pointers point into the `CSTACK`.

This is no problem, because the supervisor mode is not entered as long as `main()` is executed. All functions run in system mode. After embOS is started with `OS_Start()`, embOS internal functions run in Supervisor mode, as long as no task is running. The `CSTACK` may then be used as Supervisor stack, because it is not used anymore by other functions. Tasks run in system mode, but they do not use the "system" stack. Tasks have their own stack which is defined as some variable in any RAM location.

# 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 Interrupt stack.
- The CPU saves PC and flags in registers `LR_irq` and `SPSR_irq`.
- The CPU jumps to the vector address `0x18`, or offset `0x18` in the vector table, and continues execution from there.
- `embOS IRQ_Handler()`: save scratch registers.
- `embOS IRQ_Handler()`: save `LR_irq` and `SPSR_irq`.
- `embOS IRQ_Handler()`: switch to supervisor mode.
- `embOS IRQ_Handler()`: execute `OS_irq_handler()` (defined in `RTOSINIT*.C`).
- `embOS OS_irq_handler()`: check for interrupt source and execute timer interrupt, serial communication or user ISR.
- `embOS IRQ_Handler()`: switch to IRQ mode.
- `embOS IRQ_Handler()`: restore `LR_irq` and `SPSR_irq`.
- `embOS IRQ_Handler()`: pop scratch registers.
- Return from interrupt.

When using an ARM derivate with vectored interrupt controller, ensure that `IRQ_Handler()` is called from every interrupt. The interrupt vector itself may then be examined by the C-level interrupt handler in `RTOSInit*.c`.

## 6.2 Defining interrupt handlers in C

Interrupt handlers called from the `embOS` interrupt handler in `RTOSInit*.c` are just normal C-functions which do not take parameters and do not return any value. The default C interrupt handler `OS_irq_handler()` in `RTOSInit*.c` 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, the default interrupt handler `OS_irq_handler()` in `RTOSInit*.c` calls `OS_INT_Leave()` or `OS_INT_LeaveNestable()` and returns to the primary 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_irq_func(void) {
    if (__INTPND & 0x0800) { // Interrupt pending ?
        __INTPND = 0x0800; // reset pending condition
        OSTEST_X_ISR0();    // handle interrupt
    }
}
```

## 6.3 Interrupt handling without vectored interrupt controller

Standard ARM CPUs, without implementation of a vectored interrupt controller, always branch to address `0x18` when an interrupt occurs. The application is responsible to examine the interrupting source.

The reaction to an interrupt is as follows:

- `embOS IRQ_Handler()` is called.
- `IRQ_Handler()` saves registers and switches to supervisor mode.
- `IRQ_Handler()` calls `OS_irq_handler()`.

- OS\_irq\_handler() informs embOS that interrupt code is running by a call of OS\_INT\_Enter() and then calls OS\_USER\_irq\_func() which has to handle all interrupt sources of the application.
- OS\_irq\_handler() checks whether embOS timer interrupt has to be handled.
- OS\_irq\_handler() checks whether embOS UART interrupts for communication with embOSView have to be handled.
- OS\_irq\_handler() informs embOS that interrupt handling ended by a call of OS\_INT\_Leave() and returns to IRQ\_Handler().
- IRQ\_Handler() restores registers and performs a return from interrupt.

## Example

Simple interrupt routine:

```
void OS_USER_irq_func(void) {
    if (__INTPND & 0x0800) { // Interrupt pending ?
        __INTPND = 0x0800; // Reset pending condition
        OSTEST_X_ISR0(); // Handle interrupt
    }
    if (__INTPND & 0x0400) { // Interrupt pending ?
        __INTPND = 0x0400; // Reset pending condition
        OSTEST_X_ISR1(); // Handle interrupt
    }
}
```

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.

When using an ARM derivative with vectored interrupt controller, ensure that `IRQ_Handler()` is called from every interrupt. This is default when startup code and hardware initialization delivered with embOS is used. The interrupt vector itself will then be examined by the C-level interrupt handler `OS_irq_handler()` in `RTOSInit*.c`.

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:

- embOS interrupt handler `IRQ_Handler()` is called by CPU or interrupt controller.
- `IRQ_Handler()` saves registers and switches to supervisor mode.
- `IRQ_Handler()` calls `OS_irq_handler()` (in `RTOSInit*.c`).
- `OS_irq_handler()` examines the interrupting source by reading the interrupt vector from the interrupt controller.
- `OS_irq_handler()` informs the RTOS that interrupt code is running by a call of `OS_INT_EnterNestable()` which re-enables interrupts.
- `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()` calls `OS_INT_LeaveNestable()` which disables interrupts and informs embOS that interrupt handling has finished.
- `OS_irq_handler()` returns to `IRQ_Handler()`.
- `IRQ_Handler()` restores registers and performs a return from interrupt.

### 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
<code>OS_ARM_InstallISRHandler()</code>	Installs an interrupt handler
<code>OS_ARM_EnableISR()</code>	Enables a specific interrupt
<code>OS_ARM_DisableISR()</code>	Disables a specific interrupt
<code>OS_ARM_ISRSetPrio()</code>	Sets the priority of a specific interrupt
<code>OS_ARM_AssignISRSource()</code>	Assigns a hardware interrupt channel to an interrupt vector
<code>OS_ARM_EnableISRSource()</code>	Enables an interrupt channel of a VIC type interrupt controller
<code>OS_ARM_DisableISRSource()</code>	Disables an interrupt channel of a VIC type interrupt controller

## 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
<a href="#">ISRIndex</a>	Index of the interrupt source, normally the interrupt vector number.
<a href="#">pISRHandler</a>	Address of the interrupt handler function.

### Return Value

OS\_ISR\_HANDLER\*: The address of the previously installed interrupt function, which was installed at the addressed vector number before.

### 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
<code>ISRIndex</code>	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\_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.

## 6.4.6 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.

## 6.4.7 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.

### 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

```

## 6.5 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 primary 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 and cache support

---

This chapter describes the MMU and cache support for ARM CPUs.

## 7.1 MMU and cache support with embOS

embOS comes with functions to support the MMU and cache of ARMv4, ARMv5 and ARMv7A CPUs which allow 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.

The alignment may be forced by a `#pragma` or by the linker file. A translation table in RAM has to be set up during run time. embOS delivers API functions to set up this table. Assembly language programming is not required.



## 7.2 MMU and cache handling for ARM CPUs

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

Function	Description
<code>OS_ARM_MMU_InitTT()</code>	Initialize the MMU translation table.
<code>OS_ARM_MMU_AddTTEntries()</code>	Add address entries to the table.
<code>OS_ARM_MMU_Enable()</code>	Enable the MMU.
<code>OS_ARM_MMU_GetVirtualAddr()</code>	Translates a physical address into a virtual address
<code>OS_ARM_MMU_v2p()</code>	Translates a virtual address into a physical address.
<code>OS_ARM_ICACHE_Enable()</code>	Enable the instruction cache.
<code>OS_ARM_ICACHE_Invalidate()</code>	Invalidates the complete instruction cache.
<code>OS_ARM_DCACHE_Enable()</code>	Enable the data cache.
<code>OS_ARM_DCACHE_Invalidate()</code>	Invalidates the complete data cache.
<code>OS_ARM_DCACHE_Clean()</code>	Clean data cache.
<code>OS_ARM_DCACHE_CleanRange()</code>	Clean data cache range.
<code>OS_ARM_DCACHE_InvalidateRange()</code>	Invalidate the data cache.
<code>OS_ARM_CACHE_Sync()</code>	Syncs data and instruction cache.
<code>OS_ARM_AddL2Cache()</code>	Sets 2nd level cache API table.

## 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 zero, thus all entries are marked invalid initially.

### Prototype

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

### Parameters

Parameter	Description
<code>pTranslationTable</code>	Points to the base address of the translation table.

### Additional information

This function does not need to be called, if the translation table is located in ROM.

## 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 which should be used for the selected area. May be one of the following modes: ARMv4/ARMv5: 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_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
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

This function does not need to be called, if the translation table is located in ROM. The function adds entries for every section of one MegaByte size into the translation table for the specified memory area.

## 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
<code>pTranslationTable</code>	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 example an LCD controller accesses the display 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
<code>pVAddr</code>	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 addresses 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.

## 7.2.6 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.

## 7.2.7 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);
```



## 7.2.8 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.

## 7.2.9 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);
```

## 7.2.10 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:

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

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.

## 7.2.11 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
<code>p</code>	Points to the base address of the memory area that should be updated.
<code>NumBytes</code>	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:

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

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.

## 7.2.12 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. Invalidation 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
<code>p</code>	Points to the base address of the memory area that should be updated.
<code>NumBytes</code>	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:  $t = (\text{NumBytes} / \text{Cache line size}) * 10$  [CPU clock cycles]. Notes

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.

## 7.2.13 OS\_ARM\_CACHE\_Sync()

### Description

OS\_ARM\_CACHE\_Sync() cleans the data cache and invalidates the instruction cache 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.

## 7.2.14 OS\_ARM\_AddL2Cache()

### Description

OS\_ARM\_AddL2Cache() is used to add.

### Prototype

```
void OS_ARM_v7A_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 is 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 0x3FFF000u

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

## 7.3 MMU and cache handling program sample

The MMU and cache handling has to be set up before the data segments are initialized. 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
__low_level_init(void) {
    //
    // Init MMU and caches
    //
    OS_ARM_MMU_InitTT (&_TranslationTable[0]);
    //
    // Internal SRAM, the first MB remapped to 0, cacheable, bufferable
    //
    OS_ARM_MMU_AddTTEntries ( &_TranslationTable[0],
                             OS_ARM_CACHEMODE_C_B,
                             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.



# Chapter 8

## VFP and NEON support

---

## 8.1 Vector Floating Point and NEON support

Some ARM MCUs come with integrated vectored floation point unit VFP and NEON unit. When activating the VFP or NEON support in the project options, the compiler and linker will add efficient code which uses the VFP when floating point operations are used or NEON instrutrions where possible in the application.

With embOS, the VFP and NEON registers have to be saved and restored when task switches are performed. For efficiency reasons, embOS does not save and restore the VFP and NEON registers for every task automatically. The context switching time and stack load are therefore not affected when the VFP/NEON unit is not used or needed. Saving and restoring the VFP/NEON registers can be enabled for every task individually by extending the task context of the tasks, where VFP or NEON is used.

### 8.1.1 OS\_TASK\_SetContextExtensionVFP()

#### Description

`OS_TASK_SetContextExtensionVFP()` has to be called as first function in a task, when the VFP is used in the task and the VFP registers have to be added to the task context.

#### Prototype

```
void OS_TASK_SetContextExtensionVFP (void void);
```

#### Additional information

`OS_TASK_SetContextExtensionVFP()` extends the task context to save and restore the VFP registers during context switches.

Additional task context extension for a task by calling `OS_TASK_SetContextExtension()` is not allowed and will call the embOS error handler `OS_Error()` in debug builds of embOS. There is no need to extend the task context for every task. Only those tasks using the VFP for calculation have to be extended.

### 8.1.2 OS\_TASK\_SetContextExtensionNEON()

#### Description

`void OS_TASK_SetContextExtensionNEON()` has to be called as first function in a task, when the NEON unit is used in the task and the NEON registers have to be added to the task context.

#### Prototype

```
void OS_TASK_SetContextExtensionNEON (void void);
```

#### Additional information

`OS_TASK_SetContextExtensionNEON()` extends the task context to save and restore the NEON registers during context switches.

Additional task context extension for a task by calling `OS_TASK_SetContextExtension()` is not allowed and will call the embOS error handler `OS_Error()` in debug builds of embOS. There is no need to extend the task context for every task. Only those tasks using the NEON for calculation have to be extended.

## 8.1.3 Using VFP/NEON in interrupt service routines

Using the VFP/NEON in interrupt service routines requires additional functions to save and restore the VFP/NEON registers.

As the compiler might not add additional code to save and restore the VFP/NEON registers on entry and exit of interrupt service routines, it is the users responsibility to save the VFP/NEON registers on entry of an interrupt service routine when the VFP or NEON is used in the ISR.

embOS delivers functions to save and restore the VFP or NEON context in an interrupt service routine.

### 8.1.3.1 OS\_VFP\_Save() / OS\_NEON\_Save()

#### Description

`OS_VFP_Save()` / `OS_NEON_Save()` has to be called as first function in an interrupt service routine, when the VFP/NEON is used in the interrupt service routine. The function saves the VFP/NEON registers on the stack.

#### Prototype

```
void OS_VFP_Save (void void);
```

```
void OS_NEON_Save (void void);
```

#### Additional information

`OS_VFP_Save()` / `OS_NEON_Save()` declares a local variable which reserves space for all temporary floating point registers and stores the registers in the variable. After calling the `OS_VFP_Save()/OS_NEON_Save()` function, the interrupt service routine may use the VFP or NEON unit for calculation without destroying the saved content of the VFP/NEON registers.

To restore the registers, the ISR has to call `OS_VFP_Restore()/OS_NEON_Restore()` at the end.

The function may be used in any ISR regardless the priority. It is not restricted to low priority interrupt functions.

### 8.1.3.2 OS\_VFP\_Restore() / OS\_NEON\_Restore()

#### Description

`OS_VFP_Restore()` / `OS_NEON_Restore()` has to be called as last function in an interrupt service routine, when the VFP/NEON registers were saved by a call of `OS_VFP_Save()/OS_NEON_Save()` at the beginning of the ISR. The function restores the VFP/NEON registers from the stack.

#### Prototype

```
void OS_VFP_Restore (void void);
```

```
void OS_NEON_Restore (void void);
```

#### Additional information

`OS_VFP_Restore()` / `OS_NEON_Restore()` restores the temporary VFP registers which were saved by a previous call of `OS_VFP_Save() / OS_NEON_Restore()`. It has to be used together with `OS_VFP_Save() / OS_NEON_Restore()` and should be the last function called in the ISR.

# Chapter 9

## embOS Thread Script

---

This chapter describes how to use the thread script shipped with embOS for ES.

# 9.1 Introduction

A thread script is included with any start project shipped with embOS for ES. 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.

# 9.2 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:

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

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

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

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

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

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

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)

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

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

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

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.

### 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")

## 9.2.1 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

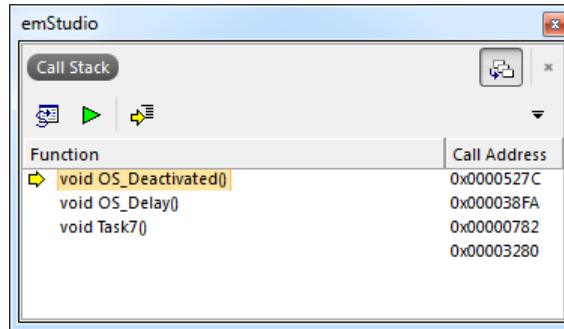
The task list displays various information about the running tasks:

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

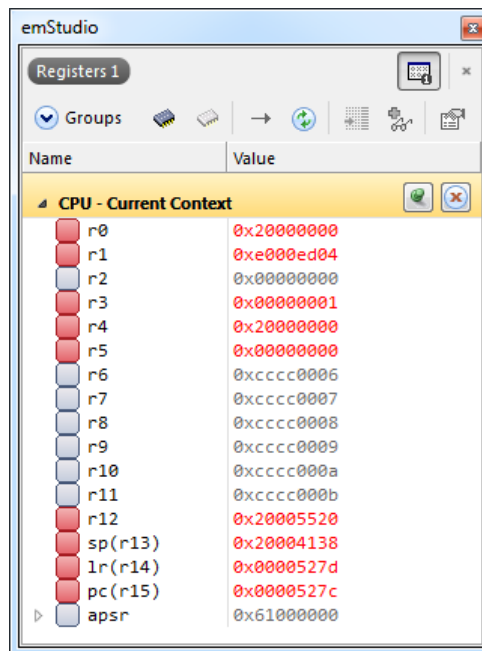
## 9.2.2 Task sensitivity

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 (`_DelayUntil()`) that has been called by this task:



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



## 9.2.3 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

## 9.2.4 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)

## 9.2.5 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)

## 9.2.6 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)



## 9.2.7 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)

## 9.2.8 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)

## 9.2.9 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)

## 9.2.10 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

## 9.2.11 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 10

## Technical data

---

This chapter lists technical data of embOS used with ARM CPUs.

## 10.1 Memory requirements

These values are neither precise nor guaranteed, but they give you a good idea of the memory requirements. They vary depending on the current version of embOS. The minimum ROM requirement for the kernel itself is about 1.700 bytes.

In the table below, which is for X-Release build, you can find minimum RAM size requirements for embOS resources. Note that the sizes depend on selected embOS library mode.

<b>embOS resource</b>	<b>RAM [bytes]</b>
Task control block	36
Software timer	20
Mutex	16
Semaphore	8
Mailbox	24
Queue	32
Task event	0
Event object	12