

# embOS

Real-Time  
Operating System

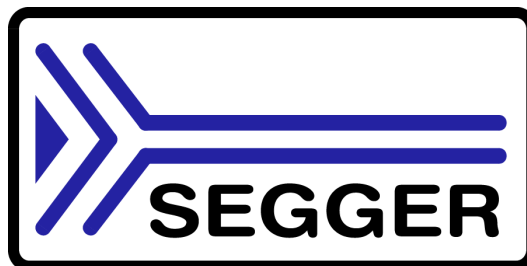
CPU & Compiler  
specifics for ARM core  
using GCC / emIDE

Document: UM01052

Software version 4.32

Revision: 0

Date: January 27, 2017



A product of SEGGER Microcontroller GmbH & Co. KG

[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 & Co. KG (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.

© 2007 - 2017 SEGGER Microcontroller GmbH & Co. KG, 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 & Co. KG

In den Weiden 11  
D-40721 Hilden

Germany

Tel. +49 2103-2878-0

Fax. +49 2103-2878-28

E-mail: [support@segger.com](mailto:support@segger.com)

Internet: <http://www.segger.com>

## Manual versions

This manual describes the current software version. If any error occurs, inform us and we will try to assist you as soon as possible.

Contact us for further information on topics or routines not yet specified.

Print date: January 27, 2017

Software	Revision	Date	By	Description
4,32	0	170125	TS	New Cache API description added.
4.22	0	160620	MC	New software version.
4.10b	0	150618	TS	New software version.
4.04a	1	150217	TS	Chapter "Naming conventions for prebuilt libraries" updated.
4.04a	0	150123	TS	New software version
3.90a	0	140430	AW	Update to embOS version 3.90a
3.90	0	140306	AW	Update to embOS version 3.90
3.88h	0	131223	MC	Initial version.



# About this document

---

## Assumptions

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

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

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

## How to use this manual

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

## Typographic conventions for syntax

This manual uses the following typographic conventions:

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

**Table 1.1: Typographic conventions**



**SEGGER Microcontroller GmbH & Co. KG** develops and distributes software development tools and ANSI C software components (middleware) for embedded systems in several industries such as telecom, medical technology, consumer electronics, automotive industry and industrial automation.

SEGGER's intention is to cut software development time for embedded applications by offering compact flexible and easy to use middleware, allowing developers to concentrate on their application.

Our most popular products are emWin, a universal graphic software package for embedded applications, and embOS, a small yet efficient real-time kernel. emWin, written entirely in ANSI C, can easily be used on any CPU and most any display. It is complemented by the available PC tools: Bitmap Converter, Font Converter, Simulator and Viewer. embOS supports most 8/16/32-bit CPUs. Its small memory footprint makes it suitable for single-chip applications.

Apart from its main focus on software tools, SEGGER develops and produces programming tools for flash micro controllers, as well as J-Link, a JTAG emulator to assist in development, debugging and production, which has rapidly become the industry standard for debug access to ARM cores.

**Corporate Office:**

<http://www.segger.com>

**United States Office:**

<http://www.segger-us.com>

## EMBEDDED SOFTWARE (Middleware)



**emWin**

**Graphics software and GUI**

emWin is designed to provide an efficient, processor- and display controller-independent graphical user interface (GUI) for any application that operates with a graphical display.



**embOS**

**Real Time Operating System**

embOS is an RTOS designed to offer the benefits of a complete multitasking system for hard real time applications with minimal resources.



**embOS/IP**

**TCP/IP stack**

embOS/IP a high-performance TCP/IP stack that has been optimized for speed, versatility and a small memory footprint.



**emFile**

**File system**

emFile is an embedded file system with FAT12, FAT16 and FAT32 support. Various Device drivers, e.g. for NAND and NOR flashes, SD/MMC and Compact-Flash cards, are available.



**USB-Stack**

**USB device/host stack**

A USB stack designed to work on any embedded system with a USB controller. Bulk communication and most standard device classes are supported.

## SEGGER TOOLS

**Flasher**

**Flash programmer**

Flash Programming tool primarily for micro controllers.

**J-Link**

**JTAG emulator for ARM cores**

USB driven JTAG interface for ARM cores.

**J-Trace**

**JTAG emulator with trace**

USB driven JTAG interface for ARM cores with Trace memory. supporting the ARM ETM (Embedded Trace Macrocell).

**J-Link / J-Trace Related Software**

Add-on software to be used with SEGGER's industry standard JTAG emulator, this includes flash programming software and flash breakpoints.



# 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 .....	17
2.1	Introduction.....	18
2.2	Required files for an embOS.....	18
2.3	Change library mode.....	18
2.4	Select another CPU .....	18
3	Libraries .....	21
3.1	Naming conventions for prebuilt libraries .....	22
4	CPU and compiler specifics .....	23
4.1	Standard system libraries .....	24
4.2	Reentrancy, thread local storage.....	24
4.3	Reentrancy, thread safe heap management.....	27
5	Stacks .....	29
5.1	Task stack .....	30
5.2	System stack .....	30
5.3	Interrupt stack.....	30
5.4	Stack specifics .....	30
6	Interrupts.....	31
6.1	What happens when an interrupt occurs .....	32
6.2	What happens when an interrupt occurs?.....	33
6.3	Defining interrupt handlers in C.....	34
6.4	Interrupt handling without vectored interrupt controller .....	35
6.5	Interrupt handling with vectored interrupt controller.....	36
6.6	Interrupt stack switching .....	44
6.7	Fast Interrupt (FIQ) .....	45
7	MMU and cache support.....	47
7.1	MMU and cache support with embOS.....	48
7.2	MMU and cache handling for ARM CPUs .....	49
7.3	MMU and cache handling for ARM720 CPUs .....	63
7.4	MMU and cache handling program sample.....	72
8	VFP and NEON support.....	73
8.1	Vector Floating Point and NEON support .....	74
9	Technical data.....	77
9.1	Memory requirements .....	78





# 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*.

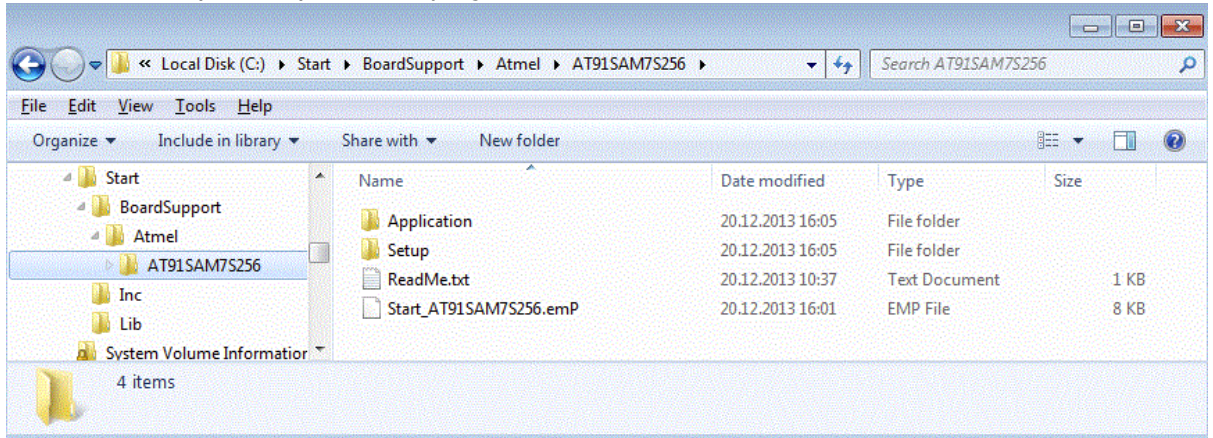
You should do this even if you do not intend to use the IDE for your application development to become familiar with embOS.

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.

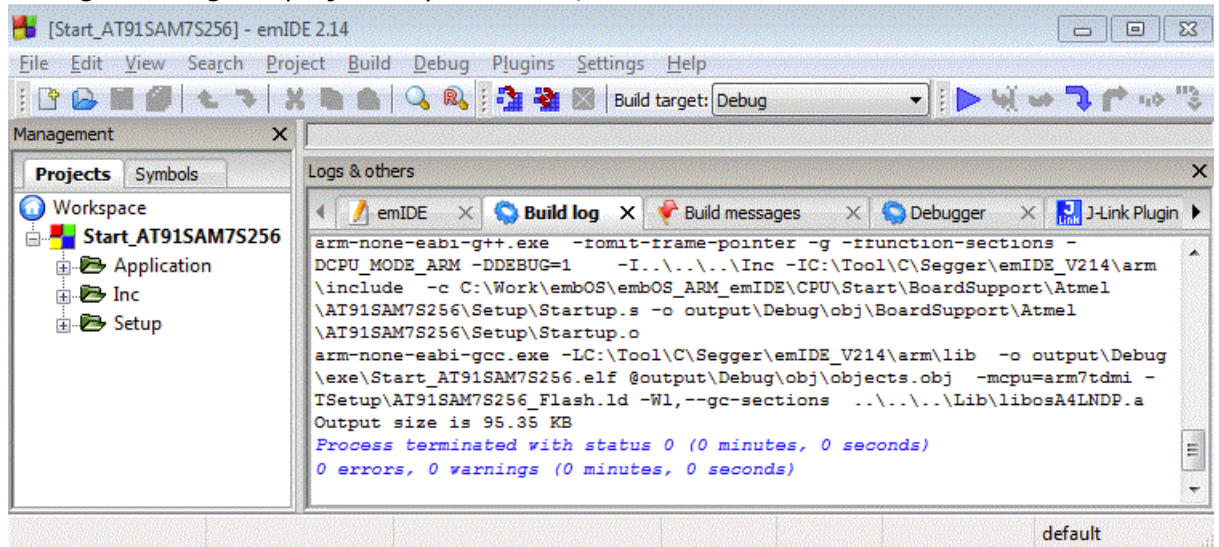
To start with, you may use the project for AT91SAM7S256 CPU:



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                           *
*****
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).
----- END-OF-HEADER -----
*/

#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_Delay (50);
    }
}

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

/*****
*
*           main()
*/
int main(void) {
    OS_InitKern(); /* Initialize OS */
    OS_InitHW(); /* Initialize Hardware for OS */
    BSP_Init(); /* Initialize LED ports */
    /* You need to create at least one task before calling OS_Start() */
    OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
    OS_CREATETASK(&TCBLP, "LP Task", LPTask, 50, StackLP);
    OS_Start(); /* Start multitasking */
    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_IncDI()` initially disables interrupts.

`OS_InitKern()` 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. Because of the previous call of `OS_IncDI()`, interrupts are not enabled during execution of `OS_InitKern()`.

`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.

```

BoardSupport\Atmel\AT91SAM7S256\Application\Start_2Tasks.c [Start_AT91SAM7S256] - emIDE 2.14
File Edit View Search Project Build Debug Plugins Settings Help
Build target: Debug
Management
Projects Symbols
Workspace
Start_AT91SAM7S256
Application
Excluded
Start_2Tasks.c
Inc
Setup
58 int main(void) {
59     OS_IncDI();           /* Initially disable
60     OS_InitKern();       /* Initialize OS
61     OS_InitHW();        /* Initialize Hardwar
62     /* You need to create at least one task here !
63     OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP)
64     OS_CREATETASK(&TCBLP, "LP Task", LPTask, 50, StackLP)
65     OS_Start();         /* Start multitasking
66     return 0;
67 }
68
C:\Work\embOS\embC WINDOWS-1252 Line 39, Column 1 Insert Read only default

```

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

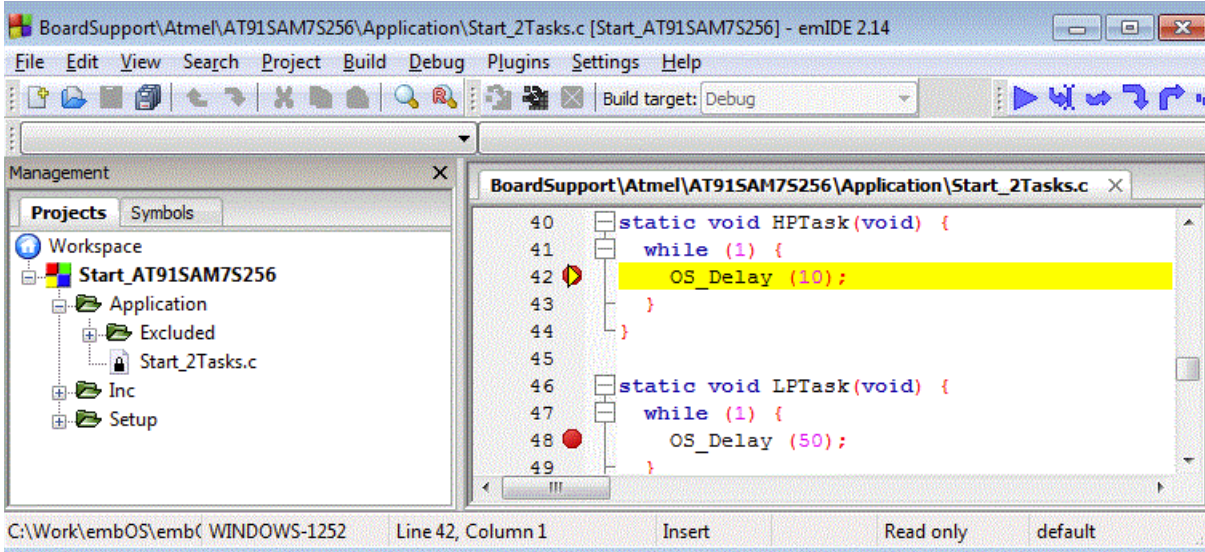
```

BoardSupport\Atmel\AT91SAM7S256\Application\Start_2Tasks.c [Start_AT91SAM7S256] - emIDE 2.14
File Edit View Search Project Build Debug Plugins Settings Help
Build target: Debug
Management
Projects Symbols
Workspace
Start_AT91SAM7S256
Application
Excluded
Start_2Tasks.c
Inc
Setup
40 static void HPTask(void) {
41     while (1) {
42         OS_Delay (10);
43     }
44 }
45
46 static void LPTask(void) {
47     while (1) {
48         OS_Delay (50);
49     }
50 }
C:\Work\embOS\embC WINDOWS-1252 Line 48, Column 2 Insert Read only default

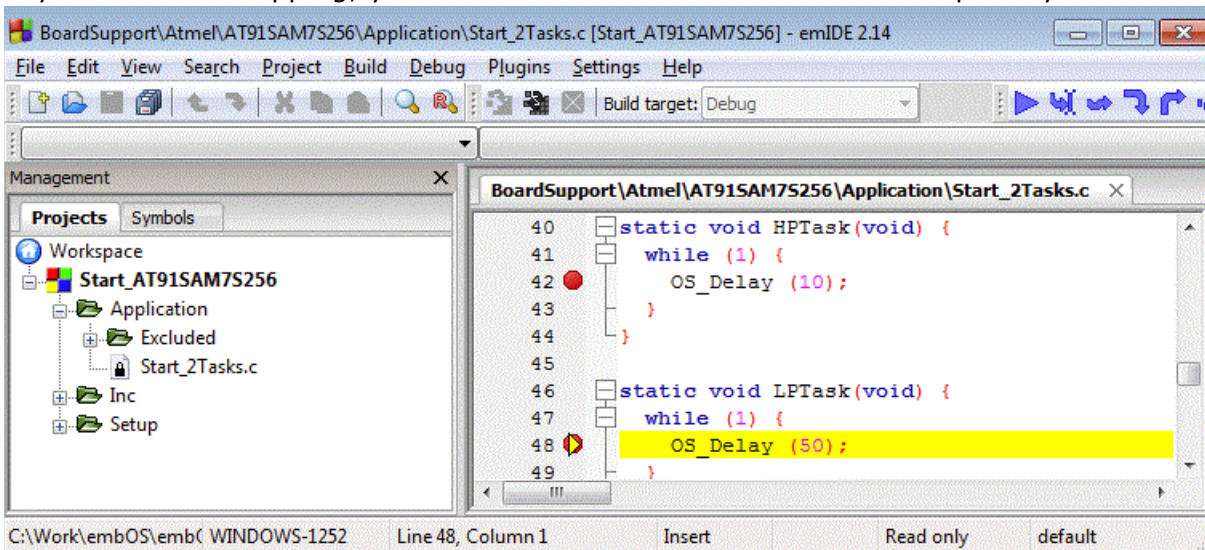
```

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

Click **Debug / Continue**, step over `OS_Start()`, or step into `OS_Start()` in disassembly mode until you reach the highest priority task.



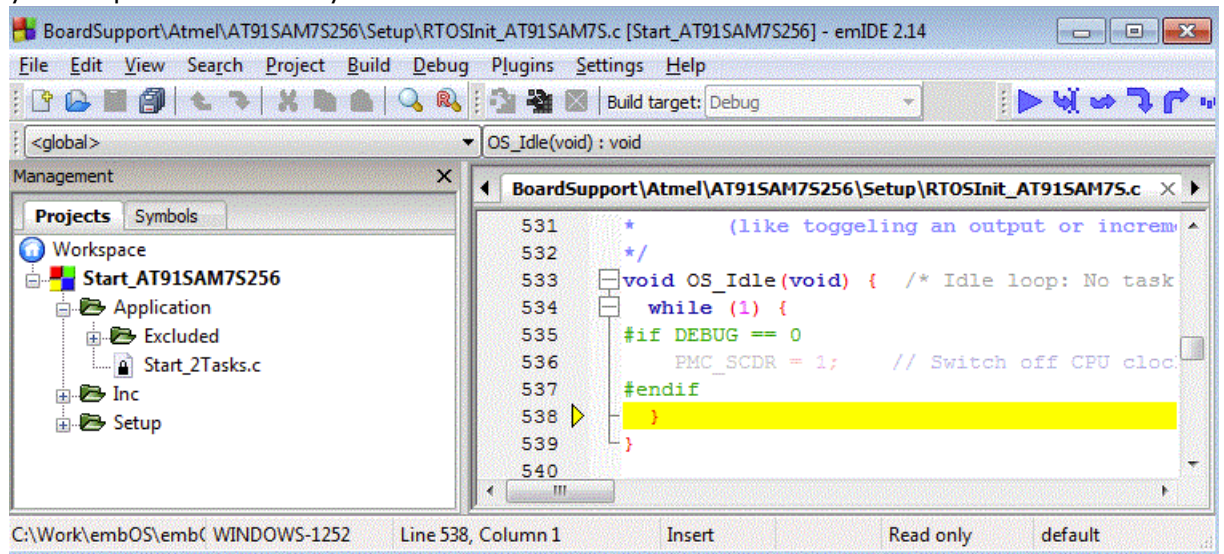
If you continue stepping, you will arrive in the task that has lower priority:



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).

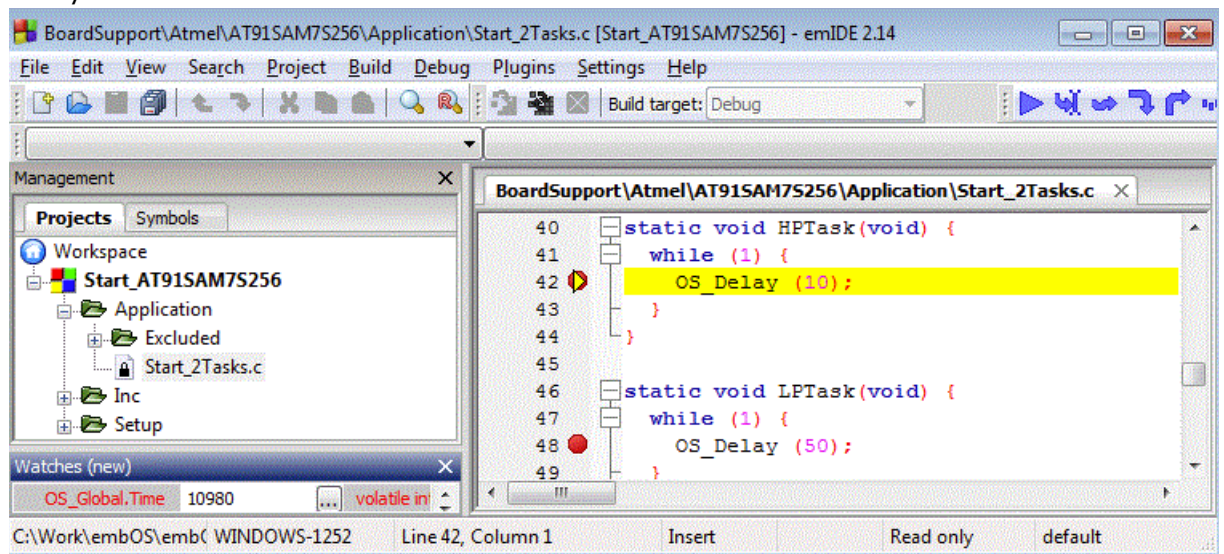


You will arrive there when you step into the `OS_Delay()` function in disassembly mode. `OS_Idle()` is part of `RTOSInit*.c`. You may also set a breakpoint there before you step over the delay in `LPTask`.



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* 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.
- One embOS library from the subfolder **Lib\**.
- `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.
- 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 a call of `OS_InitKern()` and `OS_InitHW()` prior any other embOS functions are called.

You should then modify or replace the `OS_Start_LEDBlink.c` source file in the subfolder **Application\**.

## 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 describes the available embOS 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<isa><mode><endianess><interwork><libmode>.a
```

Parameter	Meaning	Values
mode	Specifies the CPU mode.	A: ARM
		T: Thumb / Thumb2
isa	Specifies the instruction set architecture.	4: v4T
		5: v5T
		7: v7A
endianess	Specifies target endianess.	B: Big
		L: Little
interwork	Specifies if interwork option is enabled.	I: Enable interworking (not supported)
		N: Do not use interworking
libmode	Specifies the library mode	XR: Extreme Release
		R: Release
		S: Stack check
		D: Debug
		SP: Stack check + profiling
		DP: Debug + profiling + Stack check
		DT: Debug + profiling + Stack check + trace

#### Example:

libosA4LNDP.a is the library for an ARMv4 core, ARM mode, little endian mode, no interwork, with debug and profiling support.

# Chapter 4

## CPU and compiler specifics

---

## 4.1 Standard system libraries

**embOS** for ARM cores and emIDE may be used with standard GNU system libraries for most of all projects without any modification.

Heap management and file operation functions of standard system libraries are not reentrant and require a special initialization or additional modules when used with **embOS**, if non thread safe functions are used from different tasks.

Alternatively, for heap management, **embOS** delivers its own thread safe functions which may be used. These functions are described in the **embOS** generic manual.

## 4.2 Reentrancy, thread local storage

The GCC newlib supports usage of thread-local storage located in a `_reent` structure as local variable for every task.

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 GNU 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 or specific tasks.

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 defining a local variable and calling an initialization function when the task is started.

The `_reent` structure is stored on the task stack and have to be considered when the task stack size is defined. The structure may contain up to 800 bytes.

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

- 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\_ExtendTaskContext\_TLS()

### Description

OS\_ExtendTaskContext\_TLS() may be called from a task which needs thread local storage to initialize and use Thread-local storage.

### Prototype

```
void OS_ExtendTaskContext_TLS(struct _reent * pReentStruct)
```

### Parameter

pReentStruct is a pointer to the thread local storage. It is the address of the variable of type struct \_reent which holds the thread local data.

### Return value

None.

### Additional Information

OS\_ExtendTaskContext\_TLS() 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 has to be defined as local variable in the task.

### Example

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

    OS_ExtendTaskContext_TLS(&TaskReentStruct);*/
    while (1) {
        ... /* Task functionality */
    }
}
```

Please ensure sufficient task stack to hold the \_reent structure variable.

For details on the \_reent structure, \_impure\_ptr, and library functions which require precautions on reentrance, refer to the GNU documentation.

## 4.2.2 OS\_ExtendTaskContext\_TLS\_VFP()

### Description

OS\_ExtendTaskContext\_TLS\_VFP() has to be called as first function in a task, when thread-local storage and thread safe floatingpoint processor support is needed in the task.

### Prototype

```
void OS_ExtendTaskContext_TLS_VFP(struct _reent * pReentStruct)
```

### Parameter

pReentStruct is a pointer to the thread local storage. It is the address of the variable of type struct \_reent which holds the thread local data.

### Return value

None.

### Additional Information

OS\_ExtendTaskContext\_TLS\_VFP() shall be the first function called from a task when TLS and VFP should be used in the specific task.

The function must not be called multiple times from one task.

The thread-local storage should be defined as local variable in the task.

The task specific TLS management is generated as embOS task extension together with the storage needed for the VFP registers. The VFP registers are automatically saved onto the task stack when the task is suspended, and restored, when the task is resumed. Additional task extension by a call of OS\_ExtendTaskContext() is impossible.

## 4.3 Reentrancy, thread safe heap management

The heap management functions in the system libraries are not thread-safe without implementation of additional locking functions.

The GCC library calls two hook functions to lock and unlock the mutual access of the heap-management functions.

The empty locking functions from the system library may be overwritten by the application to implement a locking mechanism.

A locking is required when multiple tasks access the heap, or when objects are created dynamically on the heap by multiple tasks.

The locking functions are implemented in the source module `OS_MallocLock.c` which is included in the "Setup" subfolder in every embOS start project.

If thread safe heap management is required, the module has to be compiled and linked with the application.

### 4.3.1 `__malloc_lock()`, lock the heap against mutual access

`__malloc_lock()` is the locking function which is called by the system library whenever the heap management has to be locked against mutual access.

The implementation delivered with embOS claims a resource semaphore.

### 4.3.2 `__malloc_unlock()`

`__malloc_unlock()` is the counterpart to `__malloc_lock()`.

It is called by the system library whenever the heap management locking can be released. The implementation delivered with embOS releases the resource semaphore.

None of these functions has to be called directly by the application. They are called from the system library functions when required.

The functions are delivered in source form to allow replacement of the dummy functions in the system library.



# 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.

The minimum basic task stack size is about 44 bytes. Because any function call uses some amount of stack, the task stack size has to be large enough. 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 140 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 128 bytes of IRQ stack.

## 5.4 Stack specifics

Interrupts require space on the supervisor and interrupt stack. The interrupt stack is used to store contents of scratch registers, the ISR itself uses supervisor stack. The Supervisor stack is also used during startup, `main()`, embOS internal functions and software timers.

All other stacks are not initialized and not used by embOS. If required by the application, the startup function and linker command files have to be modified to initialize the stacks.

# Chapter 6

## Interrupts

---

## 6.1 What happens when an interrupt occurs



## 6.2 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`.

## 6.3 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_EnterInterrupt()` or `OS_EnterNestableInterrupt()` 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_LeaveInterrupt()` or `OS_LeaveNestableInterrupt()` 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

#### Example of a "simple" interrupt-routine

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

## 6.4 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_EnterInterrupt()` 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_LeaveInterrupt()` and returns to `IRQ_Handler()`.

`IRQ_Handler()` restores registers and performs a return from interrupt.

### Example

Simple `OS_USER_irq_func()` 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_ISR0(); // Handle interrupt
    }
}
```

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

## 6.5 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_EnterNestableInterrupt()` 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_LeaveNestableInterrupt()` 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

**Table 6.1: Interrupt handler functions for ARM derivatives with built in vectored interrupt controller**

## 6.5.1 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);
```

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

**Table 6.2: OS\_ARM\_InstallISRHandler() parameter list**

### Return value

OS\_ISR\_HANDLER\*: the address of the previous 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.

## 6.5.2 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)
```

Parameter	Description
ISRIndex	Index of the interrupt source which should be enabled.

**Table 6.3: OS\_ARM\_EnableISR() parameter list**

### 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.

## 6.5.3 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);
```

Parameter	Description
<a href="#">ISRIndex</a>	Index of the interrupt source which should be disabled.

**Table 6.4: OS\_ARM\_DisableISR() parameter list**

### 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.

## 6.5.4 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);
```

Parameter	Description
<a href="#">ISRIndex</a>	Index of the interrupt source which should be modified.
<a href="#">Prio</a>	The priority which should be set for the specific interrupt.

**Table 6.5: OS\_ARM\_ISRSetPrio() parameter list**

### 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.



## 6.5.5 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

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

Parameter	Description
<a href="#">ISRIndex</a>	Index of the interrupt vector which should be modified.
<a href="#">Source</a>	The source channel number which should be assigned to the specified interrupt vector.

**Table 6.6: OS\_ARM\_AssignISRSource() parameter list**

### Additional information

This function assigns a hardware interrupt line to an interrupt vector of VIC type only. It can not be used for other types of vectored interrupt controller. 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.

## 6.5.6 OS\_ARM\_EnableISRSource

### Description

OS\_ARM\_EnableISRSource() is used to enable an interrupt input channel of an interrupt controller of VIC type.

### Prototype

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

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

**Table 6.7: OS\_ARM\_EnableISRSource() parameter list**

### Additional information

This function enables a hardware interrupt input of a VIC type interrupt controller. It can not be used for other types of vectored interrupt controller. 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.

## 6.5.7 OS\_ARM\_DisableISRSource

### Description

OS\_ARM\_DisableISRSource() is used to Disable an interrupt input channel of an interrupt controller of VIC type.

### Prototype

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

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

**Table 6.8: OS\_ARM\_DisableISRSource() parameter list**

### Additional information

This function disables a hardware interrupt input of a VIC type interrupt controller. It can not be used for other types of vectored interrupt controller. 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 of a function usage

```
/* 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
OS_ARM_EnableISRSource(UART_INT_SOURCE);              // Enable UART interrupt source
```

## 6.6 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_EnterIntStack()` and `OS_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.

## 6.7 Fast Interrupt (FIQ)

The FIQ interrupt cannot be used with embOS functions, it is reserved for high speed user functions which are also called zero latency interrupts.

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_CleanRange()</code>	Clean data cache.
<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.

**Table 7.1: MMU and cache handling for ARM CPUs**

## 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 );
```

Parameter	Description
<a href="#">pTranslationTable</a>	Points to the base address of the translation table.

**Table 7.2: OS\_ARM\_MMU\_InitTT() parameter list**

### 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 );
```

Parameter	Description
<a href="#">pTranslationTable</a>	Points to the base address of the translation table.
<a href="#">CacheMode</a>	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_NC_NB - Strongly ordered OS_ARM_CACHEMODE_C_NB - Write-Through cacheable OS_ARM_CACHEMODE_NC_B - Shareable device OS_ARM_CACHEMODE_C_B - Write-Back cacheable
<a href="#">VIndex</a>	Virtual address index, which is the start address of the virtual memory address range with MBytes resolution. $VIndex = (\text{virtual address} \gg 20)$
<a href="#">PIndex</a>	Physical address index, which is the start address of the physical memory area range with MBytes resolution. $PIndex = (\text{physical address} \gg 20)$
<a href="#">NumEntries</a>	Specifies the size of the memory area in MBytes.

**Table 7.3: OS\_ARM\_MMU\_AddTTEntries() parameter list**

### 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 );
```

Parameter	Description
<a href="#">pTranslationTable</a>	Points to the base address of the translation table.

**Table 7.4: OS\_ARM\_MMU\_Enable() parameter list**

### 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  CacheMode );
```

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: OS_ARM_CACHEMODE_NC_NB OS_ARM_CACHEMODE_C_NB OS_ARM_CACHEMODE_NC_B OS_ARM_CACHEMODE_C_B OS_ARM_CACHEMODE_ANY

**Table 7.5: OS\_ARM\_MMU\_GetVirtualAddr() parameter list**

### Return value:

void\* which is 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 );
```

Parameter	Description
<a href="#">pVAddr</a>	Pointer which represents the virtual address.

**Table 7.6: OS\_ARM\_MMU\_v2p() parameter list**

### 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.

## 7.2.6 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\_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 NumBytes );
```

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.

**Table 7.7: OS\_ARM\_DCACHE\_CleanRange() parameter list**

### 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.

### Notes

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.

## 7.2.10 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 );
```

## 7.2.11 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 );
```

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.

**Table 7.8: OS\_ARM\_DCACHE\_InvalidateRange() parameter list**

### 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:

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

### 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.

## 7.2.12 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.

## 7.2.13 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 );
```

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

**Table 7.9: OS\_ARM\_AddL2Cache() parameter list**

### 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 0x3FFFF00u

//
// 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 for ARM720 CPUs

ARM720 CPUs with MMU have a unified cache for data and instructions. embOS delivers the following functions to setup and handle the MMU and cache.

Function	Description
<code>OS_ARM720_MMU_InitTT()</code>	Initialize the MMU translation table.
<code>OS_ARM720_MMU_AddTTEntries()</code>	Add address entries to the table.
<code>OS_ARM720_MMU_Enable()</code>	Enable the MMU.
<code>OS_ARM720_MMU_GetVirtualAddr()</code>	Translates a physical address into a virtual address
<code>OS_ARM720_MMU_v2p()</code>	Translates a virtual address into a physical address.
<code>OS_ARM720_CACHE_Enable()</code>	Enable the cache.
<code>OS_ARM720_CACHE_CleanRange()</code>	Clean the cache.
<code>OS_ARM720_CACHE_InvalidateRange()</code>	Invalidate the cache.

**Table 7.10: MMU and cache handling for ARM720 CPUs**

## 7.3.1 OS\_ARM720\_MMU\_InitTT()

### Description

OS\_ARM720\_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_ARM720_MMU_InitTT( unsigned int * pTranslationTable );
```

Parameter	Description
<a href="#">pTranslationTable</a>	Points to the base address of the translation table.

**Table 7.11: OS\_ARM720\_MMU\_InitTT() parameter list**

### Additional Information

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



## 7.3.2 OS\_ARM720\_MMU\_AddTTEntries()

### Description

OS\_ARM\_MMU720\_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 );
```

Parameter	Description
<a href="#">pTranslationTable</a>	Points to the base address of the translation table.
<a href="#">CacheMode</a>	Specifies the cache operating mode which should be used for the selected area. May be one of the following modes: OS_ARM_CACHEMODE_NC_NB OS_ARM_CACHEMODE_C_NB OS_ARM_CACHEMODE_NC_B OS_ARM_CACHEMODE_C_B
<a href="#">VIndex</a>	Virtual address index, which is the start address of the virtual memory address range with MBytes resolution. $VIndex = (\text{virtual address} \gg 20)$
<a href="#">PIndex</a>	Physical address index, which is the start address of the physical memory area range with MBytes resolution. $PIndex = (\text{physical address} \gg 20)$
<a href="#">NumEntries</a>	Specifies the size of the memory area in MBytes.

**Table 7.12: OS\_ARM\_MMU\_AddTTEntries() parameter list**

### 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.

### 7.3.3 OS\_ARM720\_MMU\_Enable()

#### Description

OS\_ARM720\_MMU\_Enable() is used to enable the MMU which will then perform the address mapping.

#### Prototype

```
void OS_ARM720_MMU_Enable ( unsigned int * pTranslationTable );
```

Parameter	Description
<a href="#">pTranslationTable</a>	Points to the base address of the translation table.

**Table 7.13: OS\_ARM720\_MMU\_Enable() parameter list**

#### Additional Information

As soon as the function was called, the address translation is active. The MMU table has to be setup before calling OS\_ARM720\_MMU\_Enable().

## 7.3.4 OS\_ARM720\_MMU\_GetVirtualAddr()

### Description

OS\_ARM720\_MMU\_GetVirtualAddr() is used to translate a physical address into a virtual address with specified cache mode.

### Prototype

```
void * OS_ARM720_MMU_GetVirtualAddr ( unsigned long PAddr,
                                     unsigned int  CacheMode );
```

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: OS_ARM_CACHEMODE_NC_NB OS_ARM_CACHEMODE_C_NB OS_ARM_CACHEMODE_NC_B OS_ARM_CACHEMODE_C_B OS_ARM_CACHEMODE_ANY

**Table 7.14: OS\_ARM720\_MMU\_GetVirtualAddr() parameter list**

### Return value:

void\* which is 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\_ARM720\_MMU\_GetVirtualAddress() can be used to find the address for uncached access.

The MMU table has to be setup before the function is called.

## 7.3.5 OS\_ARM720\_MMU\_v2p()

### Description

OS\_ARM720\_MMU\_v2p() is used to translate a virtual address into a physical address.

### Prototype

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

Parameter	Description
<a href="#">pVAddr</a>	Pointer which represents the virtual address.

**Table 7.15: OS\_ARM720\_MMU\_v2p() parameter list**

### 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\_ARM720\_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.

## 7.3.6 OS\_ARM720\_CACHE\_Enable()

### Description

OS\_ARM720\_CACHE\_Enable() is used to enable the instruction cache of the CPU.

### Prototype

```
void OS_ARM720_CACHE_Enable ( void );
```

### Additional Information

As soon as the function was called, the unified cache is active. The MMU has to be set up and has to be enabled before the cache is enabled.

## 7.3.7 OS\_ARM720\_CACHE\_CleanRange()

### Description

OS\_ARM720\_CACHE\_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_ARM720_CACHE_CleanRange ( void *      p,
                                   unsigned int NumBytes );
```

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.

**Table 7.16: OS\_ARM720\_CACHE\_CleanRange() parameter list**

### 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\_ARM720\_CACHE\_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. As each cache line covers 32 bytes, the total time to invalidate a range may be calculated as:

$$t = (\text{NumBytes} / 32) * (10 \text{ [CPU clock cycles]} + \text{Memory write time}).$$

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 32 bytes to be cleaned.

## 7.3.8 OS\_ARM720\_CACHE\_InvalidateRange()

### Description

OS\_ARM720\_CACHE\_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_ARM720_CACHE_InvalidateRange ( void *      p,
                                       unsigned int NumBytes );
```

Parameter	Description
<a href="#">SourceIndex</a>	Index of the interrupt channel which should be disabled.

**Table 7.17: OS\_ARM720\_CACHE\_InvalidateRange() parameter list**

### 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. 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 32 byte boundary and the number of bytes to be invalidated has to be a multiple of 32 bytes.

The debug version of embOS will call OS\_Error() with error code OS\_ERR\_NON\_ALIGNED\_INVALIDATE, if one of these restrictions is violated.

The cache is invalidated line by line. Invalidating one cache line takes approximately 10 CPU cycles. As each cache line covers 32 bytes, the total time to invalidate a range may be calculated as:

$$t = (\text{NumBytes} / 32) * 10 \text{ [CPU clock cycles].}$$

## 7.4 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 floating 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 instructions 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.

by extending the task context of the tasks, where VFP or NEON is used.

### 8.1.1 OS\_ExtendTaskContext\_VFP()

#### Description

`OS_ExtendTaskContext_VFP()` 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_ExtendTaskContext_VFP(void)
```

#### Return value

None.

#### Additional Information

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

Additional task context extension for a task by calling `OS_ExtendTaskContext()` 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\_ExtendTaskContext\_NEON()

#### Description

`OS_ExtendTaskContext_NEON()` 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_ExtendTaskContext_NEON(void)
```

#### Return value

None.

#### Additional Information

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

Additional task context extension for a task by calling `OS_ExtendTaskContext()` 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 OS_NEON_Save(void)
```

##### Return value

None.

##### 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 OS_NEON_Restore(void)
```

##### Return value

None.

##### 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_Save()`.

It has to be used together with `OS_VFP_Save()` / `OS_NEON_Save()` and should be the last function called in the ISR.



# Chapter 9

## Technical data

---

## 9.1 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 kernel itself has a minimum ROM size requirement of about 2.500 bytes.

In the table below, which is for XRelease 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
Resource semaphore	16
Counting semaphore	8
Mailbox	24
Queue	36
Task event	0
Event object	8

# Index

## Symbols

\_\_malloc\_lock() ..... 27  
 \_\_malloc\_unlock() ..... 27

## C

Cache ..... 48  
 CPU modes ..... 21

## H

Heap management ..... 27

## I

Installation ..... 10  
 Interrupt stack ..... 33, 44  
 interrupts ..... 33

## M

Memory requirements ..... 78  
 MMU ..... 48

## O

OS\_ARM\_AssignISRSource() ..... 41  
 OS\_ARM\_DCACHE\_CleanRange() ..... 58  
 OS\_ARM\_DCACHE\_Enable() ..... 57  
 OS\_ARM\_DCACHE\_InvalidateRange() ..... 60  
 OS\_ARM\_DisableISR() ..... 39  
 OS\_ARM\_DisableISRSource() ..... 43  
 OS\_ARM\_EnableISR() ..... 38  
 OS\_ARM\_EnableISRSource() ..... 42  
 OS\_ARM\_ICACHE\_Enable() ..... 55–56, 59  
 OS\_ARM\_InstallISRHandler() ..... 37  
 OS\_ARM\_ISRSetPrio() ..... 40  
 OS\_ARM\_MMU\_AddTTEntries() ..... 51  
 OS\_ARM\_MMU\_Enable() ..... 52  
 OS\_ARM\_MMU\_GetVirtualAddr() ..... 53  
 OS\_ARM\_MMU\_v2p() ..... 54  
 OS\_ARM720\_CACHE\_CleanRange() ..... 70  
 OS\_ARM720\_CACHE\_Enable() ..... 69  
 OS\_ARM720\_CACHE\_InvalidateRange() .. 71  
 OS\_ARM720\_MMU\_AddTTEntries() ..... 65  
 OS\_ARM720\_MMU\_Enable() ..... 66

OS\_ARM720\_MMU\_GetVirtualAddr() ..... 67  
 OS\_ARM720\_MMU\_InitTT() ..... 64  
 OS\_ARM720\_MMU\_v2p() ..... 68  
 OS\_ExtendTaskContext\_TLS() ..... 25  
 OS\_ExtendTaskContext\_TLS\_VFP() ..... 26  
 OS\_ExtendTaskContext\_VFP() ..... 74  
 OS\_irq\_handler() ..... 34  
 OS\_MallocLock.c ..... 27  
 OS\_USER\_irq\_func() ..... 35  
 OS\_VFP\_Restore() ..... 75  
 OS\_VFP\_Save() ..... 75

## R

Reentrancy ..... 24

## S

Stacks ..... 29  
 Stepping through the sample application 13  
 Syntax, conventions used ..... 5  
 System libraries ..... 24

## V

Vector Floating Point (VFP) ..... 74  
 Vector Floating Point support ..... 27  
 VFP ..... 74

