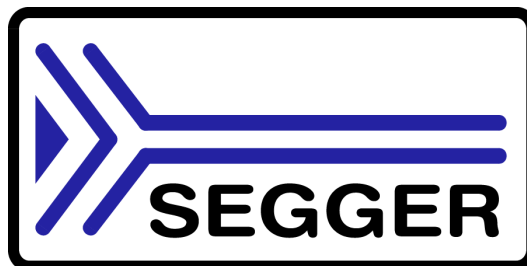


# **embOS**

Real-Time  
Operating System

CPU & Compiler  
specifics for ARM core  
using Keil MDK

Document: UM01005  
Software version 4.04a  
Revision: 0  
Date: December 2, 2014



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.

© 2010 - 2014 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: December 2, 2014

Software	Revision	Date	By	Description
4.04a	0	141202	TS	Update to new software version.
4.04	0	141120	TS	Update to new software version.
4.02a	0	141106	TS	Update to new software version.
3.86e	0	120608	TS	Update to new software version.
3.62c	0	090901	AS	First FrameMaker 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 with KEIL uVision.....	9
1.1	Installation .....	10
1.2	First steps .....	11
1.3	The sample application Start_LEDblink.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 for ARM.....	18
2.3	Change library mode.....	18
2.4	Select another CPU .....	18
3	ARM specifics .....	21
3.1	CPU modes.....	22
3.2	Available libraries .....	23
4	Compiler specifics.....	25
4.1	Standard system libraries .....	26
5	Stacks .....	27
5.1	Task stack for ARM .....	28
5.2	System stack for ARM .....	29
5.3	Interrupt stack for ARM .....	30
5.4	Stack specifics of the ARM family .....	31
6	Heap .....	33
6.1	Heap management .....	34
7	Interrupts.....	35
7.1	What happens when an interrupt occurs?.....	36
7.2	Defining interrupt handlers in C.....	37
7.3	Interrupt handling without vectored interrupt controller.....	38
7.4	Interrupt handling with vectored interrupt controller.....	39
7.5	Interrupt-stack switching.....	47
7.6	Fast Interrupt (FIQ) .....	48
8	MMU and cache support.....	49
8.1	MMU and cache support with embOS.....	50
8.2	MMU and cache handling for ARMv5 (ARM9 CPUs).....	51
8.3	MMU and cache handling program sample.....	61
9	STOP / WAIT Mode .....	63
9.1	Introduction.....	64
10	Technical data.....	65
10.1	Memory requirements .....	66

11 Files shipped with embOS .....67



# Chapter 1

## Using embOS with KEIL uVision

---

The following chapter describes how to start with and use embOS for ARM and KEIL Realview MDK compiler. You should follow these steps to become familiar with embOS for ARM and KEIL uVision.

## 1.1 Installation

**embOS** is shipped on CD-ROM or as a zip-file in electronic form.

In order to install it, proceed as follows:

If you received a CD, copy the entire contents to your hard-drive into any folder of your choice. When copying, keep all files in their respective sub directories. Make sure the files are not read only after copying.

If you received a zip-file, extract it to any folder of your choice, preserving the directory structure of the zip-file.

Assuming that you are using **KEIL uVision** project manager to develop your application, no further installation steps are required. You will find several prepared sample start projects, which you should use and modify to write your application. So follow the instructions of the next chapter 'First steps'.

You should do this even if you do not intend to use the project manager for your application development in order to become familiar with **embOS**.

If for some reason you will not work with the project manager, you should:

Copy either all or only the library-file that you need to your work-directory. This has the advantage that when you switch to an updated version of **embOS** later in a project, you do not affect older projects that use **embOS** also.

**embOS** does in no way rely on **KEIL uVision** project manager, it may be used without the project manager using batch files or a make utility without any problem.

## 1.2 First steps

After installation of **embOS** you are able to create your first multitasking application. You received several ready to go sample start projects and it is a good idea to use this as a starting point of all your applications.

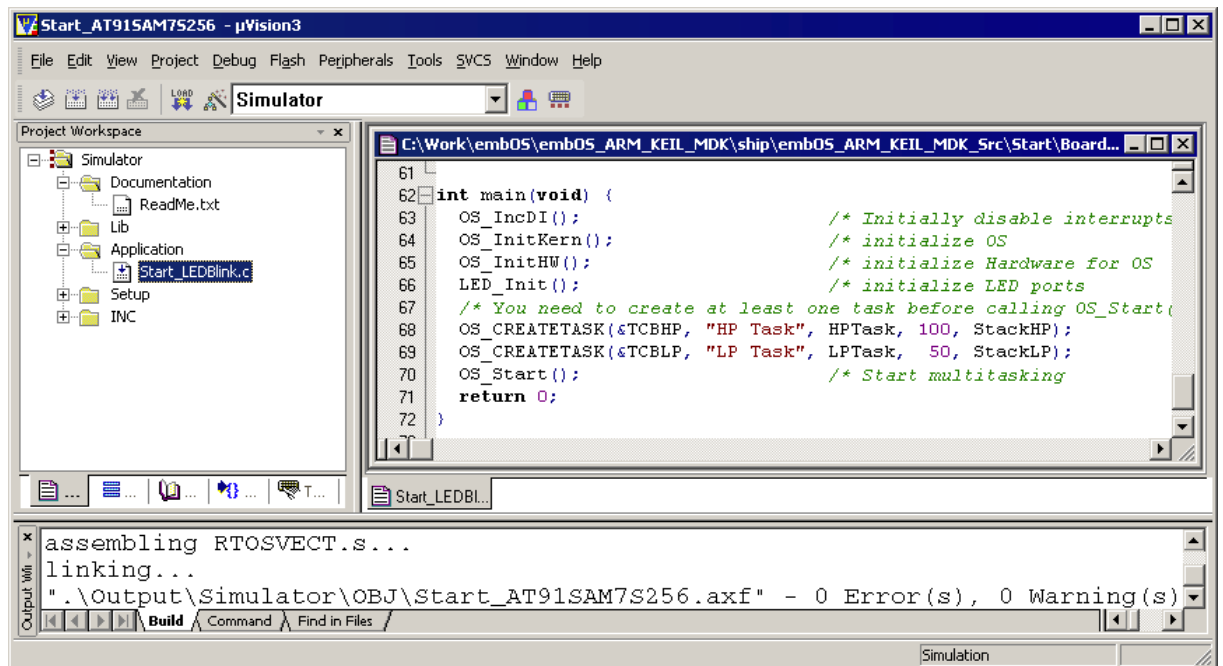
Samples start projects for various starter boards from different CPU manufacturers are found under the Start\Boards subfolder in the manufacturer and board specific subfolders.

Every additional files needed for the specific CPU are located in the subfolder "Setup" in the board specific folder.

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 project from start\Boards\manufacturer\ with the uVision project manager (e.g. by double clicking it).
- The following sample session was taken with a project for an ATMEL AT91SAM7S256 which can be used with an ATMEL AT91SAM7S-EK eval board and is located at Start\Boards\ATMEL\AT91SAM7S-EK
- For first steps, select the target for the simulator.
- Build the start project

Your screen should look like follows:



For latest information you should open the file ReadMe.txt.

## 1.3 The sample application Start\_LEDblink.c

The following is a printout of the sample application main.c. It is a good starting-point for your application. (Please 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 SYSTEME GmbH
 *      Solutions for real time microcontroller applications
 *****/

-----
File      : Start_LEDblink.c
Purpose   : Sample program for OS running on EVAL-boards with LEDs
----- END-OF-HEADER -----*/

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

OS_STACKPTR int StackHP[128], StackLP[128];          /* Task stacks */
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_IncDI();          /* Initially disable interrupts */
    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;
}

```

## 1.4 Stepping through the sample application

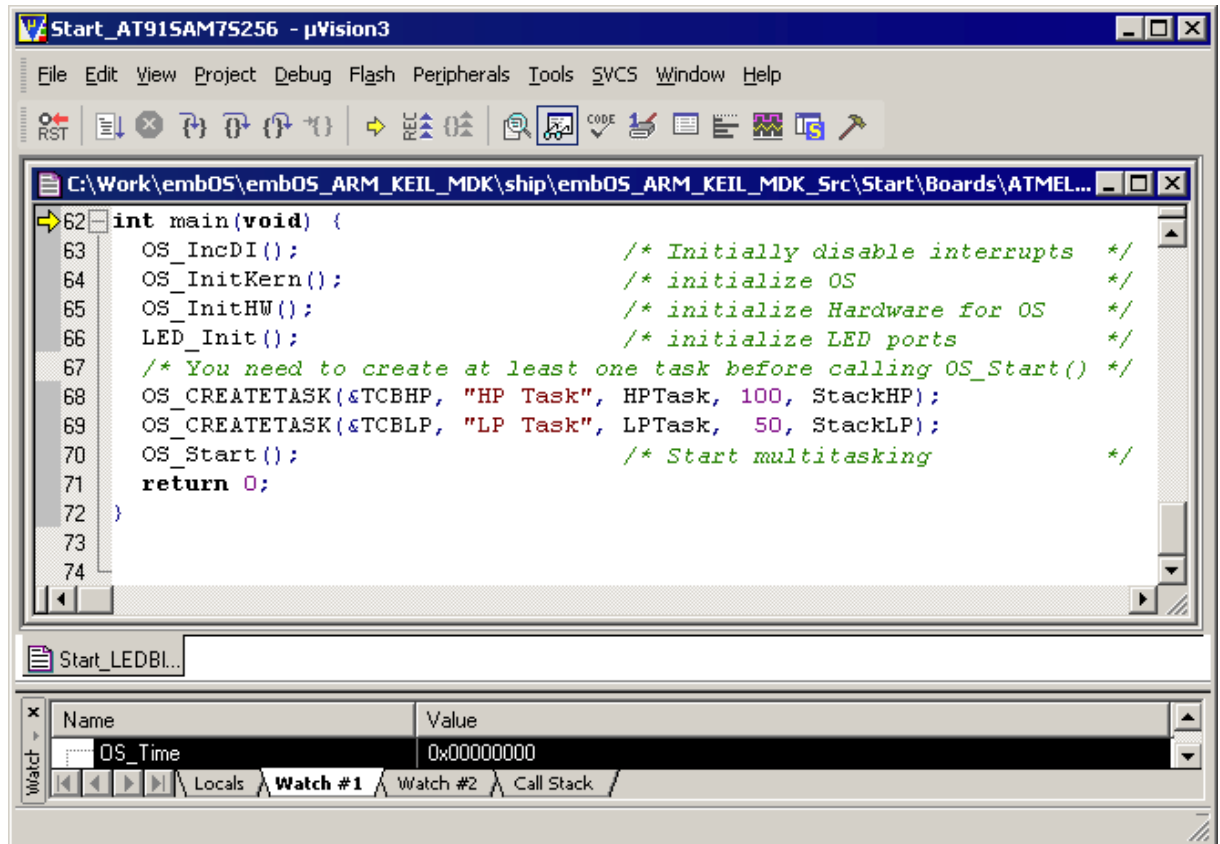
When starting the debugger, you will usually see the main function (very similar to the screenshot below). In some debuggers, you may look at the startup code and have to set a breakpoint at main. Now you can step through the program.

`OS_IncDI()` initially disables interrupts.

`OS_InitKern()` is part of the **embOS** library; 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 timer-tick-interrupt for **embOS**. Step through it to see what is done.

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



The screenshot shows the Keil uVision3 IDE with the following code in the main function:

```

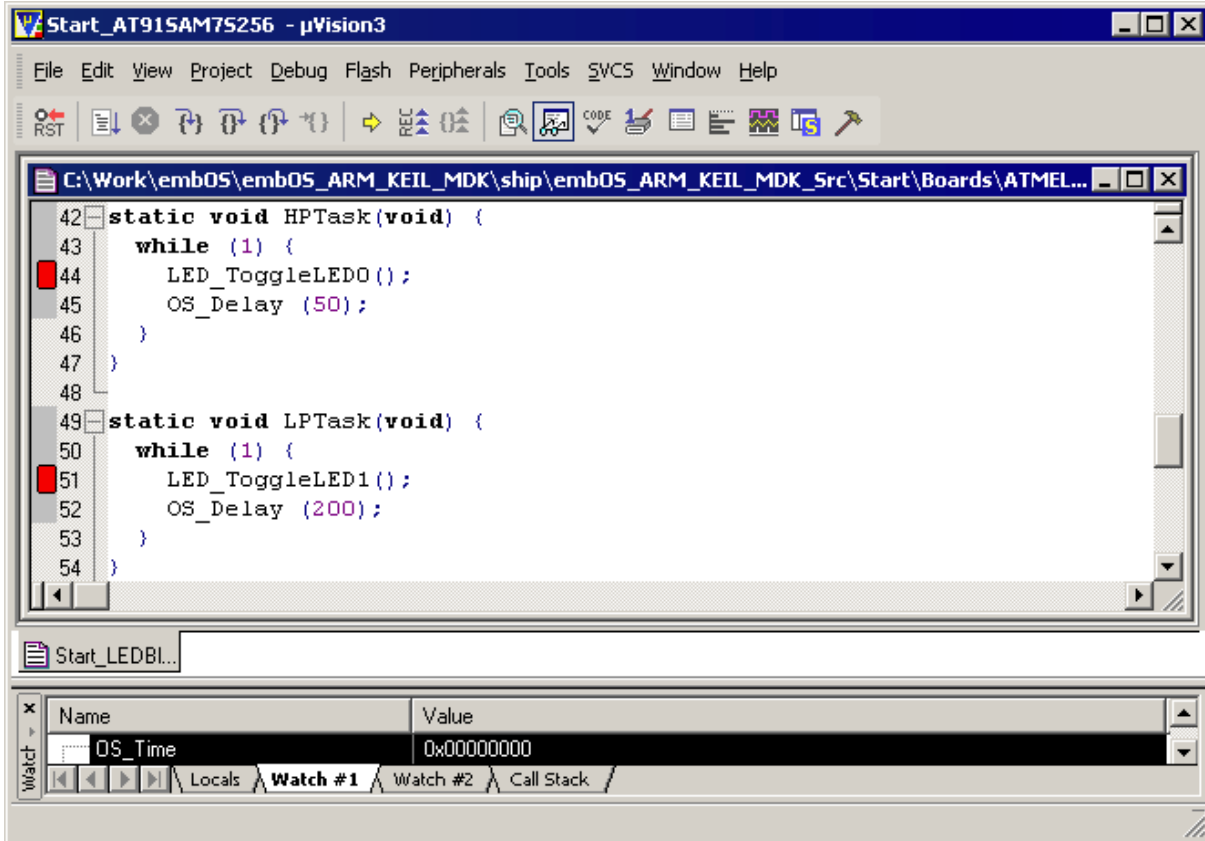
62 int main(void) {
63     OS_IncDI();           /* Initially disable interrupts */
64     OS_InitKern();       /* initialize OS */
65     OS_InitHW();        /* initialize Hardware for OS */
66     LED_Init();         /* initialize LED ports */
67     /* You need to create at least one task before calling OS_Start() */
68     OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
69     OS_CREATETASK(&TCBLP, "LP Task", LPTask, 50, StackLP);
70     OS_Start();         /* Start multitasking */
71     return 0;
72 }
73
74

```

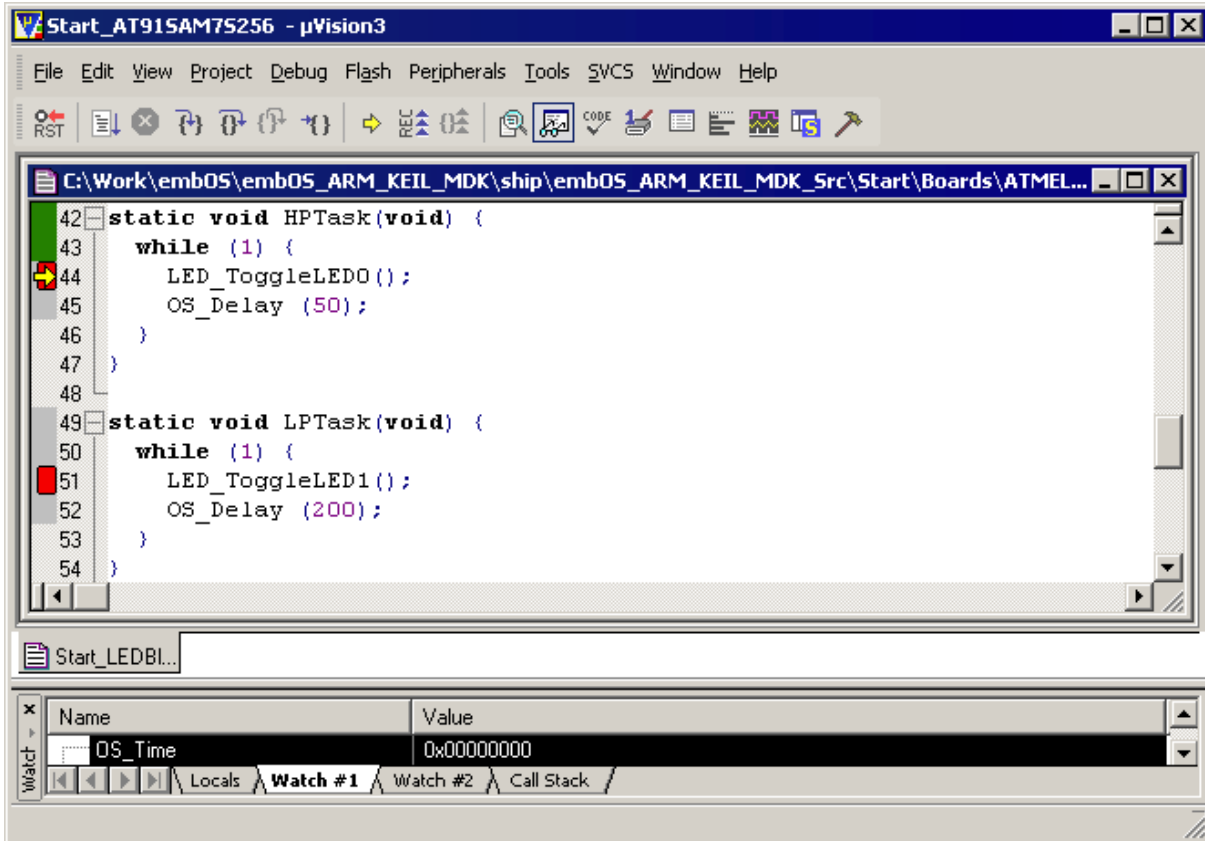
Below the code editor, the Watch window is visible, showing the value of `OS_Time` as `0x00000000`.

Name	Value
OS_Time	0x00000000

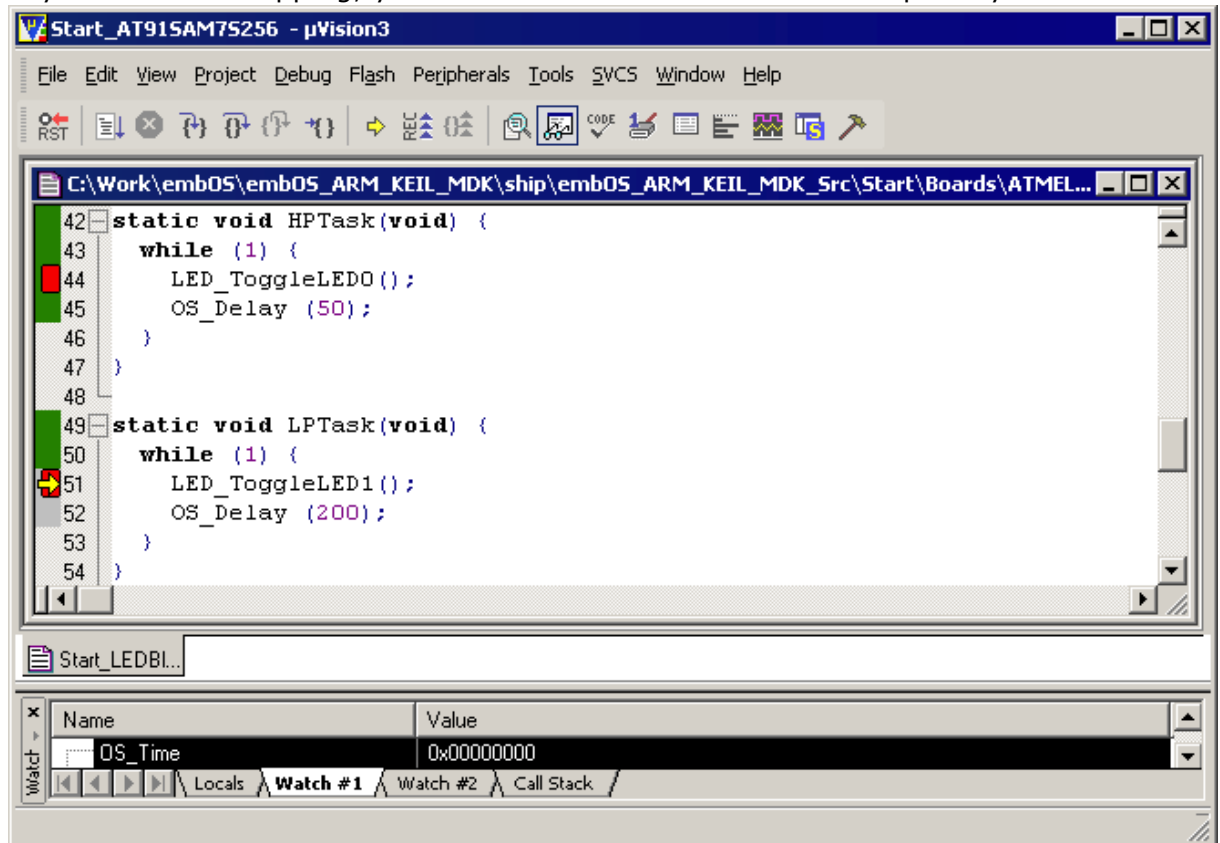
Before you step into `OS_Start()`, you should set two break points in the two tasks as shown below.



As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only. You may press 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 in the task with lower priority:



Continuing 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 which is 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 break-point there before you step over the delay in `Task1`.

The screenshot shows the Keil uVision3 IDE with the following code in the editor:

```

428 *      Please note:
429 *      This is basically the "core" of the idle loop.
430 *      This core loop can be changed, but:
431 *      The idle loop does not have a stack of its own, therefore no
432 *      functionality should be implemented that relies on the stack
433 *      to be preserved. However, a simple program loop can be program
434 *      (like toggling an output or incrementing a counter)
435 */
436 void OS_Idle(void) {           /* Idle loop: No task is ready to exec */
437     while (1) {
438     }
439 }
440

```

The Watch window at the bottom shows the following data:

Name	Value
OS_Time	0x00000000

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_Time`, shown in the watch window, `Task0` continues operation after expiration of the 50 ms delay.

The screenshot shows the Keil uVision3 IDE with the following code in the editor:

```

42 static void HPTask(void) {
43     while (1) {
44         LED_ToggleLEDO();
45         OS_Delay (50);
46     }
47 }
48
49 static void LPTask(void) {
50     while (1) {
51         LED_ToggleLED1();
52         OS_Delay (200);
53     }
54 }

```

The Watch window at the bottom shows the following data:

Name	Value
OS_Time	0x00000032



# Chapter 2

## Build your own application

---

This chapter provides all information to setup 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 First steps on page 9 and modify the project to fit your needs. Using a sample 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 for ARM

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, 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 library other than Release build library is used in your project.
- Additional low level init code 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.

Also ensure, that `main()` is called with the CPU running in supervisor or system mode.

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 `Start_2Task.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 a new `Lib` group to your project and add this library to the new group. Exclude all other library groups from build, delete unused `Lib` groups 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 ARM 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 ARM CPU is currently not supported, examine all `RTOSInit` files in the CPU-specific subfolders and select one which almost fits your CPU. You may have to modify `OS_InitHW()`, `OS_COM_Init()`, and the interrupt service routines for embOS timer tick and communication to `embOSView` and `__low_level_init()`.



# Chapter 3

## ARM specifics

---

## 3.1 CPU modes

**embOS** supports nearly all memory and code model combinations that KEILs ARM MDK C-Compiler supports.

**embOS** was compiled with interwork options. Therefore it is required to compile the projects with interwork option, too.

## 3.2 Available libraries

**embOS** for KEIL ARM MDK is shipped with 28 different libraries, one for each CPU mode / CPU core / endian mode and library type combination.

The libraries are named as follows:

**os<m><v><c><e><LibMode>.lib**

Parameter	Meaning	Values
m	Specifies the CPU mode	A: ARM mode T: THUMB mode
v	Specifies the CPU variant	4: ARMv4 instruction set 5: ARMv5 instruction set
e	Endian mode	L: Little B: Big
LibMode	Library mode	XR: Extreme Release R: Release S: Stack Check D: Debug SP: Stack check + profiling DP: Debug + profiling DT: Debug + trace

### Example:

osT4LR.lib the library for a project using THUMB mode, ARM 7/9 core, little endian mode and release build library type.

### Note:

The ARMv5 libraries can also be used for Cortex R CPUs.





# Chapter 4

## Compiler specifics

---

## 4.1 Standard system libraries

**embOS** for ARM KEIL MDK compiler may be used with standard system libraries for most of all projects.

Heap management and file operation functions of standard system libraries are not reentrant and can therefore not be used with **embOS**.

For heap management, **embOS** delivers its own thread safe functions which may be used. These functions are described in **embOS** CPU independent manual.

# Chapter 5

## Stacks

---

## 5.1 Task stack for ARM

All **embOS** tasks execute in system mode. The stack-size required is the sum of the stack-size of all routines plus basic stack size.

The basic stack size is the size of memory required to store the registers of the CPU plus the stack size required by **embOS**-routines.

For the ARM, this minimum task stack size is about 64 bytes.

## 5.2 System stack for ARM

The **embOS** system executes in supervisor mode. The minimum system stack size required by **embOS** is about 96 bytes (stack check build) However, since the system stack is also used by the application before the start of multitasking (the call to `OS_Start()`), for **embOS** software timers and high level interrupt handler. Therefore, the actual stack requirements depend on the application. The size of the system stack can be changed by modifying the stack size definition of the section `CSTACK` in the scatter file.

## 5.3 Interrupt stack for ARM

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 on the IRQ stack. For the interrupt routine itself, the supervisor stack is used. As a result, the required IRQ stack size depends on maximum nesting level of interrupts only.

The size of the interrupt stack can be changed by modifying the stack size definition of the section IRQ\_STACK in the scatter file. We recommend at least 128 bytes.

## 5.4 Stack specifics of the ARM family

Exceptions 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. When you intend to use FIQ in your system, you may have to modify the startup code and scatter files to implement an FIQ stack.





# Chapter 6

## Heap

---

## 6.1 Heap management

If you intend to use heap for dynamic memory allocation, the scatter files may have to be modified to define the required heap size.

### Example:

```
AT91SAM7S256_RAM 0x00200000 0x10000 {
  CODE 0x00200000 {
    vectors.o (Vect, +First)
    init*.o (Init)
    * (+RO)
  }

  DATA +0x0 {
    * (+RW,+ZI)
  }

  HEAP +0x0 EMPTY UNINIT 0x1000 {
  }

  CSTACK +0x0 EMPTY UNINIT 0x200 {
  }

  IRQ_STACK +0x0 EMPTY UNINIT 0x100 {
  }
}
```

# Chapter 7

## Interrupts

---

## 7.1 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`.

## 7.2 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_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

Simple interrupt routine:

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

## 7.3 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_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_ISR1(); // Handle interrupt
    }
}
```

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

## 7.4 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_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 7.1: Interrupt handler functions for ARM derivatives with built in vectored interrupt controller**

## 7.4.1 OS\_ARM\_InstallISRHandler(): Install an interrupt handler

### 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 7.2: OS\_ARM\_InstallISRHandler() parameter list**

### 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.



## 7.4.2 OS\_ARM\_EnableISR(): Enable a specific interrupt

### 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 7.3: 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.

### 7.4.3 OS\_ARM\_DisableISR(): Disable a specific interrupt

#### 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
ISRIndex	Index of the interrupt source which should be disabled.

**Table 7.4: 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.

## 7.4.4 OS\_ARM\_ISRSetPrio(): Set priority of a specific interrupt

### 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
ISRIndex	Index of the interrupt source which should be modified.
Prio	The priority which should be set for the specific interrupt.

**Table 7.5: 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.

## 7.4.5 OS\_ARM\_AssignISRSource(): Assign a hardware interrupt channel to an interrupt vector

### 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 );
```

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.

**Table 7.6: OS\_ARM\_AssignISRSource() parameter list**

### 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.

## 7.4.6 OS\_ARM\_EnableISRSource(): Enable an interrupt channel of a VIC-type interrupt controller

### 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 );;
```

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

**Table 7.7: OS\_ARM\_EnableISRSource() parameter list**

### 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.

## 7.4.7 OS\_ARM\_DisableISRSource(): Disable an interrupt channel of a VIC-type interrupt controller

### 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 );;
```

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

**Table 7.8: OS\_ARM\_DisableISRSource() parameter list**

### 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
```

## 7.5 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_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.

## 7.6 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 8

## MMU and cache support

---

## 8.1 MMU and cache support with embOS

embOS comes with functions to support the MMU and cache of ARMv5 (ARM9) 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.

## 8.2 MMU and cache handling for ARMv5 (ARM9 CPUs)

ARM9 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_DCACHE_Enable()</code>	Enable the data cache.
<code>OS_ARM_DCACHE_CleanRange()</code>	Clean data cache.
<code>OS_ARM_DCACHE_InvalidateRange()</code>	Invalidate the data cache.

**Table 8.1: MMU and cache handling for ARM9 CPUs**

## 8.2.1 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 8.2: OS\_ARM\_MMU\_InitTT() parameter list**

### Additional Information

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

## 8.2.2 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: 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
<a href="#">VIndex</a>	Virtual address index, which is the start address of the virtual memory address range with MBytes resolution. <a href="#">VIndex</a> = (virtual address >> 20)
<a href="#">PIndex</a>	Physical address index, which is the start address of the physical memory area range with MBytes resolution. <a href="#">PIndex</a> = (physical address >> 20)
<a href="#">NumEntries</a>	Specifies the size of the memory area in MBytes.

**Table 8.3: 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.

## 8.2.3 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 8.4: 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().

## 8.2.4 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 8.5: 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.

## 8.2.5 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 8.6: 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.



## 8.2.6 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.

## 8.2.7 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.

## 8.2.8 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
<a href="#">p</a>	Points to the base address of the memory area that should be updated.
<a href="#">NumBytes</a>	Number of bytes which have to be written from cache to memory.

**Table 8.7: 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. 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.

## 8.2.9 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 8.8: 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. 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:

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

## 8.3 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 Keil startup code calls the `__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) {
    //
    // Initialize SDRAM
    //
    _InitSDRAM();
    //
    // Init MMU and caches
    //
    OS_ARM_MMU_InitTT (&_TranslationTable[0]);
    //
    // SDRAM, the first MB remapped to 0 to map vectors to correct address,
    //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,
                             0x003, 0x003, 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 9

## STOP / WAIT Mode

---

## 9.1 Introduction

In case your controller does support some kind of power saving mode, it should be possible to use it also with embOS, as long as the timer keeps working and timer interrupts are processed. To enter that mode, you usually have to implement some special sequence in the function `OS_Idle()`, which you can find in embOS module `RTOSInit.c`.

Per default, the `wfi` instruction is executed in `OS_Idle()` to put the CPU into a low power mode.



# Chapter 10

## Technical data

---

## 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 2.500 bytes.

In the table below, which is for 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	44
Resource semaphore	16
Counting semaphore	8
Mailbox	24
Software timer	20

# Chapter 11

## Files shipped with embOS

---

**List of files shipped with embOS**

Directory	File	Explanation
root	*.pdf	Generic API and target specific documentation.
root	Release.html	Version control document.
root	embOSView.exe	Utility for runtime analysis, described in generic documentation.
Start\ BoardSupport\ 		Sample workspaces and project files for Keil MDK, contained in manufacturer specific sub folders.
Start\Inc	RTOS.h BSP.h	Include file for embOS, to be included in every C-file using embOS functions.
Start\Lib	os??_*.a	embOS libraries for Keil compiler.
Start\BoardSupport\ ..\Setup	OS_Error.c	embOS runtime error handler used in stack check or debug builds.
Start\BoardSupport\ ...\Setup\ 	*.*	CPU specific hardware routines for various CPUs.

Any additional files shipped serve as example.

# Index

---

<b>C</b>		<b>S</b>	
Cache .....	50	Sample application .....	12
CPU modes .....	22	Stack specifics .....	31
<b>H</b>		Standard system libraries .....	26
Heap management .....	34	Stepping .....	13
<b>I</b>		Syntax, conventions used .....	5
Installation .....	10	System stack .....	29
Interrupt stack .....	30, 36	<b>T</b>	
interrupts .....	36	Task stack .....	28
<b>L</b>			
Libraries .....	23		
<b>M</b>			
Memory requirements .....	66		
MMU .....	50		
multitasking application .....	11		
<b>O</b>			
OS_ARM_AssignISRSource() .....	44		
OS_ARM_DCACHE_CleanRange() .....	59		
OS_ARM_DCACHE_Enable() .....	58		
OS_ARM_DCACHE_InvalidateRange() .....	60		
OS_ARM_DisableISR() .....	42		
OS_ARM_DisableISRSource() .....	46		
OS_ARM_EnableISR() .....	41		
OS_ARM_EnableISRSource() .....	45		
OS_ARM_ICACHE_Enable() .....	57		
OS_ARM_InstallISRHandler() .....	40		
OS_ARM_ISRSetPrio() .....	43		
OS_ARM_MMU_AddTTEntires() .....	53		
OS_ARM_MMU_Enable() .....	54		
OS_ARM_MMU_GetVirtualAddr() .....	55		
OS_ARM_MMU_InitTT() .....	52		
OS_ARM_MMU_v2p() .....	56		
OS_irq_handler() .....	37		
OS_USER_irq_func() .....	38		

