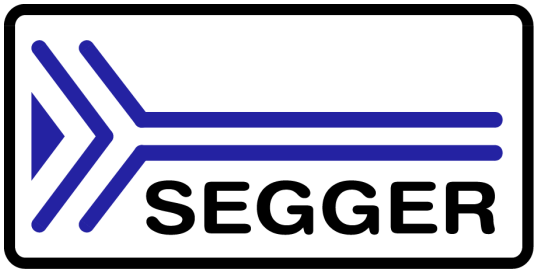
embOS

Real-Time
Operating System

CPU & Compiler
specifics for ARM cores
using emIDE

Document: UM01052
Software version 3.88h
Revision: 0
Date: December 23, 2013



A product of SEGGER Microcontroller GmbH & Co. KG

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

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
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 23, 2013

Software	Revision	Date	By	Description
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.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	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 CompactFlash 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 emIDE 9
1.1 Installation 10
1.2 First steps 11
1.3 The sample application Start2Tasks.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 cores 18
2.3 Change library mode 18
2.4 Select another CPU 19

3 ARM specifics 21
3.1 Naming conventions for prebuilt libraries 22

4 Compiler specifics 23
4.1 Standard system libraries 24
4.2 Reentrancy, thread local storage 24
4.3 Reentrancy, thread safe heap management 27
4.4 Vector Floating Point support VFP 28

5 Stacks 31
5.1 Task stack for ARM 32
5.2 System stack for ARM 32
5.3 Interrupt stack for ARM 32
5.4 Stack specifics of the ARM family 32

6 Interrupts 33
6.1 What happens when an interrupt occurs 34
6.2 Defining interrupt handlers in "C" 34
6.3 Interrupt handling without vectored interrupt controller 35
6.4 Interrupt handling with vectored interrupt controller 36
6.5 Interrupt stack switching 44
6.6 Fast interrupt FIQ 44

7 MMU and cache support 45
7.1 Introduction 46
7.2 MMU and cache handling for ARM9 CPUs 46

8 STOP / WAIT Mode 55
8.1 Introduction 56

9 Technical data 57
9.1 Memory requirements 58

10 Files shipped with embOS 59

Chapter 1

Using embOS with emIDE

The following chapter describes how to start with and use embOS for ARM cores and emIDE. You should follow these steps to become familiar with embOS for ARM cores and emIDE.

1.1 Installation

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

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, please keep all files in their respective sub directories. Make sure the files are not read only after copying.
If you received a zip-file, please extract it to any folder of your choice, preserving the directory structure of the zip-file.

Assuming that you are using the emIDE project manager to develop your application, no further installation steps are required. You will find a lot of prepared sample start projects, which you should use and modify to write your application. So follow the instructions of section “First steps” on page 11.

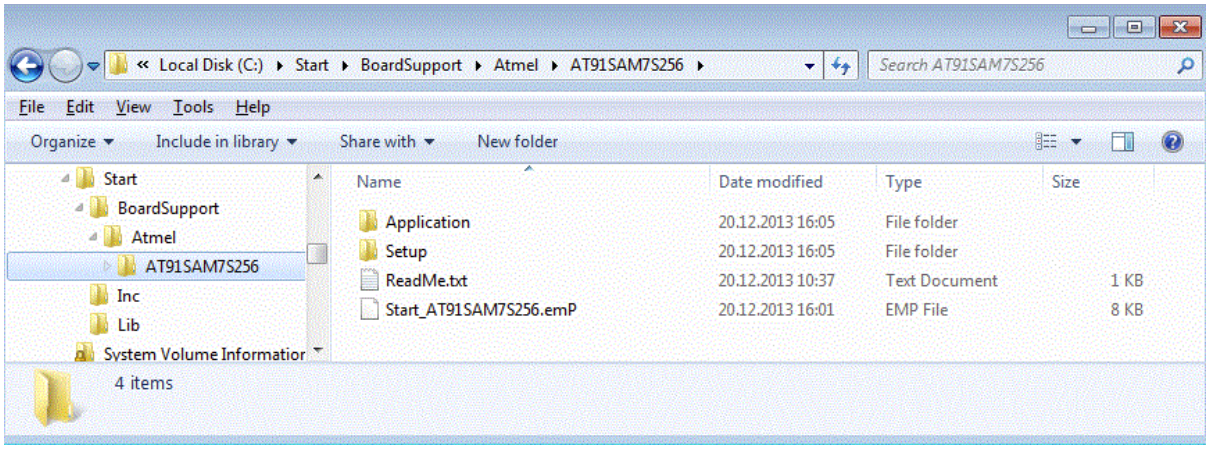
You should do this even if you do not intend to use the project manager for your application development to become familiar with embOS.

If you will not work with the emIDE, 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 the emIDE 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 (See "Installation" on page 10.) you are able to create your first multitasking application. You received ready to go sample emIDE project files and it is a good idea to use one of these as a starting point of all your applications.
Your embOS distribution contains one folder "Start\BoardSupport" which contains the sample project files and every additional files used to build your application.

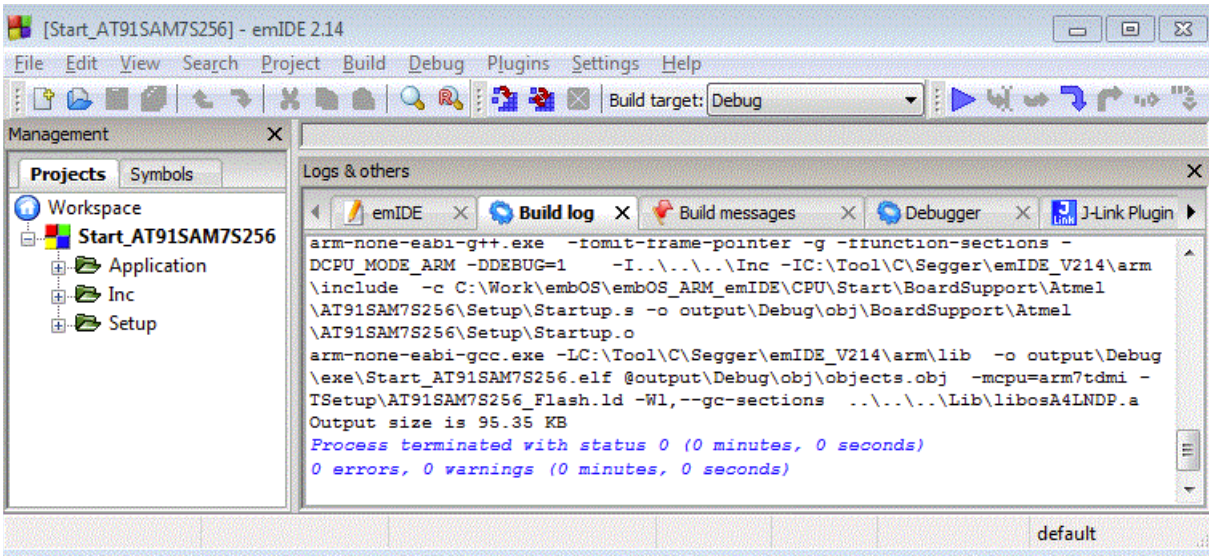
For the first step, you may use the project for AT91SAM7S256 CPU:



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 the sample workspace **Start\BoardSupport\Atmel\AT91SAM7S256** with the emIDE project manager (for example, by double clicking it).
- Build the start project. It should be build 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 sample application Start2Tasks.c

The following is a printout of the example application `Start_2Tasks.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.

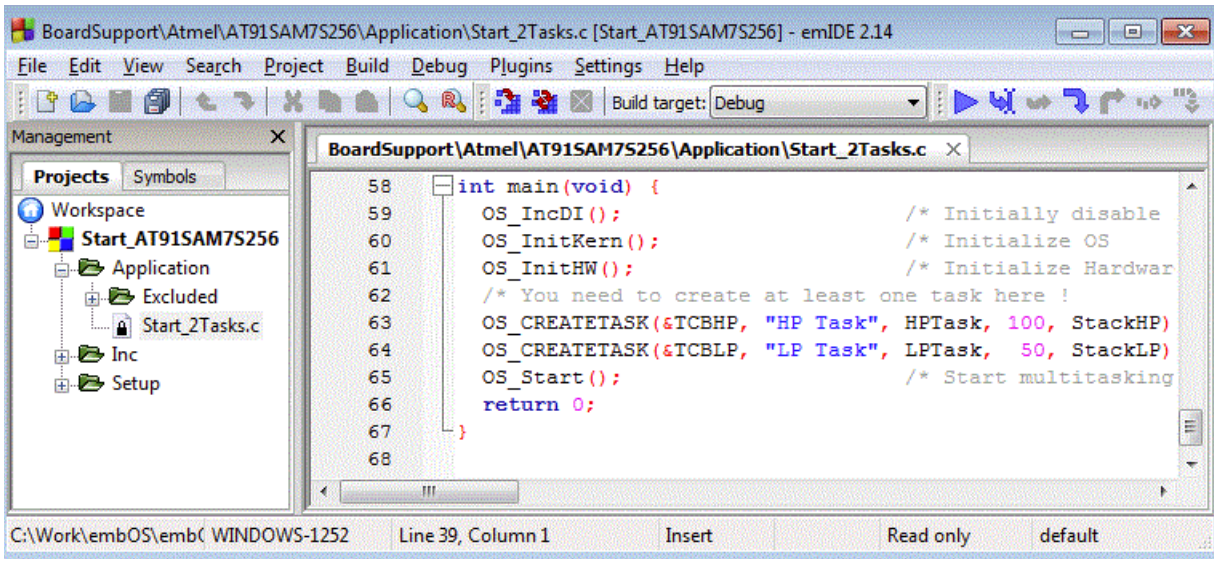
```
/*********************************************************
* SEGGER MICROCONTROLLER SYSTEME GmbH & Co.KG
* Solutions for real time microcontroller applications
**********************************************************
File    : Start2Tasks.c
Purpose : Skeleton program for embOS
--------- END-OF-HEADER ----------------------------------*/

#include "RTOS.h"

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

void HPTask(void) {
  while (1) {
    OS_Delay (10);
  }
}

void LPTask(void) {
  while (1) {
    OS_Delay (50);
  }
}

/*********************************************************
*
* main
*
*********************************************************/

void main(void) {
  OS_IncDI();                               /* Initially disable interrupts */
  OS_InitKern();                            /* Initialize OS */
  OS_InitHW();                              /* Initialize Hardware for OS */
  /* You need to create at least one task here ! */
  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

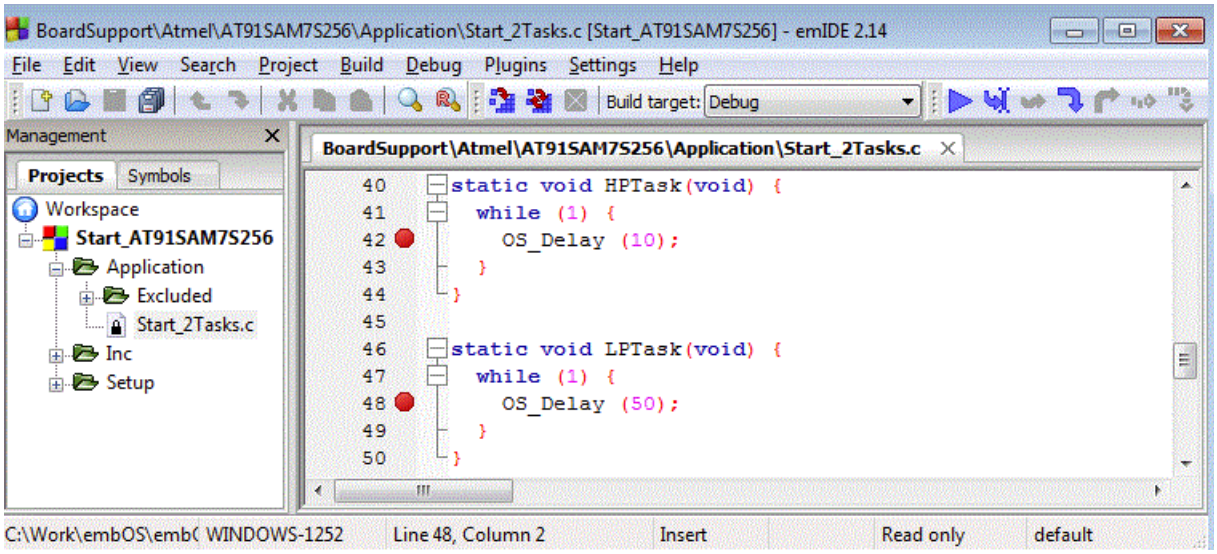
When starting the debugger, you will either see the startup code or the `main` function (see example screenshot below). The `main` function appears if the debugger option **Run and break at main** is selected, otherwise you may want to set a breakpoint at the function. Now you can step through the program.

`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 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.

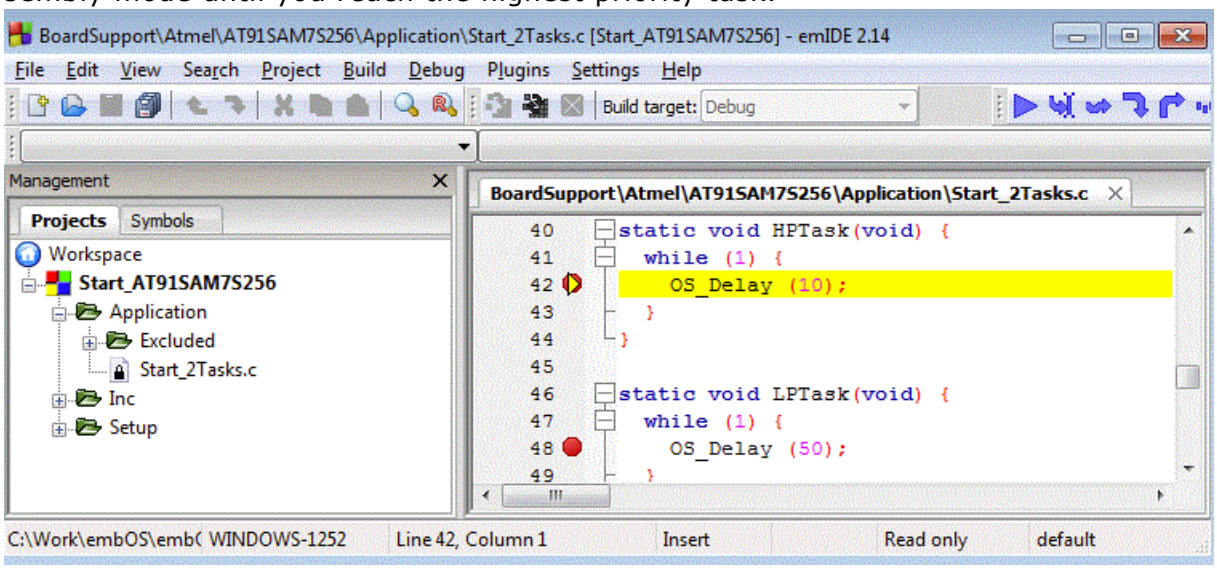


Before you continue stepping, you should set two break points in the two tasks as shown below:

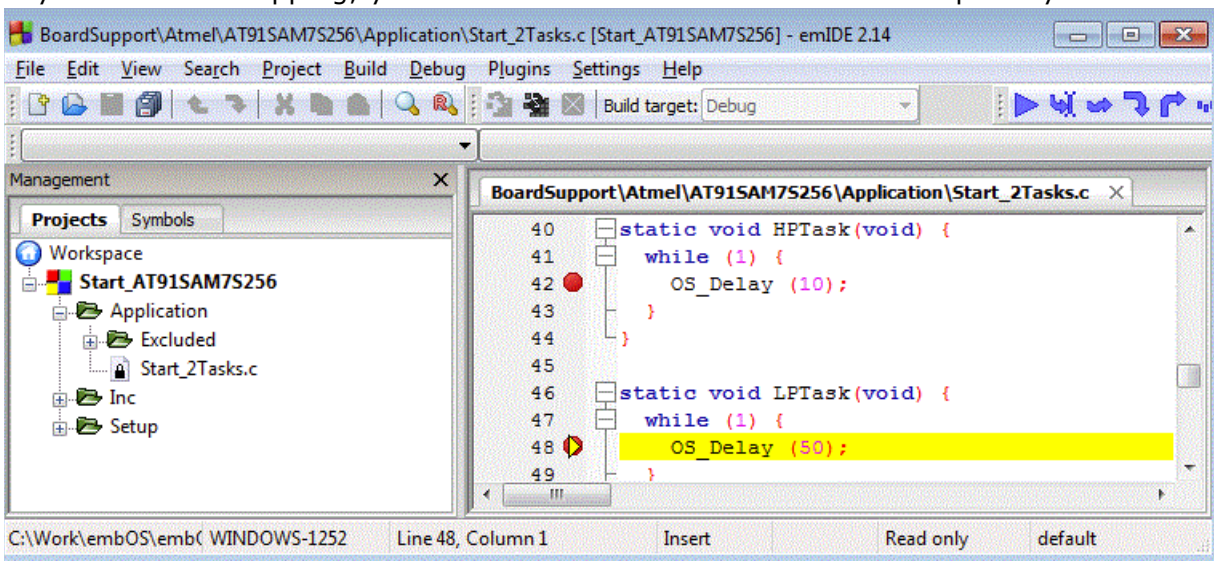


As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only. Y

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

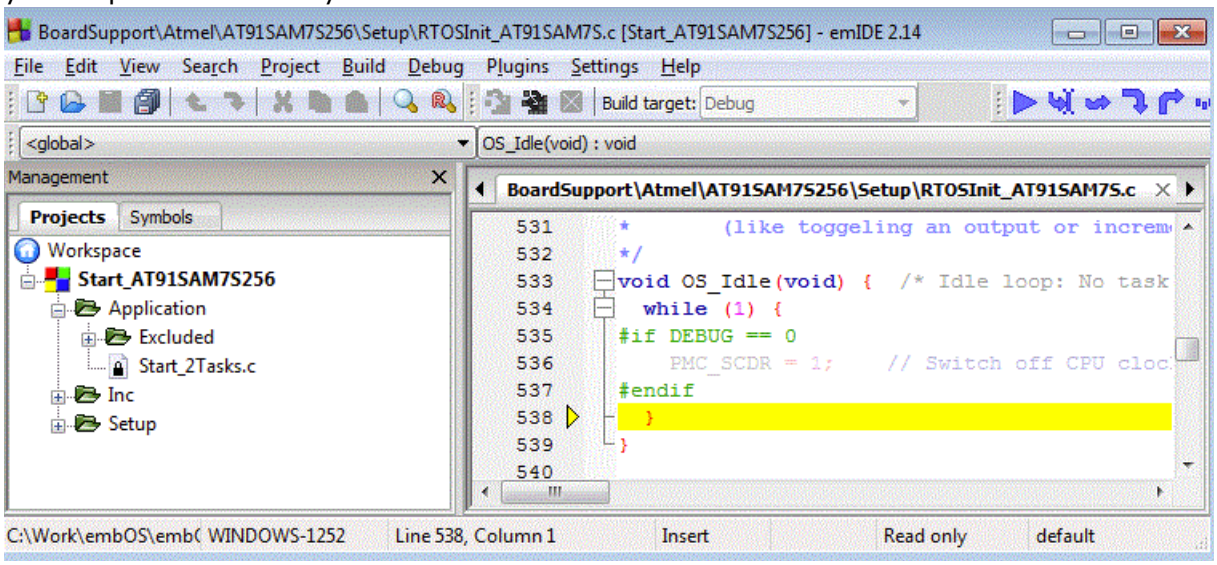


If you continue stepping, you will arrive in the task that has lower priority:



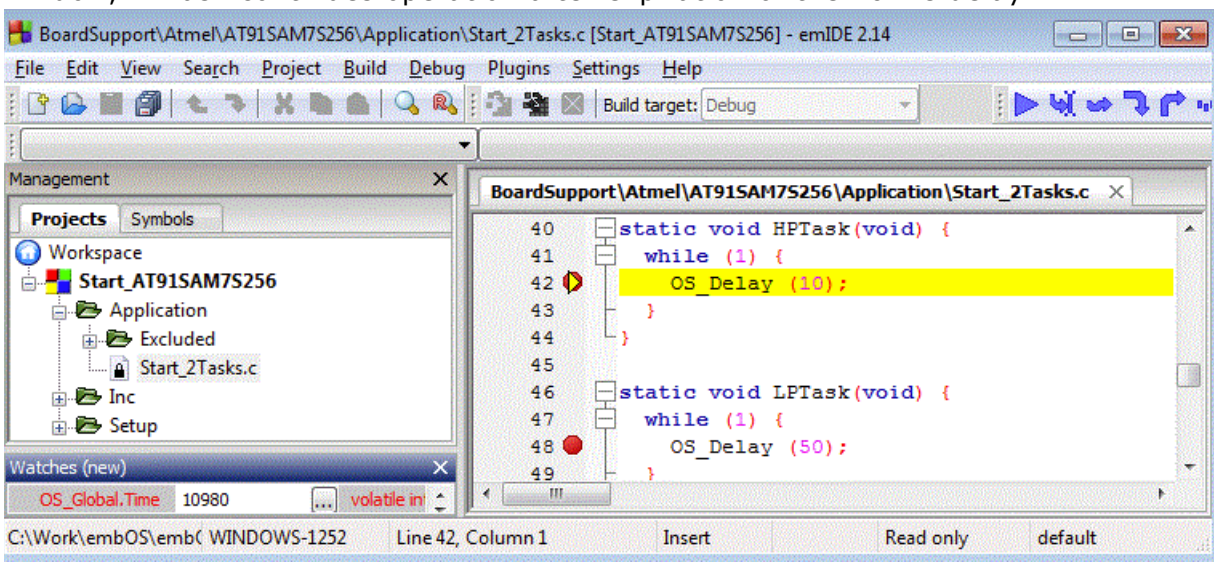
Continuing to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

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 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. Press Debug / Continue to enter the highest priority task again.

As can be seen by the value of **embOS** timer variable `OS_Time`, shown in the watch window, HPTask continues operation after expiration of the 10 ms delay.



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 cores

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>\Setup** 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>\Setup**.
  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 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 may wish to choose an other 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:

- Address the wished library in the build options, linker settings.
- Check and set `OS_DEBUG` define as preprocessor option. You may modify the OS_Config.h file to select an other library mode.

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 Naming conventions for prebuilt libraries

embOS supports nearly all memory and code model combinations that emIDE / GCC ARM compiler supports.

embOS for ARM cores and emIDE comes with 28 different libraries, one for each instruction set, endian mode and library mode combination. Concerning CPU mode and interwork option, all libraries use ARM mode and no interworking.
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 (not supported)
isa	Specifies the instruction set architecture.	4: v4T
		5: v5T
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:

`libosA4LNR.a` is the library for a project using ARM 7 core, little endian mode and release build library type.

Chapter 4

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 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.

4.4 Vector Floating Point support VFP

Some ARM MCUs come with an integrated vectored floating point unit VFP.
When selecting the CPU and activating the VFP support in the project options, the compiler and linker will add efficient code which uses the VFP when floating point operations are used in the application.
With embOS, the VFP registers have to be saved and restored when preemptive or cooperative task switches are performed.
For efficiency reasons, embOS does not save and restore the VFP registers for every task automatically. The context switching time and stack load are therefore not affected when the VFP unit is not used or needed.
Saving and restoring the VFP registers can be enabled for every task individually by extending the task context of the tasks which need and use the VFP.

4.4.1 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.
When Thread-local Storage (TLS) is also needed in a task, the new embOS function `OS_ExtendTaskContext_TLS_VFP()` has to be called to extend the task context for TLS and VFP.

4.4.2 Using the VFP in interrupt service routines

Using the VFP in interrupt service routines requires additional functions to save and restore the VFP registers.
As the GCC compiler does not add additional code to save and restore the VFP registers on entry and exit of interrupt service routines, it is the users responsibility to save the VFP registers on entry of an interrupt service routine when the VFP is used in the ISR.
embOS delivers two functions to save and restore the VFP context in an interrupt service routine.

4.4.2.1 OS_VFP_Save()

Description

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

Prototype

```
void OS_VFP_Save(void)
```


Return value

None.

Additional Information

OS_VFP_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() function, the interrupt service routine may use the VFP for calculation without destroying the saved content of the VFP registers.
To restore the registers, the ISR has to call OS_VFP_Restore() at the end.
The function may be used in any ISR regardless the priority. It is not restricted to low priority interrupt functions.

4.4.2.2 OS_VFP_Restore()

Description

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

Prototype

`void` OS_VFP_Restore(`void`)

Return value

None.

Additional Information

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

Example of a low priority interrupt service routine using VFP:

```
void ADC_ISR_Handler(void) {
  OS_VFP_Save();          // Save VFP registers
  OS_EnterInterrupt();
  DoSomeFloatOperation();
  OS_LeaveInterrupt();
  OS_VFP_Restore();       // Restore VFP registers
}
```

In low priority interrupt service routines, `OS_EnterInterrupt()` is called to inform embOS that an interrupt handler is running and blocks task switches until `OS_LeaveInterrupt()` is called.
After calling `OS_EnterInterrupt()`, or `OS_EnterNestableInterrupt()`, any embOS function which is allowed to be called from an ISR may be called.

Example of a high priority interrupt service routine using VFP:

```
void ADC_ISR_Handler(void) {
  OS_VFP_Save();          // Save VFP registers
  DoSomeFloatOperation();
  OS_VFP_Restore();       // Restore VFP registers
}
```

In interrupt service routines running at higher priority, no embOS functions except `OS_VFP_Save()` and `OS_VFP_Restore()` may be called.
Not even `OS_EnterInterrupt()`.

4.4.3 Compiler and linker options.

The selection of different CPU cores or options like VFP support has to be done by linker, compiler and assembler options.
The options have to be passed to the tool by definitions in the make-files, or when using the IDE, the options have to be defined in the "Settings" dialog for the project.
The options passed to the tools have to be defined for compiler, linker and assembler separately and have to be the same for all tools.
Beside other options, the most important options are the options to select the CPU core.

4.4.3.1 Options to select a ARM 7 core

-mcpu=arm7tdmi -marm

4.4.3.2 Options to select a ARM 9 core

-mcpu=arm9e -marm

4.4.3.3 Options to select a ARM 9 core with VFP support

-mcpu=arm9e -marm -mfpu=vfp -mfloat-abi=softfp

Chapter 5

Stacks

5.1 Task stack for ARM

All embOS tasks execute in system mode. Every embOS task has its own individual stack which can be located in any memory area. The required stacksize for a task is the sum of the stack-size used by all functions for local variables and parameter passing, 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 7/9, this minimum basic task stack size is about 68 bytes.

5.2 System stack for ARM

The embOS system executes in supervisor mode. The minimum system stack size required by embOS is about 136 bytes (stack check & profiling build). However, since the system stack is also used by the application before the start of multitasking (the call to `OS_Start()`), and because software-timers and "C"-level interrupt handlers also use the systemstack, the actual stack requirements depend on the application.
The size of the system stack can be changed by modifying "SVC_STACK_SIZE" in your `*.ld` linker script file.

5.3 Interrupt stack for ARM

If a normal hardware exception occurs, the ARM core switches to IRQ mode,which uses 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. Forthe interrupt routine itself, the supervisor stack is used. The size of the interrupt stack can be changed by modifying "IRQ_STACK_SIZE" in your `*.ld` linker script file.
Every interrupt requires 28bytes on the interrupt stack.
The maximum interrupt stack size required by the application can be calculated as isMaximum interrupt nesting level * 28 bytes.lp For task switching from within an interrupt handler, it is required, that the end address of the interrupt stack is aligned to an 8 byte boundary. This alignment is forced during stack pointer initialization in the startup routine. Therefore, an additional margin of about 8 bytes should be added to the calculated maximum interrupt stack size. For standard applications, we recommend at least 92 to 128 bytes of IRQ stack.

5.4 Stack specifics of the ARM family

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 modifiedto initialize the stacks.

Chapter 6

Interrupts

6.1 What happens when an interrupt occurs

- The CPU-core receives an interrupt request
- As soon as the interrupts are enabled, the interrupt is executed
- The CPU switches to the Interrupt stack
- The CPU saves PC and flags in registers `LR_irq` and `SPSR_irq`
- The CPU jumps to the vector address `0x18`
- embOS OS_IRQ_SERVICE: save scratch registers
- embOS OS_IRQ_SERVICE: save `LR_irq` and `SPSR_irq`
- embOS OS_IRQ_SERVICE: switch to supervisor mode
- embOS OS_IRQ_SERVICE: 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 (OS_USER_irq_func).
- embOS OS_IRQ_SERVICE: switch to IRQ mode
- embOS OS_IRQ_SERVICE: restore `LR_irq` and `SPSR_irq`
- embOS OS_IRQ_SERVICE: pop scratch registers
- Return from interrupt.

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

6.2 Defining interrupt handlers in "C"

Interrupt handlers called from 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 `OS_IRQ_SERVICE()`.
Depending on the interrupting source, it may be required to reset the interrupt pending condition of the related peripherals.

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
  }
}
```


6.3 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 `OS_IRQ_SERVICE()` is called.
- `OS_IRQ_SERVICE()` saves registers and switches to supervisor mode.
- `OS_IRQ_SERVICE()` 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 `OS_IRQ_SERVICE()`.
- `OS_IRQ_SERVICE()` restores registers and performs a return from interrupt.

`OS_USER_irq_func()` (usually defined in module `UserIRQ.c`) has to examine and handle all application specific interrupts.

Example of a "simple" OS_User_irq_func()

```
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.

6.4 Interrupt handling with vectored interrupt controller

For ARM derivates 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 derivate with vectored interrupt controller, please ensure that `OS_IRQ_SERVICE()` 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 `OS_IRQ_SERVICE()` is called by CPU or interrupt controller.
- `OS_IRQ_SERVICE()` saves registers and switches to supervisor mode.
- `OS_IRQ_SERVICE()` 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 embOS 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 finished.
- `OS_irq_handler()` returns to `OS_IRQ_SERVICE()`.
- `OS_IRQ_SERVICE()` restores registers and performs a return from interrupt.

Please note, that 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. Please 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:

6.4.1 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 6.1: 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.

6.4.2 OS_ARM_EnableISR(): Enable 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 6.2: 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.

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

6.4.3 OS_ARM_DisableISR(): Disable 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 6.3: 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.

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

6.4.4 OS_ARM_ISRSetPrio(): Set priority of 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 6.4: 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. Please refer to CPU specific manuals about allowed priority levels.

This function can not be used to modify the interrupt priority for interrupt controllers of the VIC type. The interrupt priority with VIC type controllers depends on the interrupt vector number and can not be changed.

6.4.5 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

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

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

Table 6.5: 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 derivates and has to be taken from the hardware manual of the CPU.

6.4.6 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

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

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

Table 6.6: 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 derivates and has to be taken from the hardware manual of the CPU.

6.4.7 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

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

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

Table 6.7: 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 derivates 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
```

6.5 Interrupt stack switching

Since 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 primary interrupt handler in `rtosvect.s` only.

6.6 Fast interrupt FIQ

FIQ interrupt can not be used with embOS functions, it is reserved for high speed user functions.
`FIQ` is never disabled by embOS.
Never call any embOS function from an `FIQ` handler.
Do not assign any embOS interrupt handler to `FIQ`.

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

Chapter 7

MMU and cache support

7.1 Introduction

embOS comes with functions to support the MMU and cache of ARM9 CPUs which allow virtual-to-physical address mapping with sections of one MegaByte 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 re-quired.

7.2 MMU and cache handling for ARM9 CPUs

ARM9 CPUs with MMU and cache have separate data and instruction caches. The data cache can only be enabled when the MMU is set up and enabled. embOS delivers the following functions to setup and handle the MMU and caches.

7.2.1 OS_ARM_MMU_InitTT(): Initialize the MMU translation table

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_InitTT (unsigned int* pTranslationTable)
```

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

Table 7.1: OS_ARM_MMU_InitTT() parameter list

Additional information

This function does not need to be called, if the translation table is located in ROM. The base address of the translation table has to be aligned on a 16Kbyte boundary.

7.2.2 OS_ARM_MMU_AddTTEntries(): Add address entries to the table

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
pTRanslationTable	Points to the base address of the translation table
CacheMode	Specifies the cache operating mode which should be used for the selected area. May be one of the follow-ing modes: OS_ARM_CACHEMODE_NC_NB OS_ARM_CACHEMODE_C_NB OS_ARM_CACHEMODE_NC_B OS_ARM_CACHEMODE_C_B
VIndex	Virtual address index, which is the start address of the virtual memory address range with MegaByte resolution. VIndex = (virtual address >> 20)
PIndex	Physical address index, which is the start address of the physical memory area range with MegaByte resolution. PIndex = (physical address >> 20).
NumEntries	Specifies the size of the memory area in MegaBytes.

Table 7.2: 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 trans-lation table for the specified memory area.

7.2.3 OS_ARM_MMU_Enable(): Enable the MMU

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 TranslationTable);
```

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

Table 7.3: OS_ARM_DisableISR() parameter list

Additional information

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

7.2.4 OS_ARM_ICACHE_Enable(): Enable the instruction cache

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 the instruction cache.

7.2.5 OS_ARM_DCACHE_Enable(): Enable the data cache

Description

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

Prototype

```
void OS_ARM_ICACHE_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.

7.2.6 OS_ARM_DCACHE_CleanRange(): Clean data cache

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

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

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

Table 7.4: 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.

7.2.7 OS_ARM_DCACHE_InvalidateRange(): Invalidate the data cache

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 in re-reading the data from memory into the cache, when the specified area is accessed again.

Prototype

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

Parameter	Description
`p`	Points to the base address of the memory area that should be invalidated.
`NumBytes`	Number of bytes which have to be invalidated.

Table 7.5: OS_ARM_DisableISRSource() 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.
This requires, 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.

Chapter 8

STOP / WAIT Mode

8.1 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`.

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 release build, you can find minimum RAM size requirements for embOS resources. Note that the sizes depend on selected embOS library mode.

embOS resource	RAM [bytes]
Task control block	32
Resource semaphore	8
Counting semaphore	4
Mailbox	20
Software timer	20

Chapter 10

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 emIDE, contained in manufacturer specific subfolders.
Start\ BoardSupport*\ Application	*.*	Sample programms to serve as a start.
Start\ BoardSupport*\ Setup	*.*	CPU specific hardware routines
Start\ BoardSupport*\ Setup	OS_Error.c	embOS runtime error handler used in stack check or debug builds.
Start\Inc	*.h	Include files.
Start\Inc	BSP.h	Include file for Board Support packages, to be included in every C-file using BSP-functions.
Start\Inc	OS_Config.h	Include file for embOS library mode configuration, included by RTOS.h.
Start\Inc	RTOS.h	Include file for embOS, to be included in every C-file using embOS functions.
Start\Lib	libos*.a	embOS libraries.

Any additional files shipped serve as example.

Index

Symbols

__malloc_lock() 27
__malloc_unlock() 27

C

Compiler options 30

F

FIQ 44

H

Heap management 27

I

Idle-loop 14
Installation 10
Interrupt stack 32, 44
Interrupts, FIQ 44
IRQ_STACK 32

L

Library mode 18

M

Memory models 22
Memory requirements 58

O

OS_ARM_AssignISRSource() 41
OS_ARM_DCACHE_CleanRange() 52
OS_ARM_DCACHE_InvalidateRange() 53
OS_ARM_DCHACHE_Enable() 51
OS_ARM_DisableISR() 39
OS_ARM_DisableISRSource() 43
OS_ARM_EnableISR() 38
OS_ARM_EnableISRSource() 42
OS_ARM_ICACHE_Enable() 50
OS_ARM_InstallISRHandler() 37
OS_ARM_ISRSetPrio() 40
OS_ARM_MMU_AddTTEntries() 48
OS_ARM_MMU_Enable() 49
OS_ARM_MMU_InitTT() 47
OS_ExtendTaskContext_TLS() 25
OS_ExtendTaskContext_TLS_VFP() 26
OS_ExtendTaskContext_VFP() 28
OS_MallocLock.c 27
OS_VFP_Restore() 29
OS_VFP_Save() 28

R

Reentrancy 24

S

Sample application 12
Select another CPU 19
Stacks 31
  CSTACK 32
  Interrupt stack 32
  System stack 32
Stepping through the sample application 13
Syntax, conventions used 5
System libraries 24
System stack 32

V

Vector Floating Point support 27–28
VFP 28

