embOS

Real-Time Operating System

CPU & Compiler specifics for ARM using IAR Embedded Workbench

Document: UM01002
Software Version: 5.02
Revision: 0
Date: July 10, 2018

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

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

Copyright notice

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

Manual versions

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

Print date: July 10, 2018

Software	Revision	Date	By	Description
5.02	0	180710	MC	New software version.
4.40	0	180201	MC	New software version.
4.38	0	171013	MC	New software version.
4.36	0	170802	MC	New software version.
4.34	1	170707	MC	Updated chapter "libraries" to include descriptions for both IAR EWARM 6/7 and IAR EWARM 8.
4.34	0	170330	MC	New software version.
4.16	0	160406	RH	New software version.
4.14	0	151130	TS	New software version.
4.12b	0	150928	TS	New software version.

3.90	0	140310	AW	New software version.
				Global enable and disable interrupts. Described in the generic manual.
				Chapter 4, "Debug outputs, printf" and `SWI_Handler()` description added.
				Chapter 7, "MMU and cache support", function descripition added:
				`OS_ARM_MMU_v2p()`
				`OS_ARM_MMU_GetVirtualAddr()`
				`OS_ARM720_MMU_v2p()`
				`OS_ARM720_MMU_GetVirtualAddr()`

3.88a	0	130503	AW	New software version.
3.86n	1	130312	AW	Chapter 4, "Thread safe system libraries with IAR compiler V6.4 or newer" corrected, one more required linker directive added.
3.86n	0	121210	AW	New software version. Chapter 4, "Thread safe system libraries with IAR compiler V6.4 or newer" added to describe the procedure to activate thread safe library support with newer IAR compiler.
3.86l	0	121122	AW	New software version.
3.86g	0	120806	AW	New software version.
3.86f	0	120801	AW	New software version.
3.84c	0	120110	TS	New software version.
3.84a	0	111214	TS	New software version. New function for VFP/Neon support in chapter 4: `OS_ExtendTaskContext_NEON()`
3.84	0	111105	AW	New software version.

3.82v	0	110715	AW	New software version.
				New functions for thread locale storage and VFP support in chapter 4:
				`OS_ExtendTaskContext_TLS()`
				`OS_ExtendTaskContext_TLS_VFP()`
				`OS_ExtendTaskContext_VFP()`

3.82t	0	110503	AW	New software version. Project settings and macros corrected for EWARM6. New library mode `OS_LIBMODE_DP` with low optimization.
3.82m	0	101124	AW	New software version. Thread-local storage and thread safe library support for IAR compiler V6 added, Chapter "Compiler specifics".
3.82	1	090918	TS	New software version.
3.80	0	090625	SK	New software version. Chapter "Using embOS ARM": Sample corrected.

Software	Revision	Date	By	Description
3.62	2	090513	SK	Chapter “ARM core version specifics”: “Naming conventions for prebuild libraries compatible to IAR EW V5.x” corrected.
3.62	1	081209	SK	Chapter footer corrected.
3.62	0	080904	SK	New software version. Chapter “C-SPY plug-in” added.

About this document

Assumptions

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

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

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

How to use this manual

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

Typographic conventions for syntax

This manual uses the following typographic conventions:

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

Table of contents

Chapter 1

Using embOS

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

1.1 Installation

embOS is shipped as a zip-file in electronic form.

To install it, proceed as follows:

Extract the zip-file to any folder of your choice, preserving the directory structure of this file. Keep all files in their respective sub directories. Make sure the files are not read only after copying.

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

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

If you do not or do not want to work with the IDE, you should: Copy either all or only the library-file that you need to your work-directory. The advantage is that when switching to an updated version of embOS later in a project, you do not affect older projects that use embOS, too. embOS does in no way rely on an IDE, it may be used without the IDE using batch files or a make utility without any problem.

1.2 First Steps

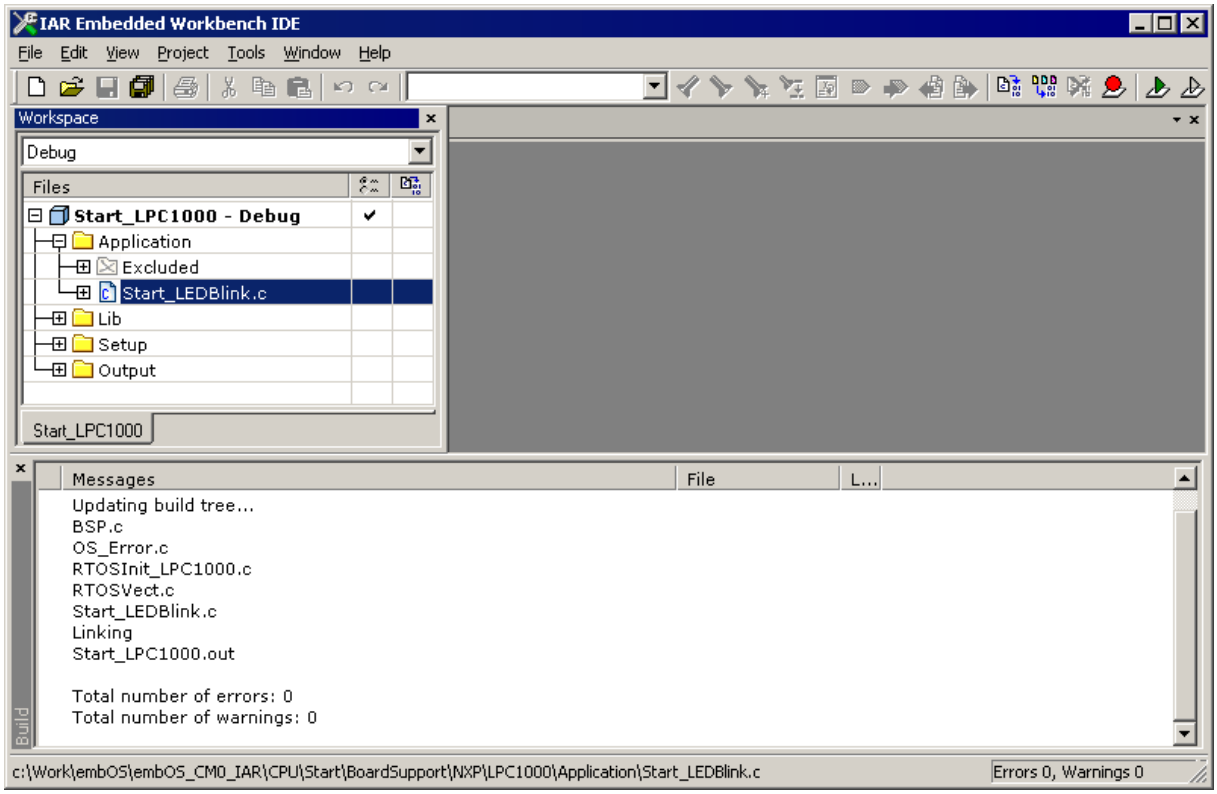
After installation of embOS you can create your first multitasking application. You have received several ready to go sample start workspaces and projects and every other files needed in the subfolder `Start`. It is a good idea to use one of them as a starting point for all of your applications. The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

To start with, you may use any project from `BoardSupport` subfolder.

To get your new application running, you should proceed as follows:

- Create a work directory for your application, for example `c:\work`.
- Copy the whole folder `Start` which is part of your embOS distribution into your work directory.
- Clear the read-only attribute of all files in the new `Start` folder.
- Open one sample workspace/project in `Start\BoardSupport\<DeviceManufacturer>\<CPU>` with your IDE (for example, by double clicking it).
- Build the project. It should be built without any error or warning messages.

After generating the project of your choice, the screen should look like this:



For additional information you should open the ReadMe.txt file which is part of every specific project. The ReadMe file describes the different configurations of the project and gives additional information about specific hardware settings of the supported eval boards, if required.

1.3 The example application OS_StartLEDBlink.c

The following is a printout of the example application `OS_StartLEDBlink.c`. It is a good starting point for your application. (Note that the file actually shipped with your port of embOS may look slightly different from this one.)

What happens is easy to see:

After initialization of embOS; two tasks are created and started. The two tasks are activated and execute until they run into the delay, then suspend for the specified time and continue execution.

```
/*********************************************************************
*                SEGGER Microcontroller GmbH & Co. KG                *
*                        The Embedded Experts                        *
*********************************************************************

-------------------------- END-OF-HEADER -----------------------------
File    : OS_StartLEDBlink.c
Purpose : embOS sample program running two simple tasks, each toggling
          a LED of the target hardware (as configured in BSP.c).
*/

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

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

static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_TASK_Delay(50);
  }
}

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

/*********************************************************************
*
*       main()
*/
int main(void) {
  OS_Init();    // Initialize embOS
  OS_InitHW();  // Initialize required hardware
  BSP_Init();   // Initialize LED ports
  OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
  OS_TASK_CREATE(&TCBLP, "LP Task",  50, LPTask, StackLP);
  OS_Start();   // Start embOS
  return 0;
}

/************************** End of file ******************************/
```

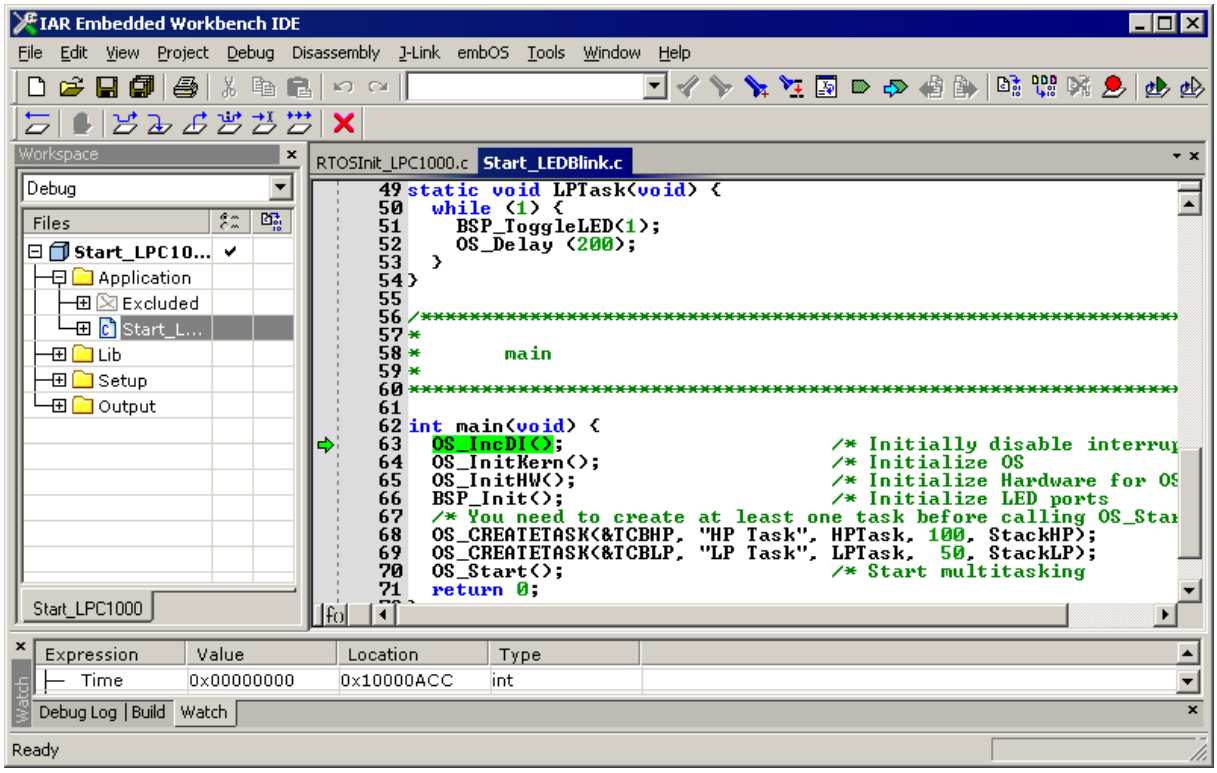

1.4 Stepping through the sample application

When starting the debugger, you will see the `main()` function (see example screen shot below). The `main()` function appears as long as project option `Run to main` is selected, which it is enabled by default. Now you can step through the program.

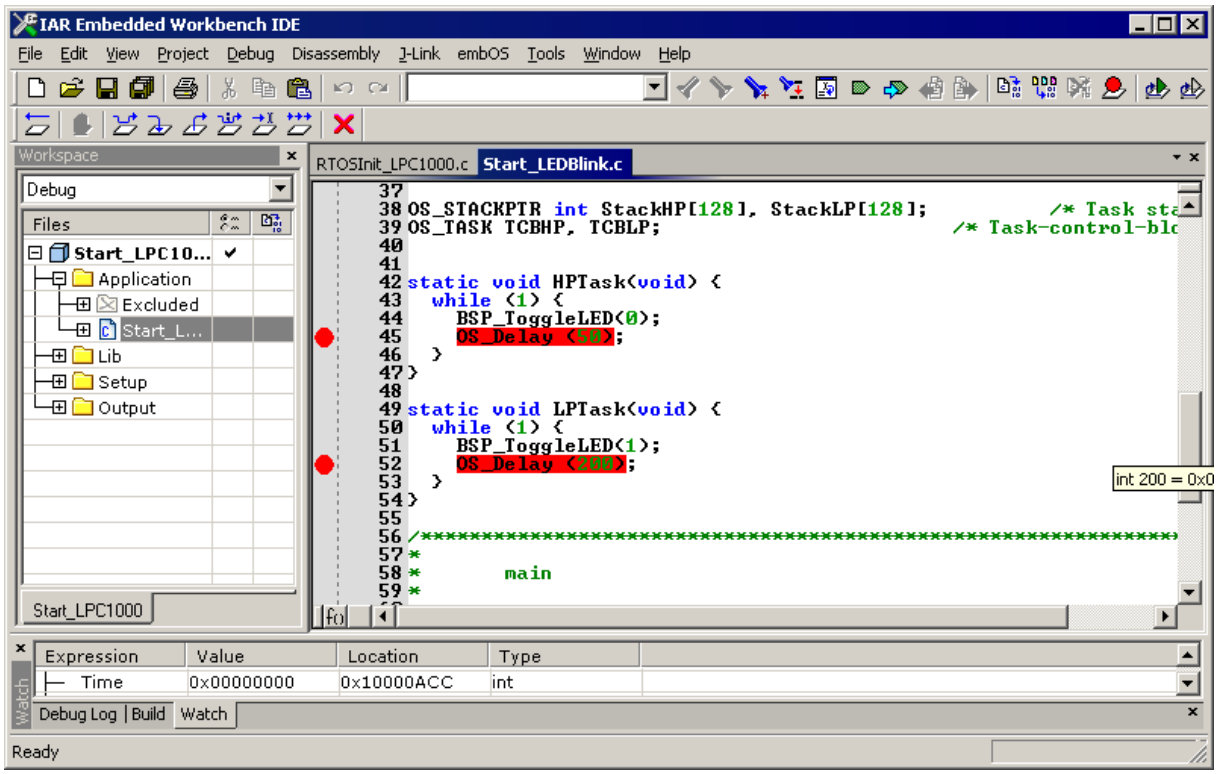
`OS_Init()` is part of the embOS library and written in assembler; you can there fore only step into it in disassembly mode. It initializes the relevant OS variables.

`OS_InitHW()` is part of `RTOSInit.c` and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the system tick interrupt for embOS. Step through it to see what is done.

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

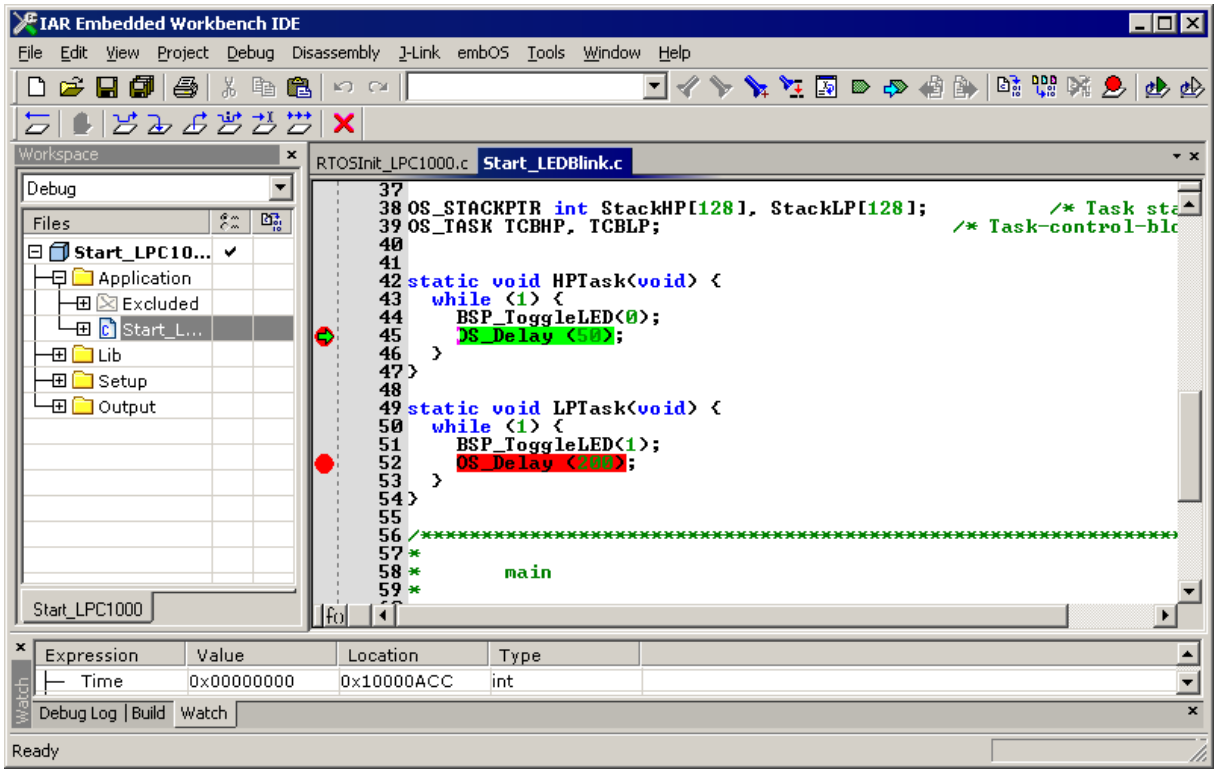


Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.

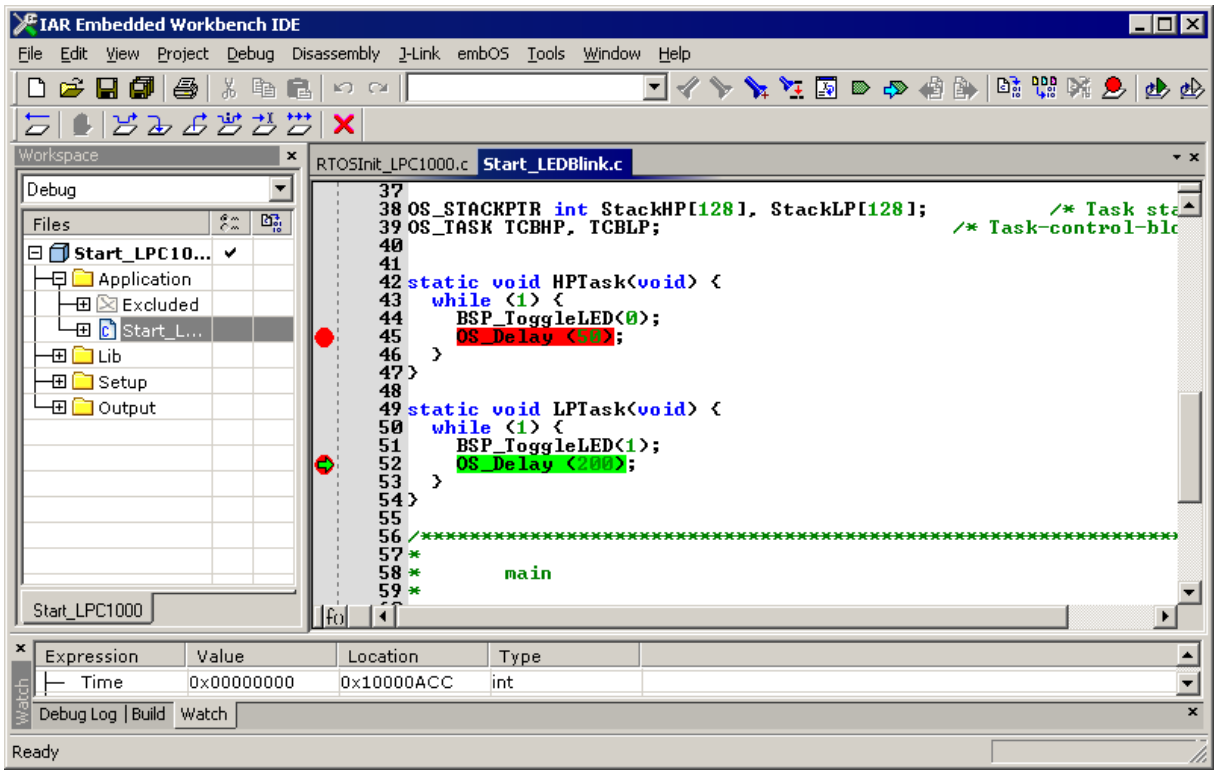


As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only.

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

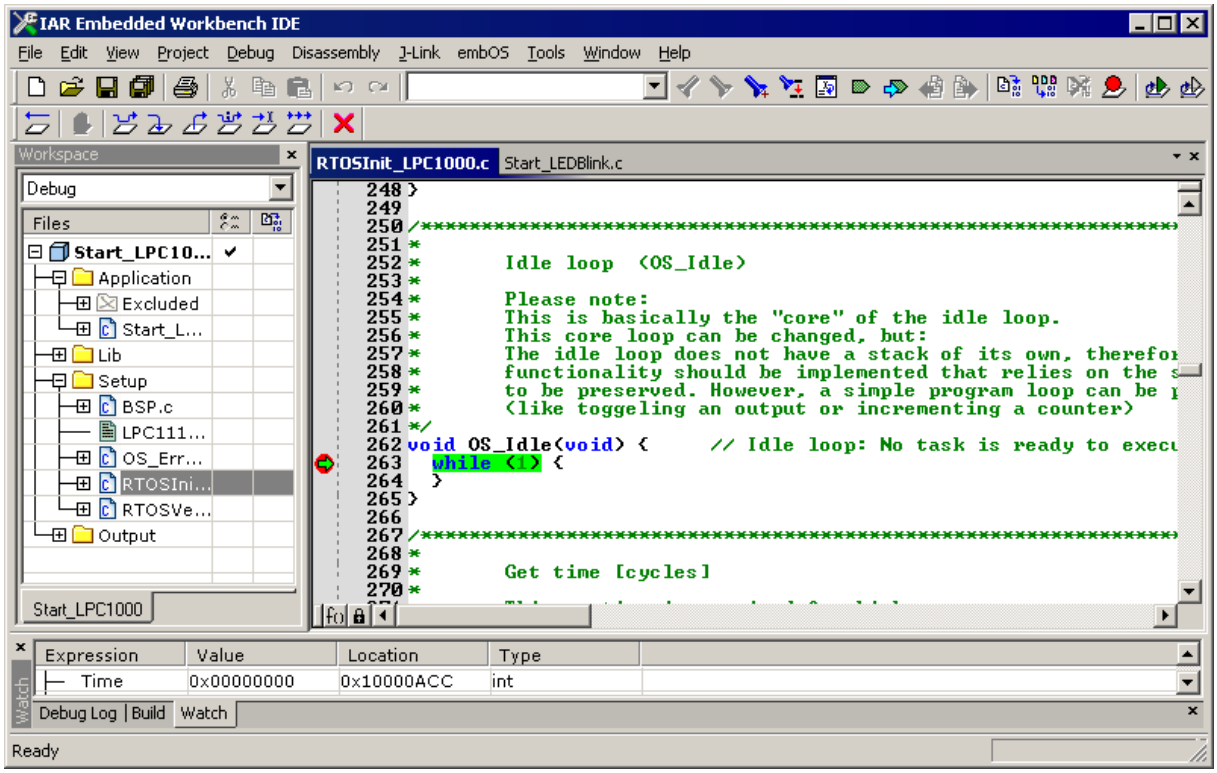


If you continue stepping, you will arrive at the task that has lower priority:



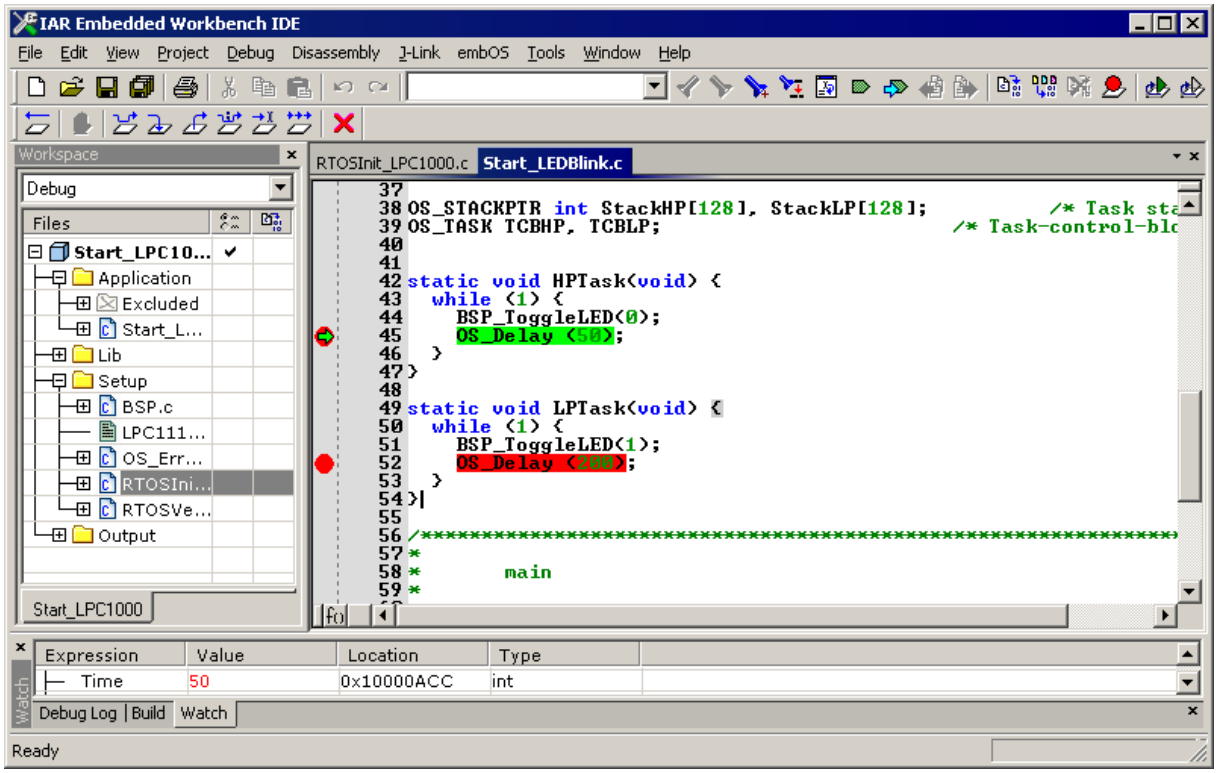
Continue to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

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



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

As can be seen by the value of embOS timer variable `OS_Global.Time`, shown in the Watch window, `HPTask()` continues operation after expiration of the 50 system tick delay.



Chapter 2

Build your own application

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

2.1 Introduction

To build your own application, you should always start with one of the supplied sample workspaces and projects. Therefore, select an embOS workspace as described in chapter *First Steps* on page 11 and modify the project to fit your needs. Using an embOS start project as starting point has the advantage that all necessary files are included and all settings for the project are already done.

2.2 Required files for an embOS

To build an application using embOS, the following files from your embOS distribution are required and have to be included in your project:

- `RTOS.h` from subfolder `Inc\`.
  This header file declares all embOS API functions and data types and has to be included in any source file using embOS functions.
- `RTOSInit*.c` from one target specific `BoardSupport\<Manufacturer>\<MCU>` subfolder. It contains hardware-dependent initialization code for embOS. It initializes the system timer interrupt and optional communication for embOSView via UART or JTAG.
- `OS_Error.c` from one target specific subfolder `BoardSupport\<Manufacturer>\<MCU>`. The error handler is used if any debug library is used in your project.
- One embOS library from the subfolder `Lib\`.
- Additional CPU and compiler specific files may be required according to CPU.

When you decide to write your own startup code or use a low level `init()` function, ensure that non-initialized variables are initialized with zero, according to C standard. This is required for some embOS internal variables. Your `main()` function has to initialize embOS by calling `OS_Init()` and `OS_InitHW()` prior to any other embOS functions that are called.

2.3 Change library mode

For your application you might want to choose another library. For debugging and program development you should use an embOS debug library. For your final application you may wish to use an embOS release library or a stack check library.

Therefore you have to select or replace the embOS library in your project or target:

- If your selected library is already available in your project, just select the appropriate configuration.
- To add a library, you may add the library to the existing Lib group. Exclude all other libraries from your build, delete unused libraries or remove them from the configuration.
- Check and set the appropriate `OS_LIBMODE_*` define as preprocessor option and/ or modify the `OS_Config.h` file accordingly.

2.4 Select another CPU

embOS contains CPU-specific code for various CPUs. Manufacturer- and CPU-specific sample start workspaces and projects are located in the subfolders of the `BoardSupport\` folder. To select a CPU which is already supported, just select the appropriate workspace from a CPU-specific folder.

If your CPU is currently not supported, examine all `RTOSInit.c` files in the CPU-specific subfolders and select one which almost fits your CPU. You may have to modify `OS_InitHW()`, `OS_COM_Init()`, the interrupt service routines for embOS system timer tick and communication to embOSView and the low level initialization.

Chapter 3

Libraries

This chapter includes CPU-specific information such as CPU-modes and available libraries.

3.1 Naming conventions for prebuilt libraries for IAR Embedded Workbench 6.x and 7.x

embOS is shipped with different pre-built libraries with different combinations of features.

The libraries are named following the naming convention of the IAR runtime libraries:

`os<Architecture>_<CpuMode><ByteOrder><VFP_support><Interwork><LibMode>.a`

Parameter	Meaning	Values
`Architecture`	Specifies the instruction set architecture	4t: v4T 5t: v5T, v6 7a: v7A
`CpuMode`	Specifies the CPU mode	a: ARM t: Thumb / Thumb2
`ByteOrder`	Endianess	b: Big endian l: Little endian
`VFP_support`	Floating point support	_: Software floating-point support v: VFP with VFP parameters
`Interwork`	Specifies if interwork option is enabled	i: Enabled interworking _: Do not use interworking
`LibMode`	Specifies the library mode	xr: Extreme Release r: Release s: Stack check sp: Stack check + profiling d: Debug dp: Debug + profiling + Stack check dt: Debug + profiling + Stack check + trace dpl: Debug + profiling + Stack check built with low optimization level

Example

`os4t_tl_ir.a` is the embOS library for a project supporting an ARM architecture v4T core, using thumb mode, little-endian byte order, software floating-point support, compiled with the interwork option and release build configuration.

The Debug and Profiling library built with low optimization level may be used during development to show a more detailed call stack when using the CSpy plugin.

3.2 Naming conventions for prebuilt libraries for IAR Embedded Workbench version 8.x

embOS for ARM and IAR is shipped with different pre-built libraries with different combinations of features:

- Instruction set architecture - `Isa`
- CPU mode - `CpuMode`
- Byte order - `endianess`
- VFP support (all prebuilt libraries are built with software floating-point support.) - `VFP_support`
- Library mode - `LibMode` The libraries are named following the naming convention of the IAR runtime libraries:

`os<Isa>_<CpuMode><ByteOrder><VFP_support><LibMode>.a`

Parameter	Meaning	Values
`Isa`	Specifies the instruction set architecture	4t: v4T 5t: v5T, v6 7a: v7A
`CpuMode`	Specifies the CPU mode	a: ARM t: Thumb / Thumb2
`ByteOrder`	Endianess	b: Big endian l: Little endian
`VFP_support`	Floating point support	_: Software floating-point support v: VFP with VFP parameters
`LibMode`	Specifies the library mode	xr: Extreme Release r: Release s: Stack check sp: Stack check + profiling d: Debug dp: Debug + profiling + Stack check dt: Debug + profiling + Stack check + trace dpl: Debug + profiling + Stack check built with low optimization level

Example

`os4t_tl_r.a` is the embOS library for a project supporting an ARM architecture v4T core, using thumb mode, little-endian byte order, software floating-point support, compiled with release build configuration.

The Debug and Profiling library built with low optimization level may be used during development to show a more detailed call stack when using the CSpy plugin.

Chapter 4

CPU and compiler specifics

4.1 Standard system libraries

embOS for ARM and IAR may be used with IAR standard libraries.

If non thread-safe functions are used from different tasks, embOS functions may be used to encapsulate these functions and guarantee mutual exclusion.

The system libraries from the IAR Embedded Workbench V6 come with built-in hook functions, which enable thread safe calls of all system functions automatically when supported by the operating system.

embOS compiled for IAR Embedded Workbench is prepared to use these hook functions. Adding one or two source code modules which are delivered with embOS activates the automatic thread locking functionality of the new IAR DLib.

4.2 Thread-safe system libraries

Using embOS with C++ projects and file operations or just normal call of heap management functions may require thread-safe system libraries if these functions are called from different tasks. Thread-safe system libraries require some locking mechanism which is RTOS specific.

To activate thread safe system library functionality, one or two special source modules delivered with embOS have to be included in the project. How to use this locking mechanisms depends on the IAR compiler version

4.2.1 Thread safe system libraries with IAR compiler V6.10 to V6.30

To enable the automatic thread safe locking functions, the source module `xmtx.c` which is included in every CPU specific Setup folder of the embOS shipment has to be included in the project.

To support thread safe file i/o functionality, the source module `xmtx2.c` has to be added.

The embOS libraries compiled for and with the new IAR compiler / workbench V6 come with all code required to automatically handle the thread safe system libraries when the source module `xmtx.c` or `xmtx2.c` from the embOS shipment are included in the project.

Note that thread safe system library and file i/o support is required only, when non thread safe functions are called from multiple tasks, or thread local storage is used in multiple tasks.

4.2.2 Thread safe system libraries with IAR compiler V6.40 to V7.80

With IAR compiler version V6.4 or newer, the thread safe system library hook functions delivered with embOS are **not** automatically linked in, even if they are included in the project.

To enable the automatic thread safe locking functions, the project options for the linker have to be setup to replace the default locking functions from the system libraries by the functions delivered with embOS. To enable thread safe system library support, include the two files `xmtx.c` and `xmtx2.c` in the project.

Activate the checkbox “`Use command line options`” in the dialog `Project -> Options -> Linker ->Extra options` then, in the “`Command line options:`” field, add the following lines:

```
--redirect __iar_Locksyslock=__iar_Locksyslock_mtx
--redirect __iar_Unlocksyslock=__iar_Unlocksyslock_mtx
--redirect __iar_Lockfilelock=__iar_Lockfilelock_mtx
--redirect __iar_Unlockfilelock=__iar_Unlockfilelock_mtx
```



```
--keep     __iar_Locksyslock_mtx
```

4.2.3 Thread safe system libraries with IAR compiler V8.10 and newer

To enable the automatic thread safe locking functions, the source module xmtx.c which is included in every CPU specific Setup folder of the embOS shipment has to be included in the project and the function `OS_INIT_SYS_LOCKS()` must be called. Additionally the option "`Enable thread support in library`" must be set in `project settings->Library configuration`.

To support thread safe file i/o functionality, the source module `xmtx2.c` has to be added.

The embOS libraries compiled for and with the new IAR compiler / workbench V8 come with all code required to automatically handle the thread safe system libraries when the source module `xmtx.c` or `xmtx2.c` from the embOS shipment are included in the project.

Note that thread safe system library and file i/o support is required only, when non thread safe functions are called from multiple tasks, or thread local storage is used in multiple tasks.

4.3 Thread-Local Storage TLS

The dlib of EWARM V6 supports usage of thread-local storage. Several library objects and functions need local variables which have to be unique to a thread. Thread-local storage will be required when these functions are called from multiple threads.

embOS for EWARM V6 is prepared to support the tread-local storage, but does not use it per default. This has the advantage of no additional overhead as long as thread-local storage is not needed by the application. The embOS implementation of thread-local storage allows activation of TLS sepa- rately for every task.

Only tasks that call functions using TLS need to activate the TLS by calling an initial- ization function when the task is started.

The IAR runtime environment allocates the TLS memory on the heap. Using TLS with multiple tasks shall therefore use thread safe system library functionality which is automatically enabled when the `xmtx.c` module from the embOS distribution is added to the project.

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.

4.3.1 OS_TASK_SetContextExtensionTLS()

Description

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

Prototype

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

Additional information

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

Thread safe system library calls are automatically enabled when the source module `xmtx.c` which is delivered with embOS in the BSP Setup folders is included in the project. The function is available in embOS for EWARM6 only.

Example

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

```
#include "RTOS.h"

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

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



```
  }
}

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

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


4.3.2 OS_TASK_SetContextExtensionTLSVFP()

Description

`OS_TASK_SetContextExtensionTLSVFP()` has to be called as first function in a task, when thread locale storage and thread safe floating point processor support is needed in the task.

prototype

```
void OS_TASK_SetContextExtensionTLSVFP (void);
```

Additional information

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

To ensure thread safe heap management, the thread safe system library functionallity should also be enabled when using TLS.

The task specific TLS management is generated as embOS task extesion 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_TASK_SetContextExtension()` is impossible.

Chapter 5

Stacks

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

5.1 Task stack

Each task uses its individual stack. The stack pointer is initialized and set every time a task is activated by the scheduler. The stack-size required for a task is the sum of the stack-size of all routines, plus a basic stack size, plus size used by exceptions.

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

For the ARM7/ARM9 and Cortex-A/R cores, the minimum basic task stack size is about 68 bytes. Because any function call uses some amount of stack and every exception also pushes at least 32 bytes onto the current stack, the task stack size has to be large enough to handle one exception too. We recommend at least 256 bytes stack as a start.

5.2 System stack

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

The size of the system stack can be changed by modifying the `CSTACK` definition in your linker command file: `*.xcl` (IAR V4.x) or `*.icf` (IAR V5.x and later). We recommend a minimum stack size of 256 bytes for the `CSTACK`.

5.3 Interrupt stack

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

As a result, only registers mentioned above are saved onto the IRQ stack. For the interrupt routine itself, the supervisor stack is used. The size of the interrupt stack can be changed by modifying `IRQ_STACK` in your `*.xcl` (IAR V4.x) or `*.icf` (IAR V5.x or later) linker command file. We recommend at least 128 bytes.

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

5.4 Stack specifics

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

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

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

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

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

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

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

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

Chapter 6

Interrupts

6.1 What happens when an interrupt occurs?

- The CPU-core receives an interrupt request.
- As soon as the interrupts are enabled, the interrupt is executed.
- The CPU switches to the Interrupt stack.
- The CPU saves PC and flags in registers `LR_irq` and `SPSR_irq`.
- The CPU jumps to the vector address `0x18`, or offset `0x18` in the vector table, and continues execution from there.
- embOS `IRQ_Handler()`: save scratch registers.
- embOS `IRQ_Handler()`: save `LR_irq` and `SPSR_irq`.
- embOS `IRQ_Handler()`: switch to supervisor mode.
- embOS `IRQ_Handler()`: execute `OS_irq_handler()` (defined in `RTOSINIT_*.C`).
- embOS `OS_irq_handler()`: check for interrupt source and execute timer interrupt, serial communication or user ISR.
- embOS `IRQ_Handler()`: switch to IRQ mode.
- embOS `IRQ_Handler()`: restore `LR_irq` and `SPSR_irq`.
- embOS `IRQ_Handler()`: pop scratch registers.
- Return from interrupt. When using an ARM derivate with vectored interrupt controller, ensure that `IRQ_Handler()` is called from every interrupt. The interrupt vector itself may then be examined by the C-level interrupt handler in `RTOSInit*.c`.

6.2 Defining interrupt handlers in C

Interrupt handlers called from the embOS interrupt handler in `RTOSInit*.c` are just normal C-functions which do not take parameters and do not return any value. The default C interrupt handler `OS_irq_handler()` in `RTOSInit*.c` first calls `OS_INT_Enter()` or `OS_INT_EnterNestable()` to inform embOS that interrupt code is running. Then this handler examines the source of interrupt and calls the related interrupt handler function. Finally, the default interrupt handler `OS_irq_handler()` in `RTOSInit*.c` calls `OS_INT_Leave()` or `OS_INT_LeaveNestable()` and returns to the primary interrupt handler `IRQ_Handler()`. Depending on the interrupting source, it may be required to reset the interrupt pending condition of the related peripherals.

Example

Simple interrupt routine:

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

6.3 Interrupt handling without vectored interrupt controller

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

The reaction to an interrupt is as follows:

- embOS `IRQ_Handler()` is called.
- `IRQ_Handler()` saves registers and switches to supervisor mode.
- `IRQ_Handler()` calls `OS_irq_handler()`.
- `OS_irq_handler()` informs embOS that interrupt code is running by a call of `OS_INT_Enter()` and then calls `OS_USER_irq_func()` which has to handle all interrupt sources of the application.

- `OS_irq_handler()` checks whether embOS timer interrupt has to be handled.
- `OS_irq_handler()` checks whether embOS UART interrupts for communication with embOSView have to be handled.
- `OS_irq_handler()` informs embOS that interrupt handling ended by a call of `OS_INT_Leave()` and returns to `IRQ_Handler()`.
- `IRQ_Handler()` restores registers and performs a return from interrupt.

Example

Simple interrupt routine:

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

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

6.4 Interrupt handling with vectored interrupt controller

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

You should not program the interrupt controller for IRQ handling directly. You should use the functions delivered with embOS.

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

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

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

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

Function	Description
`OS_ARM_InstallISRHandler()`	Installs an interrupt handler
`OS_ARM_EnableISR()`	Enables a specific interrupt
`OS_ARM_DisableISR()`	Disables a specific interrupt
`OS_ARM_ISRSetPrio()`	Sets the priority of a specific interrupt
`OS_ARM_AssignISRSource()`	Assigns a hardware interrupt channel to an interrupt vector
`OS_ARM_EnableISRSource()`	Enables an interrupt channel of a VIC type interrupt controller
`OS_ARM_DisableISRSource()`	Disables an interrupt channel of a VIC type interrupt controller

6.4.1 OS_ARM_InstallISRHandler()

Description

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

Prototype

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

Parameters

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

Return Value

`OS_ISR_HANDLER *`: The address of the previously installed interrupt function, which was installed at the addressed vector number before.

Additional Information

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

6.4.2 OS_ARM_EnableISR()

Description

`OS_ARM_EnableISR()` is used to enable interrupt acceptance of a specific interrupt source in a vectored interrupt controller.

Prototype

```
void OS_ARM_EnableISR ( int ISRIndex );
```

Parameters

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

Additional Information

This function just enables the interrupt inside the interrupt controller. It does not enable the interrupt of any peripherals. This has to be done elsewhere.

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

6.4.3 OS_ARM_DisableISR()

Description

OS_ARM_DisableISR() is used to disable interrupt acceptance of a specific interrupt source in a vectored interrupt controller which is not of the VIC type.

Prototype

```
void OS_ARM_DisableISR ( int ISRIndex );
```

Parameters

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

Additional Information

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

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

6.4.4 OS_ARM_ISRSetPrio()

Description

OS_ARM_ISRSetPrio() is used to set or modify the priority of a specific interrupt source by programming the interrupt controller.

Prototype

```
int OS_ARM_ISRSetPrio ( int ISRIndex,
                        int Prio );
```

Parameters

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

Return Value

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

Additional Information

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

6.4.5 OS_ARM_AssignISRSource()

Description

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

Prototype

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

Parameters

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

Additional Information

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

6.4.6 OS_ARM_EnableISRSource()

Description

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

Prototype

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

Parameters

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

Additional Information

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

6.4.7 OS_ARM_DisableISRSource()

Description

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

Prototype

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

Parameters

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

Additional Information

This function disables a hardware interrupt input of a VIC-type interrupt controller. It cannot be used for other types of vectored interrupt controllers. The hardware interrupt channel number of specific peripherals depends on specific CPU derivates 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 priotity
OS_ARM_EnableISR(UART_ID);                             // Enable UART interrupt
/* Install UART interrupt handler with VIC type interrupt controller*/
OS_ARM_InstallISRHandler(UART_INT_INDEX, &COM_ISR);    // UART interrupt vector
OS_ARM_AssignISRSource(UART_INT_INDEX, UART_INT_SOURCE);
OS_ARM_EnableISR(UART_INT_INDEX);
  // Enable UART interrupt vector
OS_ARM_EnableISRSource(UART_INT_SOURCE);
  // Enable UART interrupt source
```


6.5 Interrupt-stack switching

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

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

6.6 Fast Interrupt (FIQ)

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

Note the following:

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

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

Chapter 7

MMU and cache support

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

7.1 MMU and cache support with embOS

embOS comes with functions to support the MMU and cache of ARMv4, ARMv5 and ARMv7A CPUs which allow virtual-to-physical address mapping with sections of one MByte and cache control. The MMU requires a translation table which can be located in any data area, RAM or ROM, but has to be aligned at a 16Kbyte boundary.

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

7.2 MMU and cache handling for ARM CPUs

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

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

7.2.1 OS_ARM_MMU_InitTT()

Description

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

Prototype

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

Parameters

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

Additional information

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

7.2.2 OS_ARM_MMU_AddTTEntries()

Description

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

Prototype

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

Parameters

Parameter	Description
pTranslationTable	Points to the base address of the translation table.
CacheMode	Specifies the cache operating mode which should be used for the selected area. May be one of the following modes: ARMv4/ARMv5: OS_ARM_CACHEMODE_NC_NB: non-cacheable, non-bufferable OS_ARM_CACHEMODE_C_NB: cacheable, non-bufferable OS_ARM_CACHEMODE_NC_B: non-cacheable, bufferable OS_ARM_CACHEMODE_C_B: cacheable, bufferable ARMv7A: OS_ARM_CACHEMODE_STRONGLY_ORDERED: Strongly ordered OS_ARM_CACHEMODE_SHAREABLE_DEVICE: Shareable Device OS_ARM_CACHEMODE_WRITE_THROUGH: Outer and Inner Write-Through, no Write-Allocate OS_ARM_CACHEMODE_WRITE_BACK_NO_ALLOC: Outer and Inner Write-Back, no Write-Allocate OS_ARM_CACHEMODE_NON_CACHEABLE: Outer and Inner Non-cacheable OS_ARM_CACHEMODE_WRITE_BACK_ALLOC: Outer and Inner Write-Back, Write-Allocate OS_ARM_CACHEMODE_NON_SHAREABLE_DEVICE: Non-shareable Device
VIndex	Virtual address index, which is the start address of the virtual memory address range with MBytes resolution. VIndex = (virtual address >> 20)
PIndex	Physical address index, which is the start address of the physical memory area range with MBytes resolution. PIndex = (physical address >> 20)
NumEntries	Specifies the size of the memory area in MBytes.

Additional information

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

7.2.3 OS_ARM_MMU_Enable()

Description

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

Prototype

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

Parameters

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

Additional information

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

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

7.2.4 OS_ARM_MMU_GetVirtualAddr()

Description

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

Prototype

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

Parameters

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

Return value

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

Additional information

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

7.2.5 OS_ARM_MMU_v2p()

Description

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

Prototype

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

Parameters

Parameter	Description
pVAddr	Pointer which represents the virtual address.

Return value

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

Additional information

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

7.2.6 OS_ARM_ICACHE_Enable()

Description

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

Prototype

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

Additional information

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

7.2.7 OS_ARM_ICACHE_Invalidate()

Description

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

Prototype

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


7.2.8 OS_ARM_DCACHE_Enable()

Description

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

Prototype

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

Additional information

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

7.2.9 OS_ARM_DCACHE_Invalidate()

Description

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

Prototype

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


7.2.10 OS_ARM_DCACHE_Clean()

Description

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

Prototype

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

Additional information

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

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

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

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

7.2.11 OS_ARM_DCACHE_CleanRange()

Description

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

Prototype

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

Parameters

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

Additional information

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

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

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

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

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

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

7.2.12 OS_ARM_DCACHE_InvalidateRange()

Description

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

Prototype

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

Parameters

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

Additional information

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

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

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

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

7.2.13 OS_ARM_CACHE_Sync()

Description

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

Prototype

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

Additional information

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

7.2.14 OS_ARM_AddL2Cache()

Description

OS_ARM_AddL2Cache() is used to add.

Prototype

```
void OS_ARM_v7A_AddL2Cache(const OS_ARM_L2CACHE_API* pCacheAPI,
                           void*                     pParam);
```

Parameters

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

Additional information

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

Example

```
#define L2CACHE_BASE_ADDR 0x3FFFF000u

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


7.3 MMU and cache handling for ARM720 CPUs

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

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

7.3.1 OS_ARM720_MMU_InitTT()

Description

`OS_ARM720_MMU_InitTT()` is used to initialize an MMU translation table which is located in RAM. The table is filled with zero, thus all entries are marked invalid ini- tially.

Prototype

```
void OS_ARM720_MMU_InitTT (unsigned int* pTranslationTable);
```

Parameters

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

Additional information

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

7.3.2 OS_ARM720_MMU_AddTTEntries()

Description

OS_ARM720_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_ARM720_MMU_AddTTEntries(unsigned int* pTranslationTable,
                                unsigned int  CacheMode,
                                unsigned int  VIndex,
                                unsigned int  PIndex,
                                unsigned int  NumEntries);
```

Parameters

Parameter	Description
pTranslationTable	Points to the base address of the translation table.
CacheMode	Specifies the cache operating mode which should be used for the selected area. May be one of the following modes: 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
VIndex	Virtual address index, which is the start address of the virtual memory address range with MBytes resolution. VIndex = (virtual address >> 20)
PIndex	Physical address index, which is the start address of the physical memory area range with MBytes resolution. PIndex = (physical address >> 20)
NumEntries	Specifies the size of the memory area in MBytes.

Additional information

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

7.3.3 OS_ARM720_MMU_Enable()

Description

`OS_ARM720_MMU_Enable()` is used to enable the MMU which will then perform the address mapping.

Prototype

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

Parameters

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

Additional information

As soon as the function was called, the address translation is active. The MMU table has to be setup before calling `OS_ARM720_MMU_Enable()`.

7.3.4 OS_ARM720_MMU_GetVirtualAddr()

Description

OS_ARM720_MMU_GetVirtualAddr() is used to translate a physical address into a virtual address with specified cache mode.

Prototype

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

Parameters

Parameter	Description
PAddr	The physical address as unsigned long.
CacheMode	The cache mode of the requested virtual address May be one of the defined cache modes: OS_ARM_CACHEMODE_NC_NB OS_ARM_CACHEMODE_C_NB OS_ARM_CACHEMODE_NC_B OS_ARM_CACHEMODE_C_B OS_ARM_CACHEMODE_ANY

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 exaplme an LCD controller accesses the diplay buffer, or an Ethernet MAC access a transferbuffer, the CPU has to write the data uncached into this memory, or has to clean the cache after writing.
The function OS_ARM720_MMU_GetVirtualAddress() can be used to find the address for uncached access.
The MMU table has to be setup before the function is called.

7.3.5 OS_ARM720_MMU_v2p()

Description

OS_ARM720_MMU_v2p() is used to translate a virtual address into a physical address.

Prototype

```
unsigned long OS_ARM720_MMU_v2p (void* pVAddr);
```

Parameters

Parameter	Description
pVAddr	Pointer which represents the virtual address.

Return value

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

Additional information

The function can be used to examine the physical addresss of memory. The CPU normally operates with virtual addresses which may differ from the physical address of the memory. When a peripheral or DMA has to be programmed to access the same memory, the peripheral has to be programmed to access the physical memory. The function OS_ARM720_MMU_v2p() can be used to find the physical address of a memory area. The MMU table has to be setup before the function is called.

7.3.6 OS_ARM720_CACHE_Enable()

Description

`OS_ARM720_CACHE_Enable()` is used to enable the data cache of the CPU.

Prototype

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

Additional information

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

7.3.7 OS_ARM720_CACHE_CleanRange()

Description

OS_ARM720_CACHE_CleanRange() is used to clean a range in the data cache memory to ensure that the data is written from the data cache into the memory.

Prototype

```
void OS_ARM720_CACHE_CleanRange(void*        p,
                                unsigned int NumEntries);
```

Parameters

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

Additional information

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

Before starting a DMA transfer, a call of OS_ARM720_CACHE_CleanRange() ensures, that the data is transferred from the data cache into the memory and the write buff- ers 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 = (NumBytes / 32) * (10 [CPU clock cycles] + Memory write time).

The real time depends on the content of the cache. If data in the cache is marked as dirty, the cache line has to be written to memory. The memory write time depends on the memory BUS clock and memory speed. If data has to be written to memory, the required cycles for this memory operation has to be added to the 10 CPU clock cycles for every 32 bytes to be cleaned.

7.3.8 OS_ARM720_CACHE_InvalidateRange()

Description

OS_ARM720_CACHE_InvalidateRange() is used to invalidate a memory area in the data cache. Invalidating means, mark all entries in the specified area as invalid. Invalidation forces re-reading the data from memory into the cache, when the speci- fied area is accessed again.

Prototype

```
void OS_ARM720_CACHE_InvalidateRange(void*        p,
                                     unsigned int NumBytes);
```

Parameters

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

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 com- plete cache lines can be invalidated.

Therefore, it is 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.

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 = (NumBytes / 32) * 10 [CPU clock cycles].

7.4 MMU and cache handling program sample

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

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

```
/*********************************************************************
*
* MMU and cache configuration
*
* The MMU translation table has to be aligned to 16KB boundary
* and has to be located in uninitialized data area
*/
#pragma data_alignment=16384
__no_init static unsigned int _TranslationTable [0x1000]; // OS_INTERWORK int
__low_level_init(void) {
  //
  // Init MMU and caches
  //
  OS_ARM_MMU_InitTT (&_TranslationTable[0]);
  //
  // Internal SRAM, the first MB remapped to 0,cacheable, bufferable
  //
  OS_ARM_MMU_AddTTEntries ( &_TranslationTable[0],
                            OS_ARM_CACHEMODE_C_B,
                            0x000, 0x200, 0x001);
  //
  // Internal SRAM, original address, NON cachable, NON bufferable
  //
  OS_ARM_MMU_AddTTEntries ( &_TranslationTable[0],
                            OS_ARM_CACHEMODE_NC_NB,
                            0x200, 0x200, 0x001);
  OS_ARM_MMU_Enable (&_TranslationTable[0]);
  OS_ARM_ICACHE_Enable();
  OS_ARM_DCACHE_Enable();
  return 1;
}
```

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

Chapter 8

VFP and NEON support

8.1 Vector Floating Point and NEON support

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

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

8.1.1 OS_TASK_SetContextExtensionVFP()

Description

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

Prototype

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

Additional information

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

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

8.1.2 OS_TASK_SetContextExtensionNEON()

Description

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

Prototype

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

Additional information

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

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

8.1.3 Using VFP/NEON in interrupt service routines

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

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

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

8.1.3.1 OS_VFP_Save() / OS_NEON_Save()

Description

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

Prototype

```
void OS_VFP_Save (void void);

void OS_NEON_Save (void void);
```

Additional information

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

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

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

8.1.3.2 OS_VFP_Restore() / OS_NEON_Restore()

Description

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

Prototype

```
void OS_VFP_Restore (void void);

void OS_NEON_Restore (void void);
```

Additional information

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

Chapter 9

Technical data

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

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 minimum ROM requirement for the kernel itself is about 1.700 bytes.

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

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

