embOS

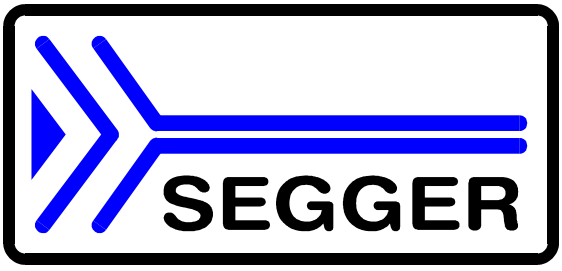
Real Time Operating System

CPU & Compiler specifics for

ARM core

using IAR Embedded Workbench

Document Rev. 13



A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Contents

1. About this document

This guide describes how to use ***embOS*** Real Time Operating System for the ARM series of microcontrollers using *IAR Embedded Workbench*.

1.1. How to use this manual

This manual describes all CPU and compiler specifics of ***embOS*** using ARM based controllers with *IAR Embedded Workbench*. Before actually using ***embOS***, you should read or at least glance through this manual in order to become familiar with the software.
Chapter 2 gives you a step-by-step introduction, how to install and use ***embOS*** for ARM using *IAR Embedded Workbench.* If you have no experience using ***embOS***, you should follow this introduction, because it is the easiest way to learn how to use ***embOS*** in your application.
Most of the other chapters in this document are intended to provide you with important detailed information about functionality and fine-tuning of ***embOS*** for the ARM based controllers using *IAR Embedded Workbench*.

2. Using *embOS* with IAR Embedded Workbench

The following chapter describes how to start with and use ***embOS*** for ARM and IAR compiler. You should follow these steps to become familiar with ***embOS*** for ARM and *IAR Embedded Workbench*

2.1. Installation

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

In order to install it, proceed as follows:

If you received a CD, copy the entire contents to your hard-drive into any folder of your choice. When copying, 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.

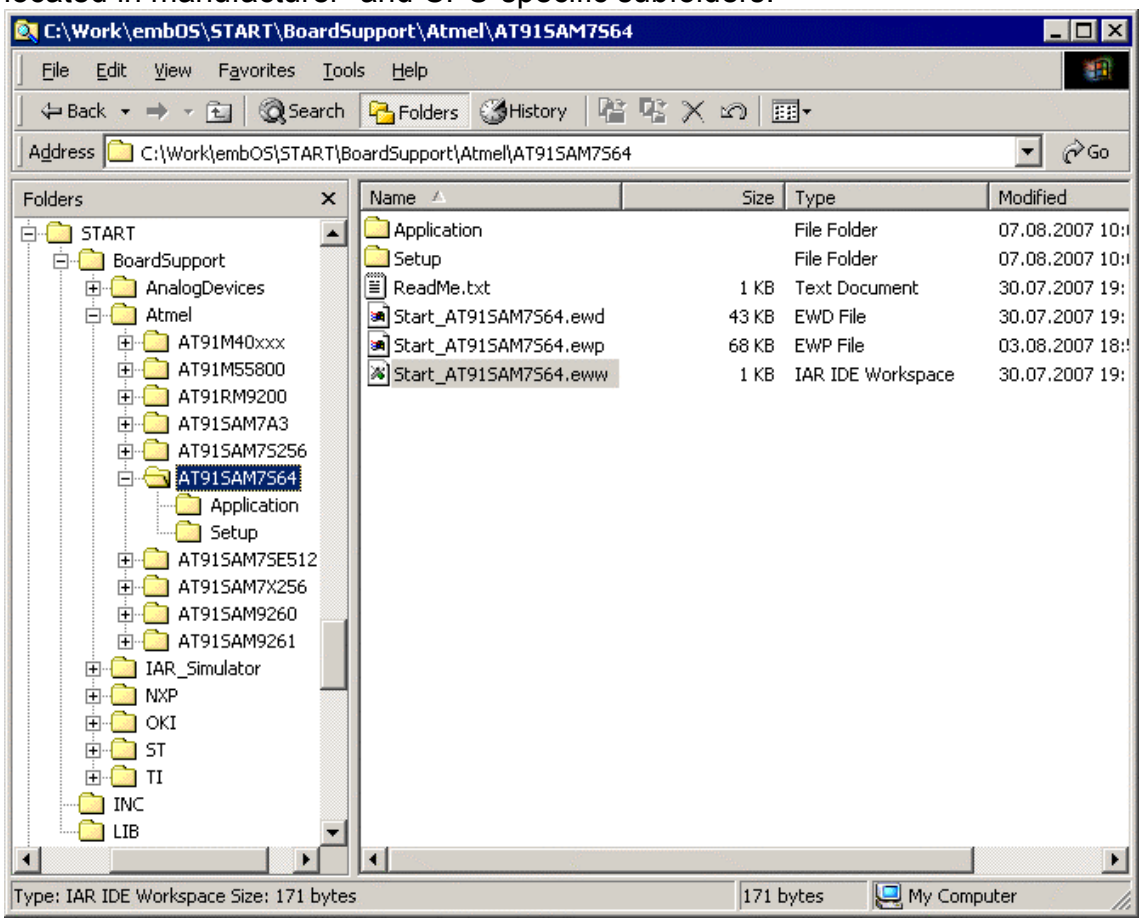
Assuming that you are using *IAR Embedded Workbench* project manager to develop your application, no further installation steps are required. You will find a prepared sample start application, which you should use and modify to write your application. So follow the instructions of the next chapter 'First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

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

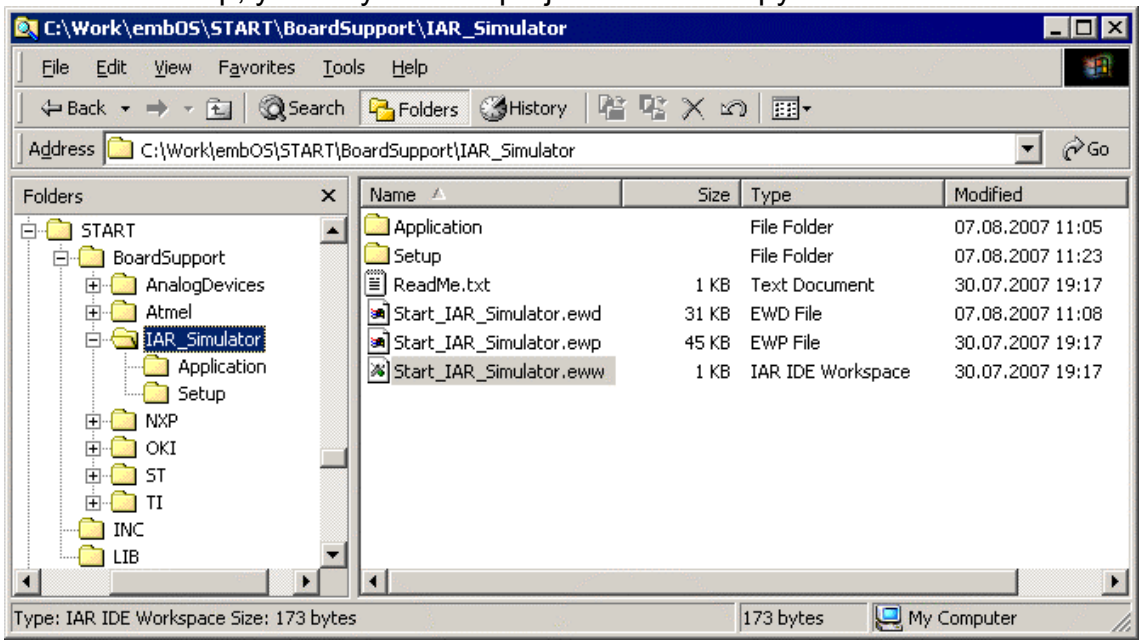
If for some reason you will not work with the project manager, you should:
Copy either all or only the library-file that you need to your work-directory. This has the advantage that when you switch to an updated version of ***embOS*** later in a project, you do not affect older projects that use ***embOS*** also.
embOS does in no way rely on *IAR Embedded Workbench* project manager, it may be used without the project manager using batch files or a make utility without any problem.

2.2. First steps

After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You 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:



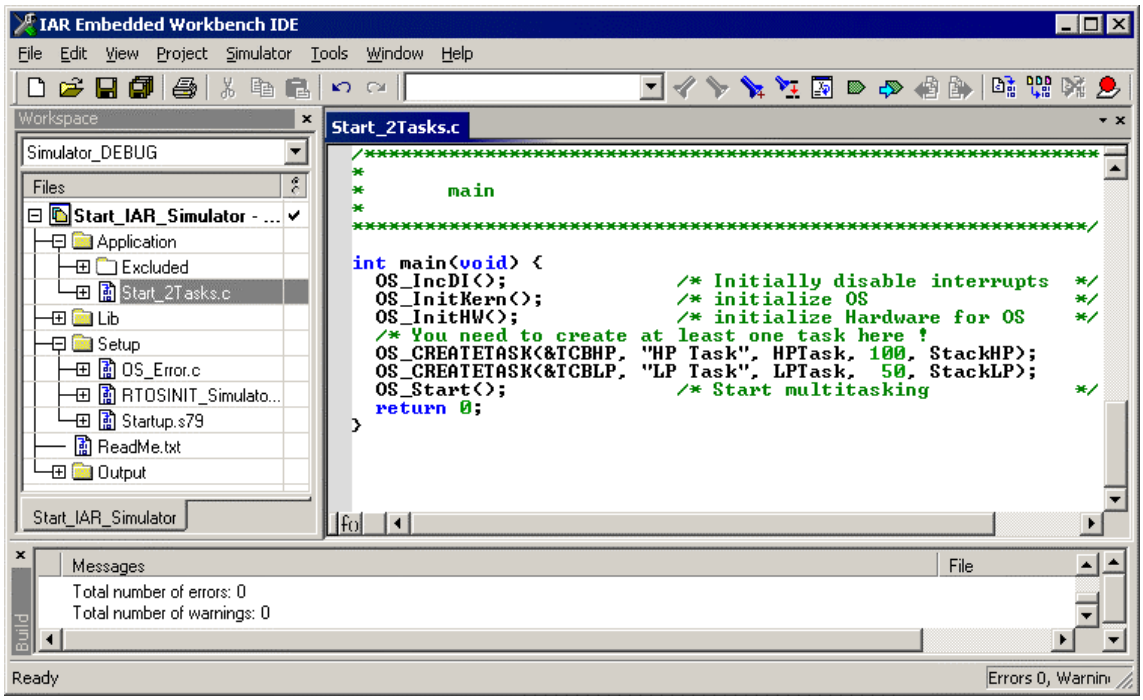
The screenshot above shows the start workspace and project for an ATMEL AT91SAM7S64,
which is located under “Start\BoardSupport\Atmel\AT91SAM7S64”.
For the first step, you may use the project for IAR CSpy simulator:



To get your first multitasking 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\BoardSuppord\IAR_Simulator\Start_IAR_Simulator.eww” with the *IAR Embedded Workbench* project manager (e.g. by double clicking it)
- Build the start project

Your screen should look like follows:



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..

2.3. The sample application Main.c

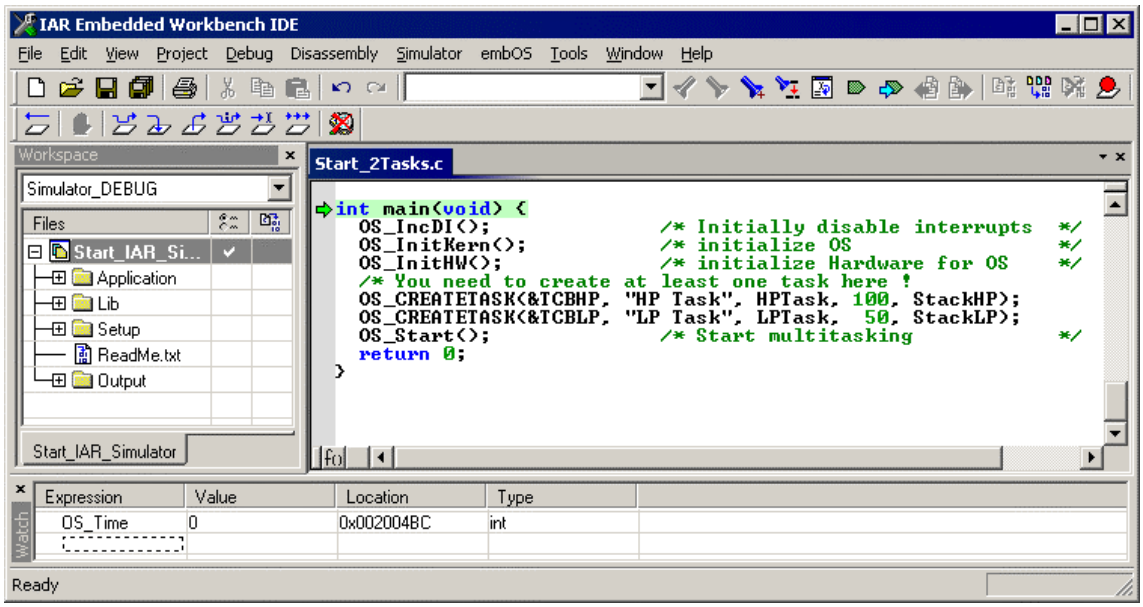
The following is a printout of the sample application main.c. It is a good starting-point for your application. (Please note that the file actually shipped with your port of *embOS* may look slightly different from this one)
What happens is easy to see:
After initialization of *embOS;* two tasks are created and started
The two tasks are activated and execute until they run into the delay, then suspend for the specified time and continue execution.

```
/*********************************************************************
*               SEGGER MICROCONTROLLER SYSTEME GmbH                  *
*       Solutions for real time microcontroller applications         *
**********************************************************************
File    : Main.c
Purpose : Skeleton program for OS
--------  END-OF-HEADER  -------------------------------------------
*/

#include "RTOS.h"

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

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

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

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

int 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;
}
```

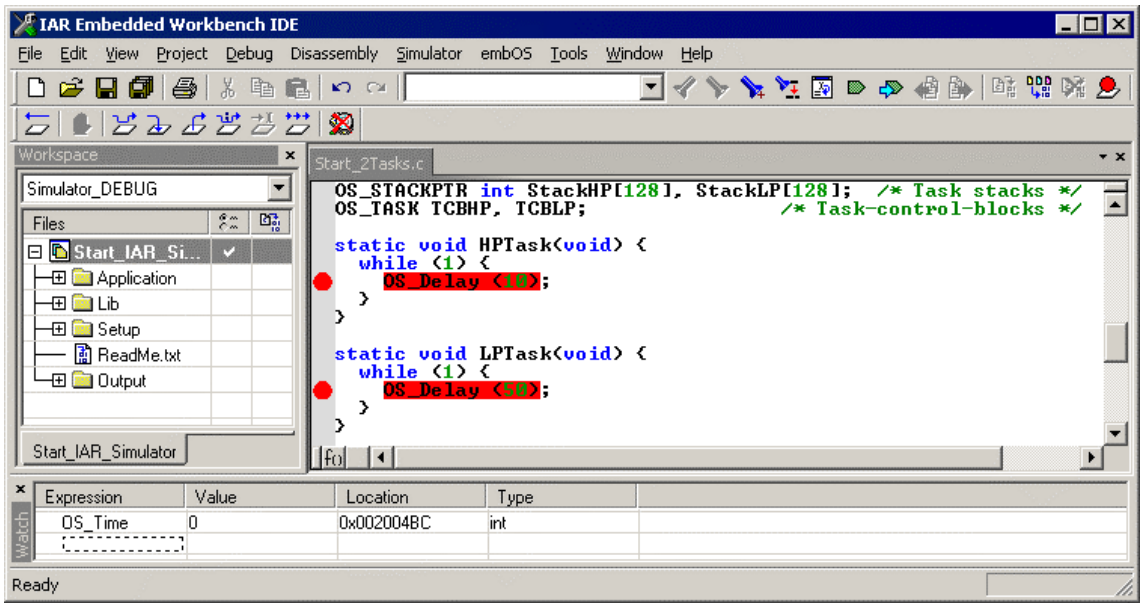
2.4. Stepping through the sample application using C-SPY

When starting the debugger, you will usually see the main function (very similar to the screenshot below). In some debuggers, you may look at the startup code and have to set a breakpoint at main. Now you can step through the program.
`OS_IncDI()` initially disables interrupts.
`OS_InitKern()` is part of the *embOS* library; you can therefore only step into it in disassembly mode. It initializes the relevant OS-Variables. Because of the previous call of `OS_IncDI()`, interrupts are not enabled during execution of `OS_InitKern()`.

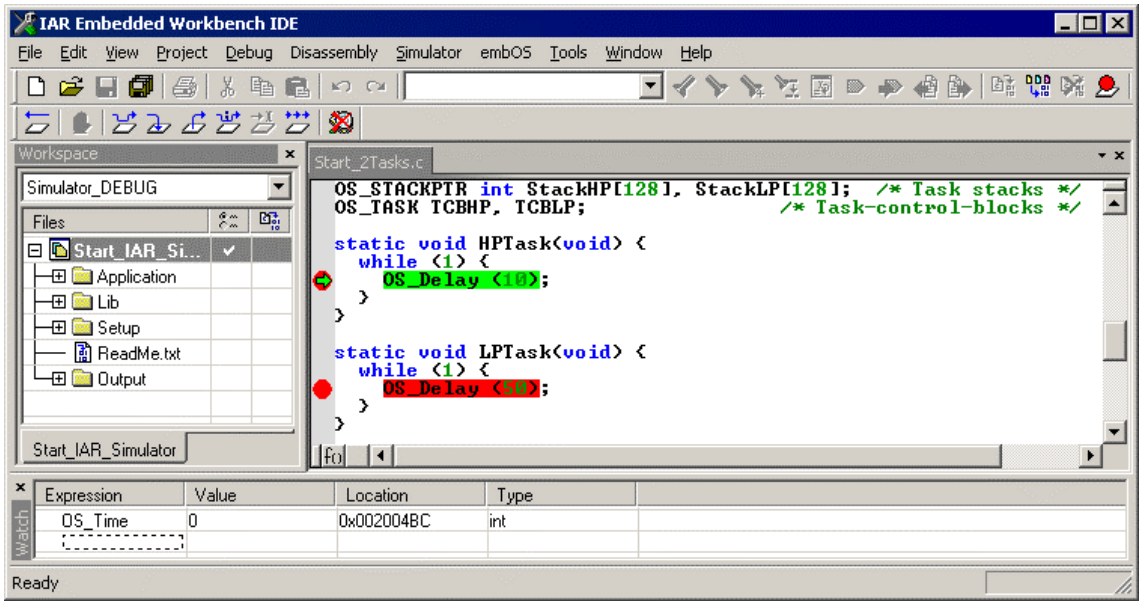
OS_InitHW() is part of RTOSInit_*.c and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the timer-tick-interrupt for ***embOS.*** Step through it to see what is done.
OS_Start() should be the last line in main, since it starts multitasking and does not return.



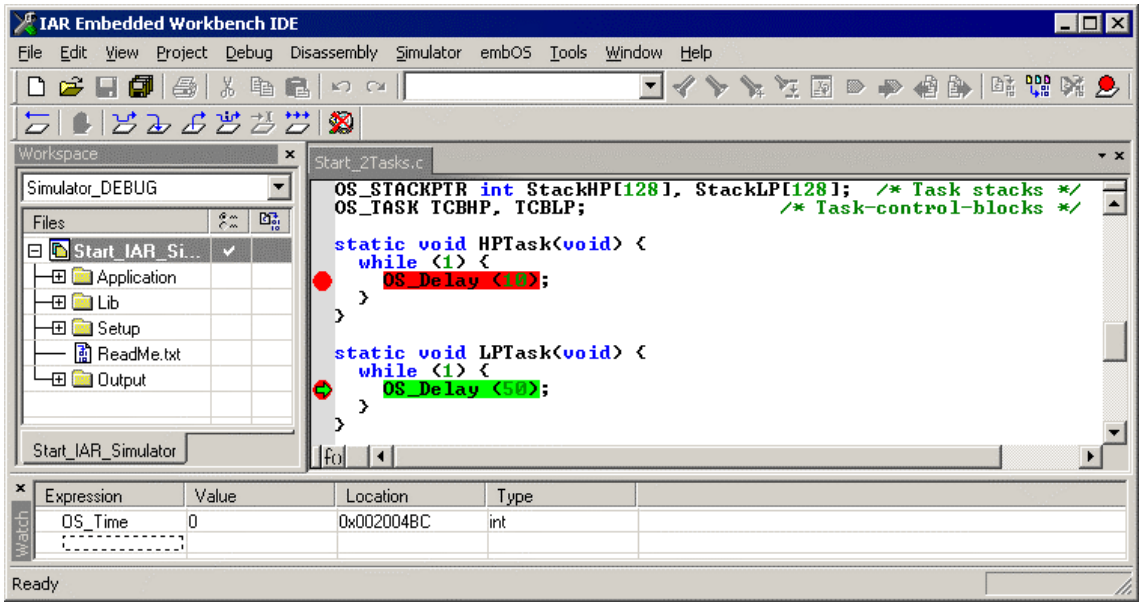
Before you step into OS_Start(), you should set two break points in the two tasks as shown below.



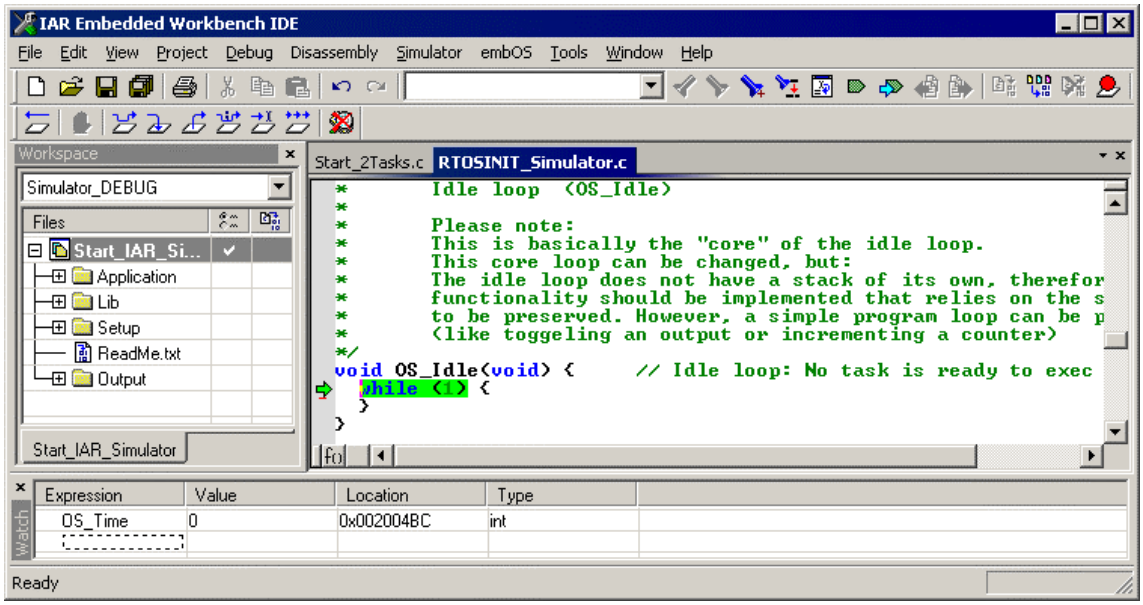
As OS_Start() is part of the ***embOS*** library, you can step through it in disassembly mode only. You may press GO, step over OS_Start(), or step into OS_Start() in disassembly mode until you reach the highest priority task.



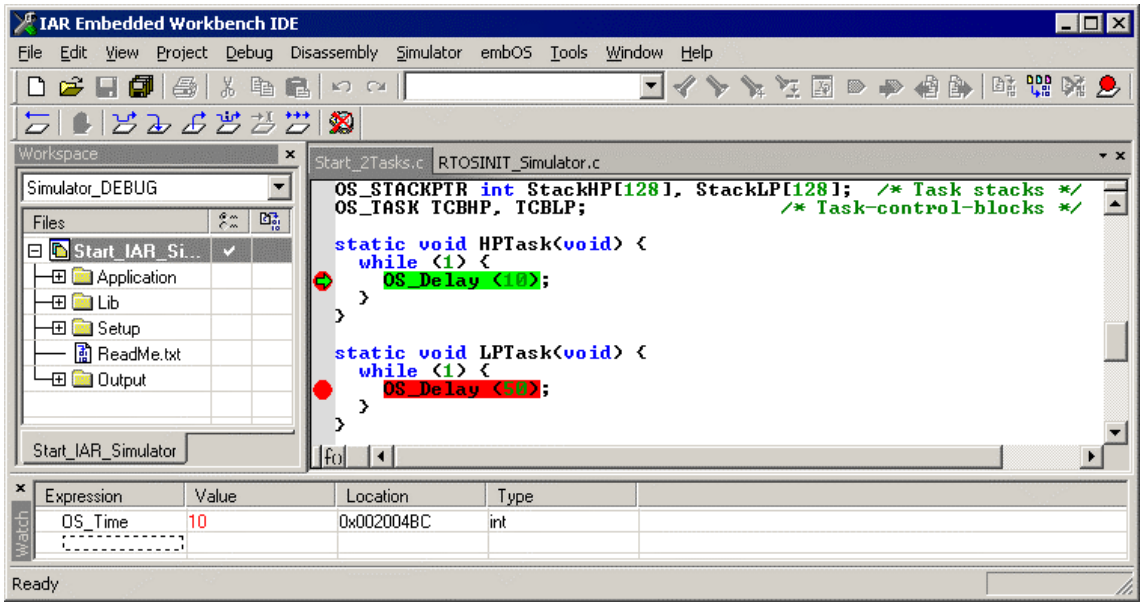
If you continue stepping, you will arrive in the task with lower priority:



Continuing to step through the program, there is no other task ready for execution. ***embOS*** will therefore start the idle-loop, which is an endless loop which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).
You will arrive there when you step into the `OS_Delay()` function in disassembly mode. `OS_Idle()` is part of `RTOSInit*.c`. You may also set a breakpoint there before you step over the delay in Task1.



If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay.
As can be seen by the value of **embOS** timer variable `OS_Time`, shown in the watch window, Task0 continues operation after expiration of the 10 ms delay.



3. Build your own application

To build your own application, you should always start with a copy of a sample start workspace and project. Therefore copy the entire folder “Start” from your *embOS* distribution into a working folder of your choice and then modify the start projects there. This has the advantage, that all necessary files are included and all settings for the project are already done.

3.1. Required files for an *embOS* application

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 sub folder 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*\” - subfolder.
  It contains the hardware dependent initialization code for the specific CPU, the *embOS* timer and optional UART for embOSView.
- One *embOS* **library** from the Lib\ subfolder
- **OS_Error.c** from one target specific “BoardSupport*\” subfolder. The error handler is used if any *embOS* library other than the Release build library is used in your project.

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 CPU running in supervisor or system mode.
Your main() function has to initialize *embOS* by call of `OS_InitKern()` and `OS_InitHW()` prior any other *embOS* functions except `OS_IncDI()` are called.
You should then modify or replace the main.c source file in the subfolder src\.

3.2. 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:

- If your wished library is already contained in your project, just select the appropriate configuration.
- To add a library, you may add the library to the “Lib” group of your project and exclude all other libraries from build. You may alternatively remove unused libraries from the “Lib” group.
- Check and set the appropriate OS_LIBMODE_* define which you would like to use for debug and release builds and modify the OS_Config.h file accordingly.

3.3. Select an other CPU

embOS for ARM and IAR compiler 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 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()`, the interrupt service routines for ***embOS*** timer tick and communication to embOSView and `__lowLevel_init()`.

4. ARM specifics

4.1. CPU modes

embOS supports nearly all memory and code model combinations that IAR' s C-Compiler supports.
embOS comes with libraries for interworking and non interworking mode.

4.2. Available libraries

embOS for ARM for IAR compiler is shipped in 2 versions for two different IAR compilers.
Both versions use different start projects and libraries.

4.2.1. Libraries for IAR compiler version 4.42A

embOS for IAR compiler version 4.42A comes with 112 different libraries, one for each CPU mode / CPU core / endianess / library mode combination.
The libraries are named as follows:

os<v>tp<m>n<e>8<LibMode>.r79

Parameter	Meaning	Values	
v	Specifies the CPU variant	4:	core type4: ARM 7/9
		5:	core type5: ARM9E
m	Specifies the CPU mode	a:	ARM mode
		t:	THUMB mode
i	Specifies interworking mode	i:	interworking allowed
		n:	NON interworking
e	Endianess	l:	Little
		b:	Big
LibMode	Library mode	XR:	Extreme Release
		R:	Release
		S:	Stack check
		D:	Debug
		SP:	Stack check + profiling
		DP:	Debug + profiling
		DT:	Debug + profiling + trace

Example:
os4tptnnl8dp.r79 is the library for a project using ARM 7/9 core, THUMB mode, non interworking, little endian mode and debug and profiling features of ***embOS***.
The naming convention of ***embOS*** libraries follows the one used for IAR system libraries. In case of doubt which ***embOS*** library to use, just examine which IAR system library is used in your project, and select the ***embOS*** library with the similar name.

4.2.2. Libraries for IAR compiler version 5.10 and above

embOS for IAR compiler version 5.1x comes with 112 different libraries, one for each CPU mode / CPU core / endianess / library mode combination.
The library names follow the naming convention used by IAR for the system libraries.

The libraries are named as follows:

os<cpu>_<mode><endianess>_<interwork><LibMode>.a

Parameter	Meaning	Values	
cpu	Specifies the CPU variant	4t:	core type4: ARM 7/9
		5t:	core type5: ARM9E
mode	Specifies the CPU mode	a:	ARM mode
		t:	THUMB mode
endianess	Endianess of CPU	l:	Little
		b:	Big
interwork	Specifies interworking mode	i:	interworking allowed
		_:	NON interworking
LibMode	Library mode	XR:	Extreme Release
		R:	Release
		S:	Stack check
		D:	Debug
		SP:	Stack check + profiling
		DP:	Debug + profiling
		DP:	Debug + profiling + trace

Example:
os4t_al__xr.a is the library for a project using ARM 7/9 core, ARM mode, little endian, non interworking mode and extreme release features of ***embOS***.

5. Compiler specifics

5.1. 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.
How to use this locking mechanisms depends on the IAR compiler version

- **IAR compiler V4.41A or newer:**
  System libraries delivered with compiler version 4.41A or newer already contain a locking mechanism which may be used with *embOS* if required. The locking has to be initialized once by a call of `OS_INIT_SYS_LOCKS()`.
- **IAR compiler version 4.30A to 4.40A:**
  System libraries which come with the compiler version 4.30A to 4.40A are not thread safe per default. The current version of *embOS* was built for IAR compiler version 4.41A or newer. Support for thread safe system libraries of older IAR system libraries is not supported.

5.2. OS_INIT_SYS_LOCKS(), thread safe system locking

To use the automatic locking mechanism which is implemented in the system libraries which came with compiler version 4.41A or newer, it has to be initialized once by calling
OS_INIT_SYS_LOCKS()
This function should be called from main() during the initialization, directly after the call of `OS_InitKern()`. After calling `OS_INIT_SYS_LOCKS()` all functions which are normally not thread safe can be used from any task without any precaution. `OS_INIT_SYS_LOCKS()` does not work when the system libraries which came with compiler version 4.40A or older are used.

6. Stacks

6.1. Task stack for ARM 7 and ARM 9

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 stack-size 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.

6.2. System stack for ARM 7 and ARM 9

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 definition of `CSTACK` in your *.XCL file.

6.3. Interrupt stack for ARM 7 and ARM 9

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. For the interrupt routine itself, the supervisor stack is used.
The size of the interrupt stack can be changed by modifying the definition of `IRQ_STACK` in your *.XCL file.
Every interrupt requires 28bytes on the interrupt stack.
The maximum interrupt stack size required by the application can be calculated as “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.

6.4. Stack specifics of the ARM 7 and ARM 9 family

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 start up 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 start up 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.

7. Interrupts

7.1. What happens when an interrupt occurs?

- The CPU-core receives an interrupt request
- As soon as the interrupts are enabled, the interrupt is executed
- the CPU switches to the Interrupt stack
- the CPU saves PC and flags in registers LR_irq and SPSR_irq
- the CPU jumps to the vector address 0x18 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, please ensure that `IRQ_Handler()` is called from every interrupt. The interrupt vector itself may then be examined by the “C”-level interrupt handler `OS_irq_handler()` in RTOSInit*.c.

7.2. 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 `IRQ_Handler()` which is part of the ***embOS*** library.
Depending on the interrupting source, it may be required to reset the interrupt pending condition of the related peripherals.

Example

"Simple" interrupt-routine

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

7.3. Interrupt handling without vectored interrupt controller

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

The reaction to an interrupt is as follows:

- The ***embOS*** interrupt handler `IRQ_Handler()` is called.
- `IRQ_Handler()` saves registers and switches to supervisor mode.
- `IRQ_Handler()` calls `OS_irq_handler()`.
- `OS_irq_handler()` informs ***embOS*** that interrupt code is running by a call of `OS_EnterInterrupt()` and then calls `OS_USER_irq_func()` which has to handle all interrupt sources of the application.
- `OS_irq_handler()` checks whether ***embOS*** timer interrupt has to be handled.
- `OS_irq_handler()` checks whether ***embOS*** UART interrupts for communication with embOSView have to be handled.
- `OS_irq_handler()` informs ***embOS*** that interrupt handling ended by a call of `OS_LeaveInterrupt()` and returns to `IRQ_Handler()`.
- `IRQ_Handler()` restores registers and performs a return from interrupt.

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

Example

"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_ISR1();         // handle interrupt
  }
}
```

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

7.4. Interrupt handling with vectored interrupt controller

For ARM 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 the embOS interrupt handler `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:

- The *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 *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 `IRQ_Handler()`.
- `OS_Handler()` 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:

7.4.1. OS_ARM_InstallISRHandler(): Install an interrupt handler

Description

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

Prototype

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

Parameter	Meaning
`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 previous installed interrupt function, which was installed at the addressed vector number before.

Add. information

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

7.4.2. OS_ARM_EnableISR(): Enable 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	Meaning
`ISRIndex`	Index of the interrupt source which should be enabled.

Return value

NONE.

Add. 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.

7.4.3. 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	Meaning
`ISRIndex`	Index of the interrupt source which should be disabled.

Return value

NONE.

Add. 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.

7.4.4. 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	Meaning
`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().

Add. 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.

7.4.5. OS_ARM_AssignISRSource(): Assign a hardware interrupt channel to an interrupt vector

Description

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

Prototype

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

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

Return value

None.

Add. 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.

7.4.6. OS_ARM_EnableISRSource(): Enable an interrupt channel of a VIC type interrupt controller

Description

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

Prototype

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

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

Return value

None.

Add. 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.

7.4.7. OS_ARM_DisableISRSource(): Disable an interrupt channel of a VIC type interrupt controller

Description

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

Prototype

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

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

Return value

None.

Add. 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 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 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.
```

7.5. 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.asm only.

7.6. 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 an FIQ stack is initialized during startup and an interrupt vector for FIQ handling is included in your application.

8. MMU and cache support

embOS comes with functions to support the MMU and cache of ARM9 and ARM720 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 required.

8.1. MMU and cache handling for ARM9 CPUs

ARM9 CPUs with MMU and cache have separate data and instruction caches. *embOS* delivers the following functions to setup and handle the MMU and caches.

8.1.1. 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_MMU_InitTT(unsigned int* pTranslationTable);
```

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

Return value

Void, none.

Add. information

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

8.1.2. 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 TranslationTable,
    unsigned int CacheMode,
    unsigned int VIndex,
    unsigned int PIndex,
    unsigned int NumEntries);
```

Parameter	Meaning
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 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.

Return value

Void, none.

Add. information

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

8.1.3. 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	Meaning
`pTranslationTable`	Points to the base address of the translation table.

Return value

Void, none.

Add. information

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

8.1.4. 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);
```

Return value

Void, none.

Add. information

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

8.1.5. 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);
```

Return value

Void, none.

Add. 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 or write back cache. Most CPUs will have implemented a write back cache.

8.1.6. 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	Meaning
`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.

Return value

Void, none.

Add. 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.

8.1.7. 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. Which results 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	Meaning
`p`	Points to the base address of the memory area that should be invalidated.
`NumBytes`	Number of bytes which have to be invalidated.

Return value

Void, none.

Add. 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.
As soon as the function was called, the instruction cache is active.
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.

8.2. 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.

8.2.1. OS_ARM720_MMU_InitTT(): Initialize the MMU translation table

Description

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

Prototype

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

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

Return value

Void, none.

Add. information

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

8.2.2. OS_ARM720_MMU_AddTTEntries(): Add address entries to the table

Description

`OS_ARM_MMU720_AddTTEntries ()` is used to add entries to the MMU address translation table. The start address of the virtual address, physical address, area size and cache modes are passed as parameter.

Prototype

```
void OS_ARM720_MMU_AddTTEntries(
    unsigned int TranslationTable,
    unsigned int CacheMode,
    unsigned int VIndex,
    unsigned int PIndex,
    unsigned int NumEntries);
```

Parameter	Meaning
`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`
	`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.

Return value

Void, none.

Add. information

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

8.2.3. OS_ARM720_MMU_Enable(): Enable the MMU

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 TranslationTable);
```

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

Return value

Void, none.

Add. 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()`.

8.2.4. OS_ARM720_CACHE_Enable(): Enable unified cache

Description

`OS_ARM720_CACHE_Enable()` is used to enable the unified cache of an ARM720 CPU, enabling caching of data and instructions.

Prototype

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

Return value

Void, none.

Add. 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.

8.2.5. OS_ARM720_CACHE_CleanRange(): Clean cache

Description

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

Prototype

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

Parameter	Meaning
`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.

Return value

Void, none.

Add. information

Cleaning the cache is needed, when data should be transferred by a DMA or other BUS master that does not use the data cache.
When the CPU writes data into a cacheable area, the data might not be written into the memory immediately. When then a DMA cycle is started to transfer the data from memory to any other location or peripheral, the wrong data will be written.
Before starting a DMA transfer, a call of `OS_ARM720_CACHE_CleanRange()` ensures, that the data is transferred from the data cache into the memory and the write buffers are drained.

8.2.6. OS_ARM720_CACHE_InvalidateRange(): Invalidate the cache

Description

`OS_ARM720_DCACHE_InvalidateRange()` is used to invalidate a memory area in the data cache. Invalidating means, mark all entries in the specified area as invalid. Which results in re-reading the data from memory into the cache, when the specified area is accessed again.

Prototype

```
OS_ARM720_DCACHE_ InvalidateRange(void* p,
                                  unsigned int NumBytes);
```

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

Return value

Void, none.

Add. 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 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.
As soon as the function was called, the instruction cache is active.
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.

8.3. MMU and cache handling program sample

The MMU und cache handling has to be set up before the data segments are initialized. Otherwise a virtual address mapping would not work.
The IAR startup code calls the `__low_level_init()` function before sections are initialized.
It is a good idea to initialize memory access, the MMU table and the cache control during `__low_level_init()`.
The following sample is an excerpt from one `__low_level_init()` function which is part af an RTOSInit.c file:

```
/*********************************************************************
*
*       MMU and cache configuration
*
*       The MMU translation table has to be aligned to 16KB boundary
*       and has to be located in uninitialized data area
*/
#pragma data_alignment=16384
__no_init static unsigned int _TranslationTable [0x1000];  //


OS_INTERWORK int __low_level_init(void) {
  //
  //  Initialize SDRAM
  //
  _InitSDRAM();
  //
  // Init MMU and caches
  //
  OS_ARM_MMU_InitTT      (&_TranslationTable[0]);
  //
  // SDRAM, the first MB remapped to 0 to map vectors to correct address,
  //cacheable, bufferable
  OS_ARM_MMU_AddTTEntries(&_TranslationTable[0],
                          OS_ARM_CACHEMODE_C_B,
                          0x000, 0x200, 0x001);
   // Internal SRAM, original address, NON cachable, NON bufferable
  OS_ARM_MMU_AddTTEntries(&_TranslationTable[0],
                          OS_ARM_CACHEMODE_NC_NB,
                          0x003, 0x003, 0x001);
  OS_ARM_MMU_Enable      (&_TranslationTable[0]);
  OS_ARM_ICACHE_Enable();
  OS_ARM_DCACHE_Enable();
  return 1;
}
```

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

9. STOP / WAIT Mode

In case your controller supports 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 function OS_Idle() in ***embOS*** module RTOSINIT.c.

10. Technical data

10.1. Memory requirements

These values are neither precise nor guaranteed but they give you a good idea of the memory-requirements. They vary depending on the current version of ***embOS***. In THUMB mode, the minimum ROM size required for the ***embOS*** kernel is about 1.500 bytes.
In the table below, you find the minimum RAM size for ***embOS*** resources. The sizes depend on selected ***embOS*** library mode; the table below is for a release build.

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

11. 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 IAR Embedded Workbench, contained in Manufacturer specific sub folders.
START\INC	`RTOS.H`	Include file for ***embOS***, to be included in every "C"-file using ***embOS*** -functions
START\LIB	`os*.r79`	***embOS*** libraries for compiler V4.4x
START\LIB	`os*.a`	***embOS*** libraries for compiler V5.x
START\SRC	`main.c`	Sample frame program to serve as a start
START\SRC	`OS_Error.c`	***embOS*** runtime error handler used in stack check or debug builds
START\CPU_*	`*.*`	CPU specific hardware routines for various CPUs.

Any additional files shipped serve as example.

12. Index