embOS

Real Time Operating System

CPU & Compiler specifics for

ARM core with

ARM RealView Developer Suite 3.0

Document Rev. 1

SEGGER

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 *ARM RealView Developer Suite*.

1.1. How to use this manual

This manual describes all CPU and compiler specifics for ***embOS*** using ARM based controllers with *ARM RealView Developer Suite*. 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*** using *ARM RealView Developer Suite*. 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 detailed information about functionality and fine-tuning of ***embOS*** for the ARM based controllers using *ARM RealView Developer Suite*.

2. Using *embOS* with ARM RealView Developer Suite

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 *ARM RealView Developer Suite* project manager to develop your application, no further installation steps are required. You will find prepared sample start projects an a sample application, which you should use and modify to write your own 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. Then copy the new `RTOS.h` file and CPU specific files also. 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 *ARM RealView Developer Suite* project manager, it may be used without the project manager using batch files or a make utility without any problem.

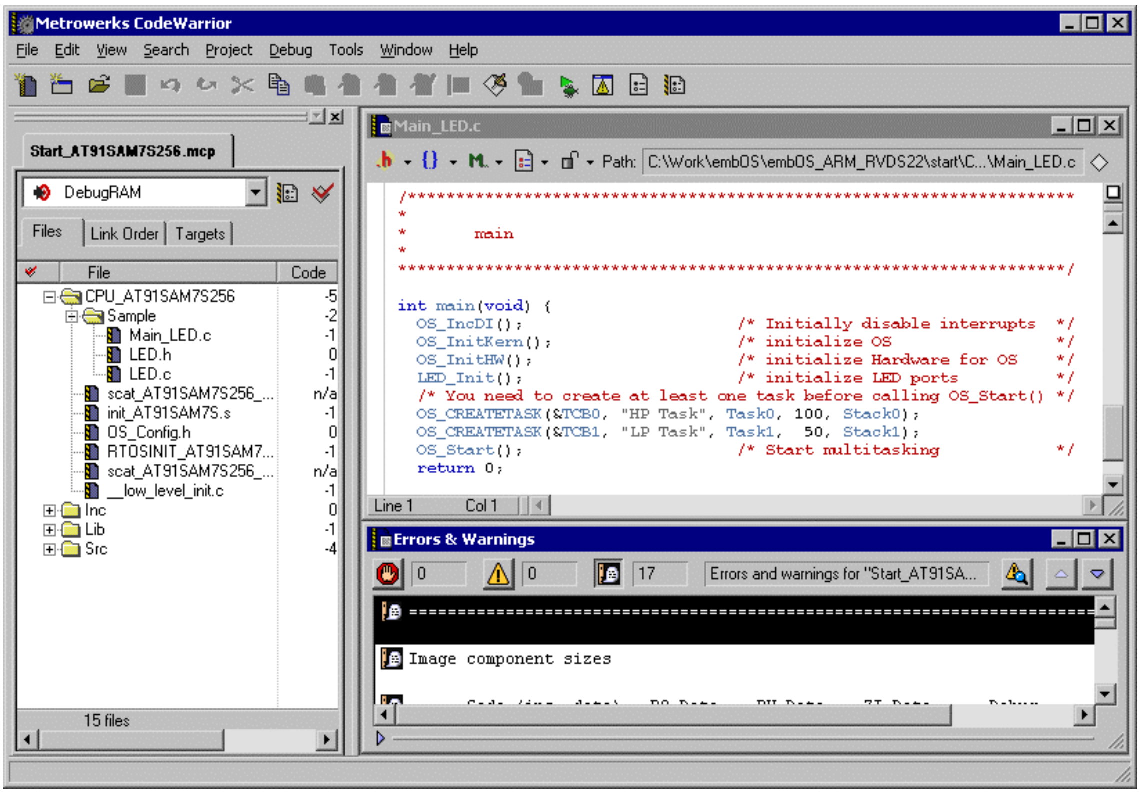
2.2. First steps

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

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.
- For first steps, open one of the projects with *ARM RealView Developer Suite* project manager (e.g. by double clicking it)
- Build the start project

Your screen should look like follows:



```
/**********************************************************************
*
*       main
*
**********************************************************************/

int main(void) {
  OS_IncDI();                      /* Initially disable interrupts  */
  OS_InitKern();                   /* initialize OS                 */
  OS_InitHW();                     /* initialize Hardware for OS    */
  LED_Init();                      /* initialize LED ports          */
  /* You need to create at least one task before calling OS_Start() */
  OS_CREATETASK(&TCB0, "HP Task", Task0, 100, Stack0);
  OS_CREATETASK(&TCB1, "LP Task", Task1,  50, Stack1);
  OS_Start();                      /* Start multitasking            */
  return 0;
```

```
=========================================================================
Image component sizes
```

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

2.3. The sample application Main.c

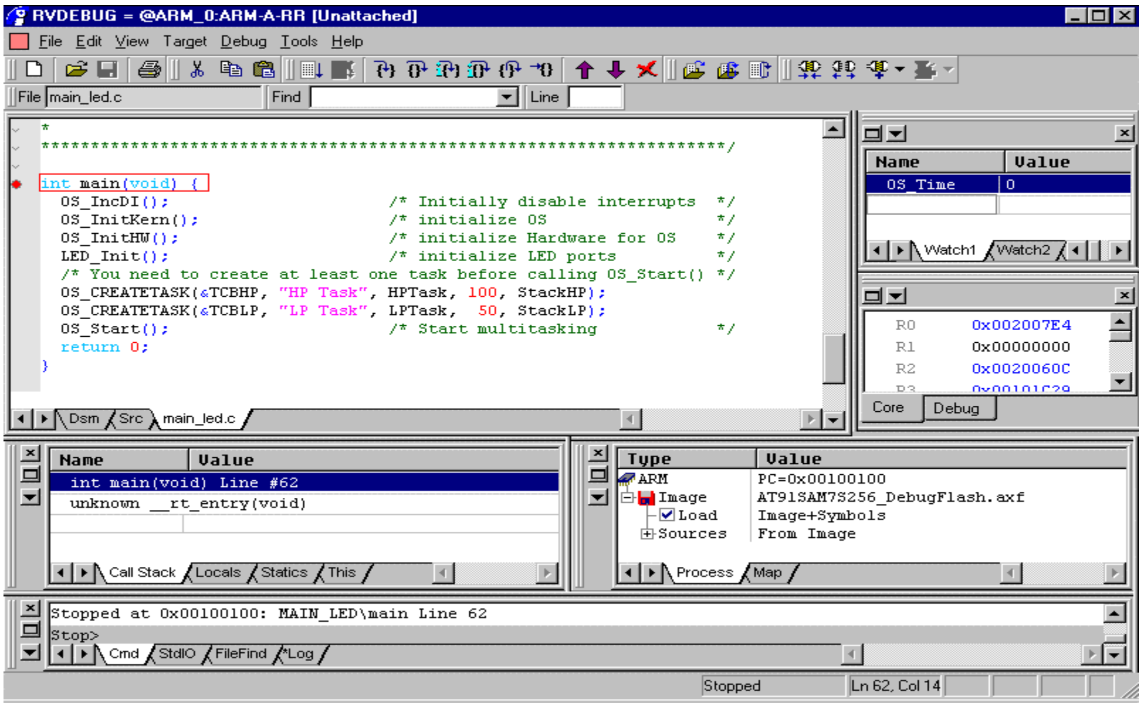
The following is a printout of the sample application main.c. It is a good starting-point for your application. (Please note that the file actually shipped with your port of ***embOS*** may look slightly different from this one)
What happens is easy to see:
After initialization of ***embOS;*** two tasks are created and started
The 2 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 embOS
--------  END-OF-HEADER  --------------------------------
*/

#include "RTOS.H"

OS_STACKPTR int Stack0[128], Stack1[128];            /* Task stacks */
OS_TASK TCB0, TCB1;                          /* Task-control-blocks */

void Task0(void) {
  while (1) {
    OS_Delay (10);
  }
}

void Task1(void) {
  while (1) {
    OS_Delay (50);
  }
}

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

int main(void) {
/*ensure no library functions that use semihosting SWIs are linked  */
#pragma import(__use_no_semihosting_swi)
  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(&TCB0, "HP Task", Task0, 100, Stack0);
  OS_CREATETASK(&TCB1, "LP Task", Task1,  50, Stack1);
  OS_SendString("Start project will start multitasking !\n");
  OS_Start();                      /* Start multitasking            */
  return 0;
} }
```

2.4. Stepping through the sample application Main.c using ARM Realview Debugger

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.


RVDEBUG = @ARM_0:ARM-A-RR [Unattached]
File Edit View Target Debug Tools Help
File main_led.c
Find
Line
Dsm Src main_led.c
Watch1 Watch2
Core Debug
Call Stack Locals Statics This
Process Map
Cmd StdIO FileFind Log

```
*
*********************************************************************/

int main(void) {
  OS_IncDI();                      /* Initially disable interrupts  */
  OS_InitKern();                   /* initialize OS                 */
  OS_InitHW();                     /* initialize Hardware for OS    */
  LED_Init();                      /* initialize LED ports          */
  /* You need to create at least one task before calling OS_Start() */
  OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
  OS_Start();                      /* Start multitasking            */
  return 0;
}
```

Name	Value
OS_Time	0

```
R0   0x002007E4
R1   0x00000000
R2   0x0020060C
R3   0x00101C29
```

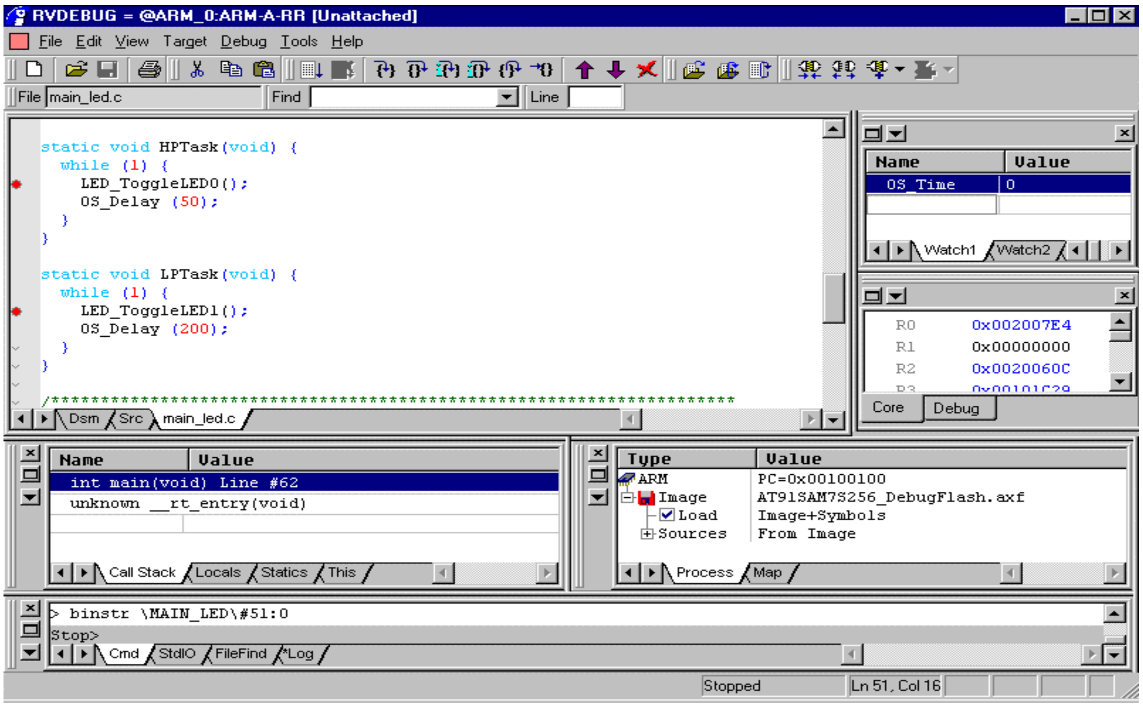
Name	Value
int main(void) Line #62	
unknown __rt_entry(void)	

Type	Value
ARM	PC=0x00100100
Image	AT91SAM7S256_DebugFlash.axf
Load	Image+Symbols
Sources	From Image

```
Stopped at 0x00100100: MAIN_LED\main Line 62
Stop>
```

Stopped   Ln 62, Col 14

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


RVDEBUG = @ARM_0:ARM-A-RR [Unattached]
File Edit View Target Debug Tools Help
File main_led.c
Find
Line
Dsm Src main_led.c
Watch1 Watch2
Core Debug
Call Stack Locals Statics This
Process Map
Cmd StdIO FileFind Log

```
static void HPTask(void) {
  while (1) {
    LED_ToggleLED0();
    OS_Delay (50);
  }
}

static void LPTask(void) {
  while (1) {
    LED_ToggleLED1();
    OS_Delay (200);
  }
}

/*********************************************************************
```

Name	Value
OS_Time	0

```
R0   0x002007E4
R1   0x00000000
R2   0x0020060C
R3   0x00101C29
```

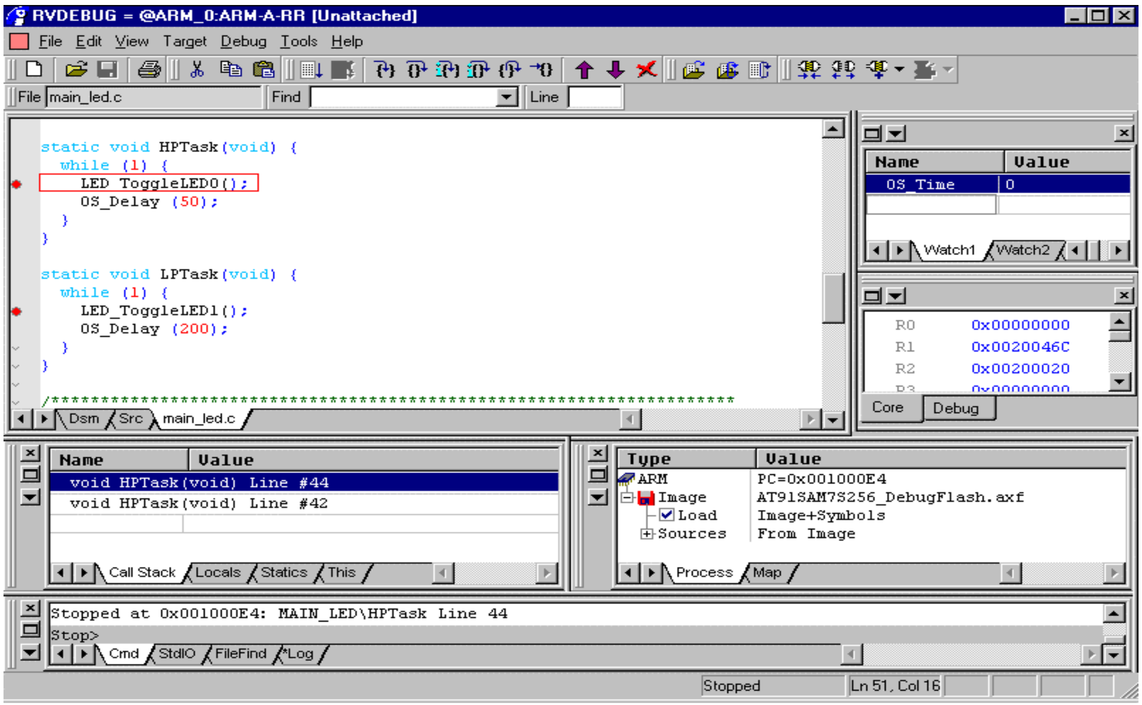
Name	Value
int main(void) Line #62	
unknown __rt_entry(void)	

Type	Value
ARM	PC=0x00100100
Image	AT91SAM7S256_DebugFlash.axf
Load	Image+Symbols
Sources	From Image

```
> binstr \MAIN_LED\#51:0
Stop>
```

Stopped   Ln 51, Col 16

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.



```
static void HPTask(void) {
  while (1) {
    LED_ToggleLED0();
    OS_Delay (50);
  }
}

static void LPTask(void) {
  while (1) {
    LED_ToggleLED1();
    OS_Delay (200);
  }
}
```

Name	Value
OS_Time	0

R0	0x00000000
R1	0x0020046C
R2	0x00200020

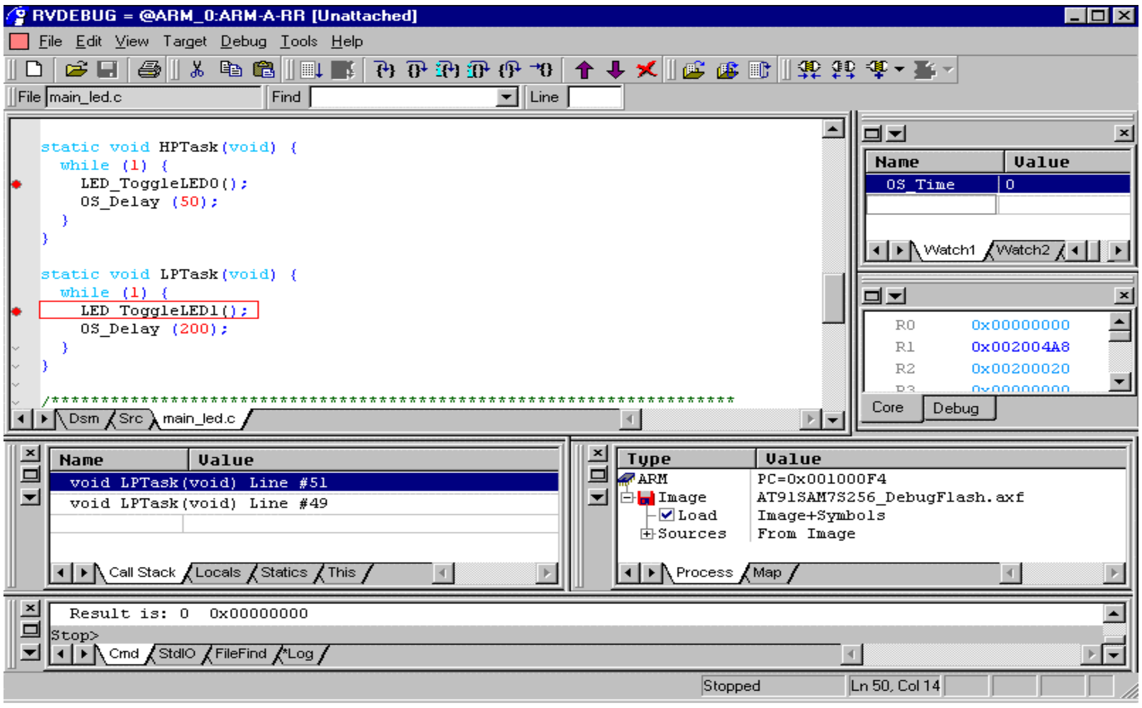
Name	Value
void HPTask(void) Line #44	
void HPTask(void) Line #42	

Type	Value
ARM	PC=0x001000E4
Image	AT91SAM7S256_DebugFlash.axf
Load	Image+Symbols
Sources	From Image

```
Stopped at 0x001000E4: MAIN_LED\HPTask Line 44
Stop>
```

RVDEBUG = @ARM_0:ARM-A-RR [Unattached]
File Edit View Target Debug Tools Help
File main_led.c Find Line
Dsm Src main_led.c
Watch1 Watch2
Core Debug
Call Stack Locals Statics This
Process Map
Cmd StdIO FileFind Log
Stopped Ln 51, Col 16

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



```
static void HPTask(void) {
  while (1) {
    LED_ToggleLED0();
    OS_Delay (50);
  }
}

static void LPTask(void) {
  while (1) {
    LED_ToggleLED1();
    OS_Delay (200);
  }
}
```

Name	Value
OS_Time	0

R0	0x00000000
R1	0x002004A8
R2	0x00200020

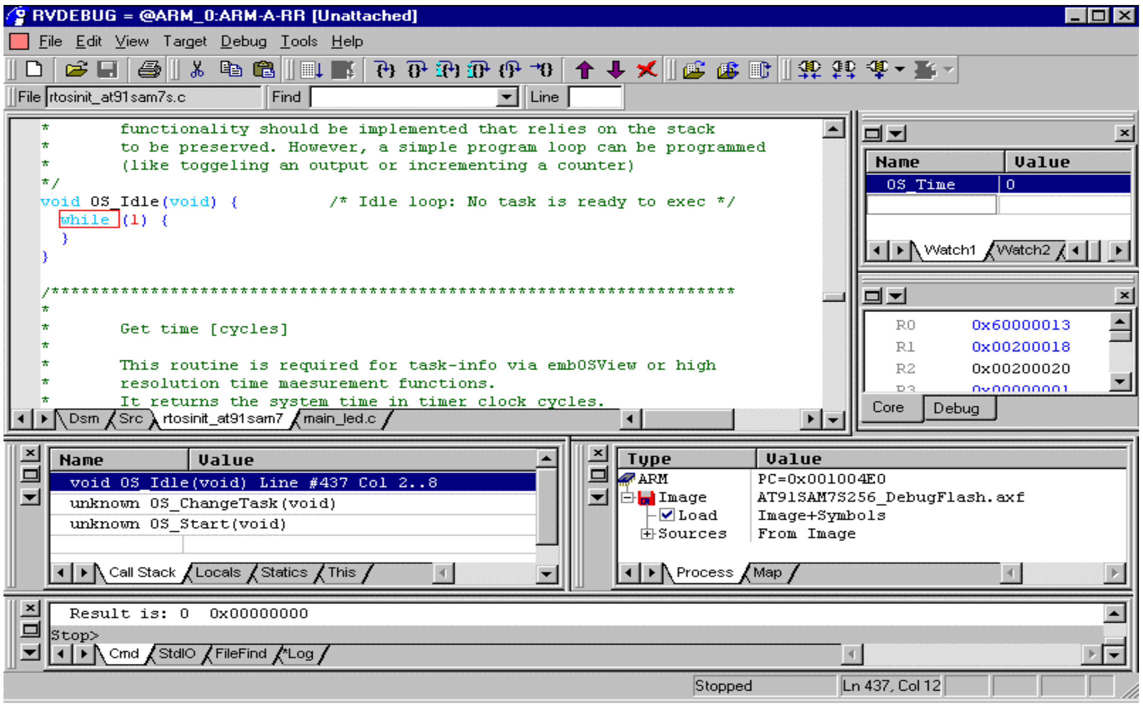
Name	Value
void LPTask(void) Line #51	
void LPTask(void) Line #49	

Type	Value
ARM	PC=0x001000F4
Image	AT91SAM7S256_DebugFlash.axf
Load	Image+Symbols
Sources	From Image

```
  Result is: 0   0x00000000
Stop>
```

RVDEBUG = @ARM_0:ARM-A-RR [Unattached]
File Edit View Target Debug Tools Help
File main_led.c Find Line
Dsm Src main_led.c
Watch1 Watch2
Core Debug
Call Stack Locals Statics This
Process Map
Cmd StdIO FileFind Log
Stopped Ln 50, Col 14

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.


RVDEBUG = @ARM_0:ARM-A-RR [Unattached]
File Edit View Target Debug Tools Help
File rtosinit_at91sam7s.c
Find
Line
Dsm Src rtosinit_at91sam7 main_led.c
Watch1 Watch2
Core Debug
Call Stack Locals Statics This
Process Map
Cmd StdIO FileFind Log
Stopped
Ln 437, Col 12

```
*        functionality should be implemented that relies on the stack
*        to be preserved. However, a simple program loop can be programmed
*        (like toggeling an output or incrementing a counter)
*/
void OS_Idle(void) {          /* Idle loop: No task is ready to exec */
  while (1) {
  }
}

/*********************************************************************
*
*        Get time [cycles]
*
*        This routine is required for task-info via embOSView or high
*        resolution time maesurement functions.
*        It returns the system time in timer clock cycles.
```

Name	Value
OS_Time	0

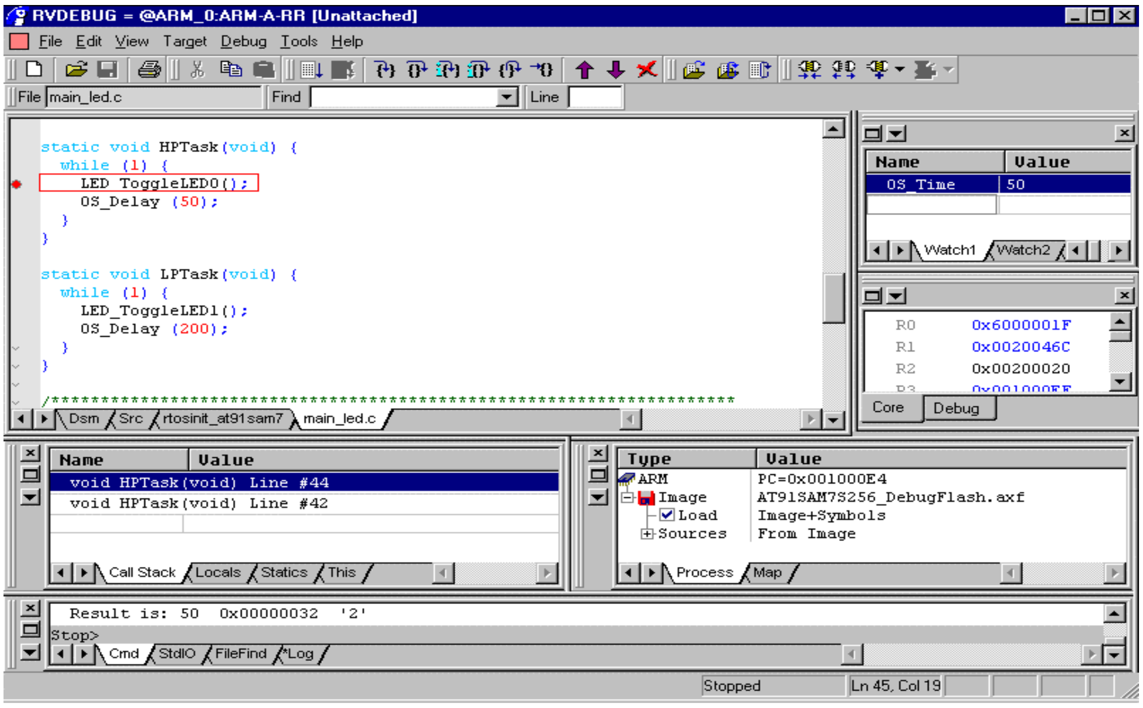
```
R0   0x60000013
R1   0x00200018
R2   0x00200020
```

Name	Value
void OS_Idle(void) Line #437 Col 2..8	
unknown OS_ChangeTask(void)	
unknown OS_Start(void)	

Type	Value
ARM	PC=0x001004E0
Image	AT91SAM7S256_DebugFlash.axf
✓ Load	Image+Symbols
Sources	From Image

```
Result is: 0  0x00000000
Stop>
```

If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay.
As can be seen by the value of ***embOS*** timer variable `OS_Time`, shown in the watch window, Task0 continues operation after expiration of the 50 ms delay.


RVDEBUG = @ARM_0:ARM-A-RR [Unattached]
File Edit View Target Debug Tools Help
File main_led.c
Find
Line
Dsm Src rtosinit_at91sam7 main_led.c
Watch1 Watch2
Core Debug
Call Stack Locals Statics This
Process Map
Cmd StdIO FileFind Log
Stopped
Ln 45, Col 19

```
static void HPTask(void) {
  while (1) {
    LED_ToggleLED0();
    OS_Delay (50);
  }
}

static void LPTask(void) {
  while (1) {
    LED_ToggleLED1();
    OS_Delay (200);
  }
}
/*********************************************************************
```

Name	Value
OS_Time	50

```
R0   0x6000001F
R1   0x0020046C
R2   0x00200020
```

Name	Value
void HPTask(void) Line #44	
void HPTask(void) Line #42	

Type	Value
ARM	PC=0x001000E4
Image	AT91SAM7S256_DebugFlash.axf
✓ Load	Image+Symbols
Sources	From Image

```
Result is: 50  0x00000032  '2'
Stop>
```

3. Build your own application

To build your own application, you should always start with a copy of the sample start project. Therefore copy the entire folder “Start” from your ***embOS*** distribution into a working folder of your choice and then modify the start project 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 CPU subfolder.
It contains hardware dependent initialization code for ***embOS*** timer and optional UART for embOSView.
- **UserIRQ.c** from one CPU subfolder.
It contains the application specific interrupt handler function which is called from ***embOS*** IRQ handler. This file may not be necessary for target CPUs with built in vectored interrupt controller.
- **init_*.s** from one of the CPU subfolder.
It contains the low level initialization of the hardware and setup of the various stack pointer.
- **scat_*.scat** from one of the CPU subfolder.
It contains the linker settings.
- One ***embOS*** **library** from the Lib\ subfolder
- **RTOSVect.s** from the Src\ subfolder.
It contains the low level interrupt handler entry for ARM CPUs running with ***embOS***.
- **OS_Error.c** from subfolder Src\ The error handler is used if any library other than Release build library is used in your project.
- **vector.s** from the Src\ subfolder.
This file contains the ARM vector table
- **retarget.c** from the Src\ subfolder.
Since you use your own linker file (scatter file) some functions need to be rewritten for the ARM RVDS runtime library. The delivered retarget.c file is already prepared for usage with ***embOS***. If you want to modify this file, please refer to the *ARM RealView Compilation tools* manuals.
- Additional low level init code may be required according to CPU.

When you decide to write your own startup code, please use one of the init_*.s as template. 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 a new embOSLib group to your project and add this library to the new group. Exclude all other library groups from build, delete unused Lib groups or remove them from the configuration.
- Check and set the appropriate OS_LIBMODE_* define as preprocessor option.

3.3. Select an other CPU

embOS for ARM and IAR compiler contains CPU specific code for various ARM CPUs. ***embOS*** comes with projects for target CPUs.
Check whether your CPU is supported by ***embOS***. CPU specific functions are located in the CPU_* subfolders of the start project folder.
To select a CPU which is already supported, just open the appropriate project.
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() and the interrupt service routines for ***embOS*** timer tick and communication to embOSView.

4. ARM specifics

4.1. CPU modes

embOS supports nearly all memory and code model combinations that ARM RVDK's C-Compiler supports.
embOS was compiled with interwork options. Therefore it is required to compile the projects with interwork option too.

4.2. Available libraries

embOS for ARM RealView developer Suite is shipped with 24 different libraries, one for each CPU mode / CPU core / endian mode and library type combination.
The libraries are named as follows:

OS<m><v><c><e><LibMode>.a

Parameter	Meaning	Values	
m	Specifies the CPU mode	A:	ARM mode
		T:	THUMB mode
v	Specifies the CPU variant	4:	core type4: ARM 7/9
e	Endian mode	L:	Little
		B:	Big
LibMode	Library mode	R:	Release
		S:	Stack check
		D:	Debug
		SP:	Stack check + profiling
		DP:	Debug + profiling
		DT:	Debug + trace

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

5. Stacks

5.1. Task stack for ARM

All ***embOS*** tasks execute in *system mode.* The stack-size required is the sum of the stack-size of all routines plus basic stack size.
The basic stack size is the size of memory required to store the registers of the CPU plus the stack size required by ***embOS*** -routines.
For the ARM, this minimum task stack size is about 64 bytes.

5.2. System stack for ARM

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

5.3. Interrupt stack for ARM

If a normal hardware exception occurs, the ARM core switches to IRQ mode, which has a separate stack pointer. To enable support for nested interrupts, execution of the ISR itself in a different CPU mode than *IRQ mode* is necessary*.* ***embOS*** switches to *supervisor mode* after saving scratch registers, LR_irq and SPSR_irq onto the IRQ stack.
As a result, only registers mentioned above are saved on the IRQ stack. For the interrupt routine itself, the supervisor stack is used.
The size of the interrupt stack can be changed by modifying the stack size definition of the section IRQ_STACK in the scatter file. We recommend at least 128 bytes.

5.4. Stack specifics of the ARM family

Exceptions require space on the supervisor and interrupt stack. The interrupt stack is used to store contents of scratch registers, the ISR itself uses supervisor stack.
When you intend to use FIQ in your system, you may have to modify the startup code and scatter files to implement an FIQ stack.

6. Heap

6.1. Heap management

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

Example

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

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

  HEAP +0x0 EMPTY UNINIT 0x1000 {
  }

  CSTACK +0x0 EMPTY UNINIT 0x200 {
  }

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

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
- ***embOS*** OS_IRQ_SERVICE: save scratch registers
- ***embOS*** OS_IRQ_SERVICE: save LR_irq and SPSR_irq
- ***embOS*** OS_IRQ_SERVICE: switch to *supervisor mode*
- ***embOS*** OS_IRQ_SERVICE: execute OS_irq_handler (defined in RTOSINIT_*.C)
- ***embOS*** OS_irq_handler: check for interrupt source and execute timer interrupt, serial communication or user ISR.
- ***embOS*** OS_IRQ_SERVICE: switch to *IRQ mode*
- ***embOS*** OS_IRQ_SERVICE: restore LR_irq and SPSR_irq
- ***embOS*** OS_IRQ_SERVICE: pop scratch registers
- return from interrupt

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

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

Example

"Simple" interrupt-routine

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

7.3. Interrupt handling without vectored interrupt controller

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

The reaction to an interrupt is as follows:

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

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

Example

"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, please ensure that `OS_IRQ_SERVICE()` is called from every interrupt. This is default when startup code and hardware initialization delivered with ***embOS*** is used.
The interrupt vector itself will then be examined by the “C”-level interrupt handler `OS_irq_handler()` in RTOSInit*.c.
You should not program the interrupt controller for IRQ handling directly. You should use the functions delivered with ***embOS***.

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

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

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

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

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

8. STOP / WAIT Mode

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

9. Technical data

9.1. Memory requirements

These values are neither precise nor guaranteed but they give you a good idea of the memory-requirements. They vary depending on the current version of ***embOS***. Using ARM mode, the minimum ROM requirement for the kernel itself is about 2.500 bytes. In THUMB mode kernel itself does have a minimum ROM size of about 1.700 bytes.
In the table below, you can find minimum RAM size for ***embOS*** resources. Please note, that sizes depend on selected ***embOS*** library mode; table below is for a release build.

embOS resource	RAM [bytes]
Task control block	32
Resource semaphore	8
Counting semaphore	2
Mailbox	20
Software timer	20

10. 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	`Start*.mcp`	Sample project files for ARM Codewarrior IDE.
START\INC	`RTOS.H`	Include file for ***embOS***, to be included in every "C"-file using ***embOS*** –functions.
START\LIB	`os*.a`	***embOS*** libraries
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\SRC	`retarget.c`	ARM runtime library initialization
START\SRC	`RTOSVect.s`	***embOS*** interrupt handler
START\SRC	`Vector.s`	ARM vector table.
START\CPU_*	`*.*`	CPU specific hardware routines for various CPUs.

Any other additional files shipped serve as example.

11. Index