embOS

Real Time Operating System

CPU & Compiler specifics for
ARM core with ARM Software
Development Toolkit 2.50

Document Rev. 1

A product of Segger Microcontroller Systeme GmbH

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 Software Development Toolkit*.

1.1. How to use this manual

This manual describes all CPU and compiler specifics for embOS using ARM based controllers with *ARM Software Development Toolkit*. 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 Software Development Toolkit*. 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 Software Development Toolkit*.

2. Using ***embOS*** with ARM Software Development Toolkit

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 Software Development Toolkit* 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 *ARM Software Development Toolkit* project manager, it may be used without the project manager using batch files or a make utility without any problem.

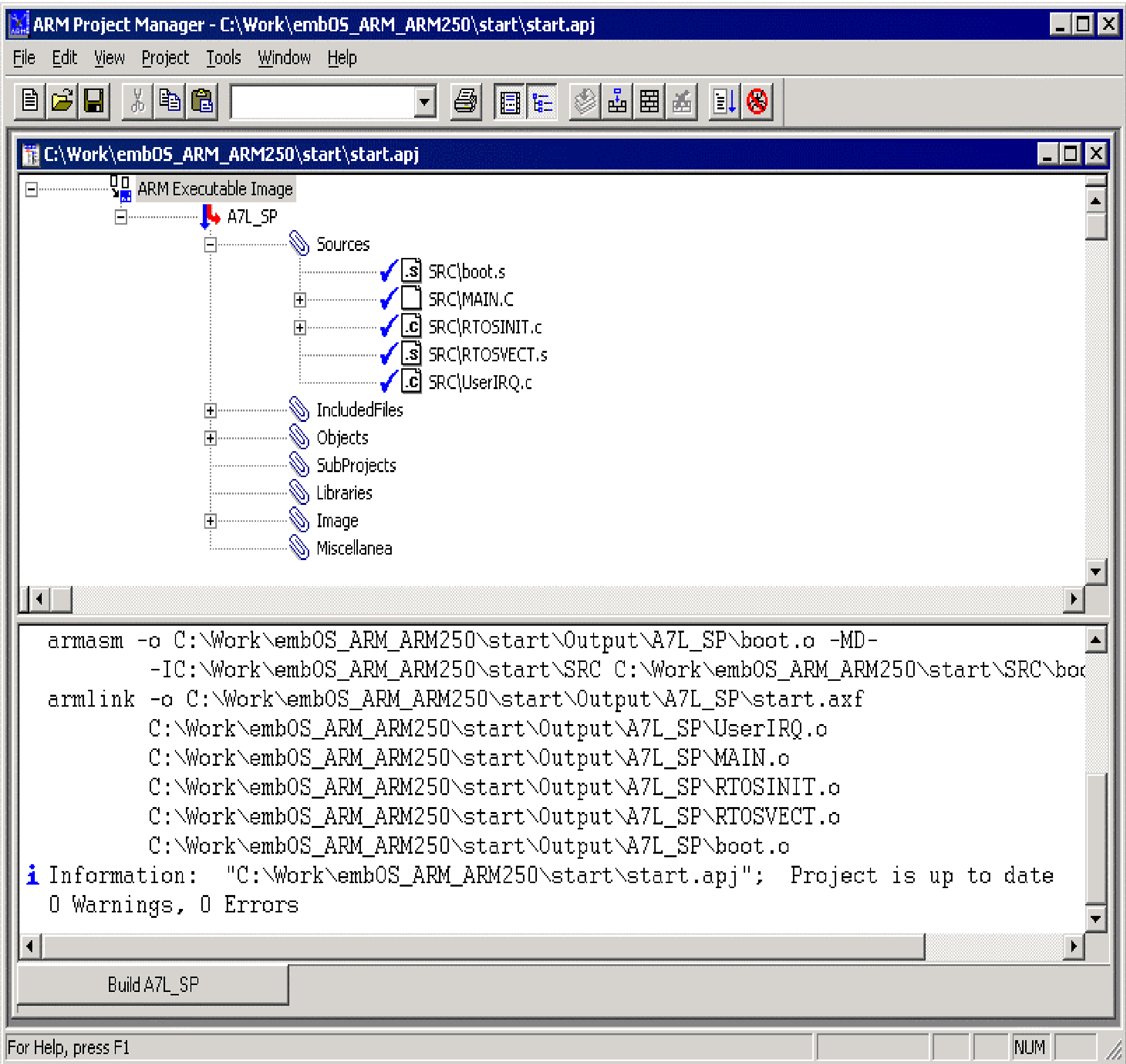
2.2. First steps

After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received a ready to go sample start project and it is a good idea to use this 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.
Open the sample project start\start.apj with *ARM Software Development Toolkit* project manager (e.g. by double clicking it)
Build the start project

Your screen should look like follows:



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

2.3. The sample application Main.c

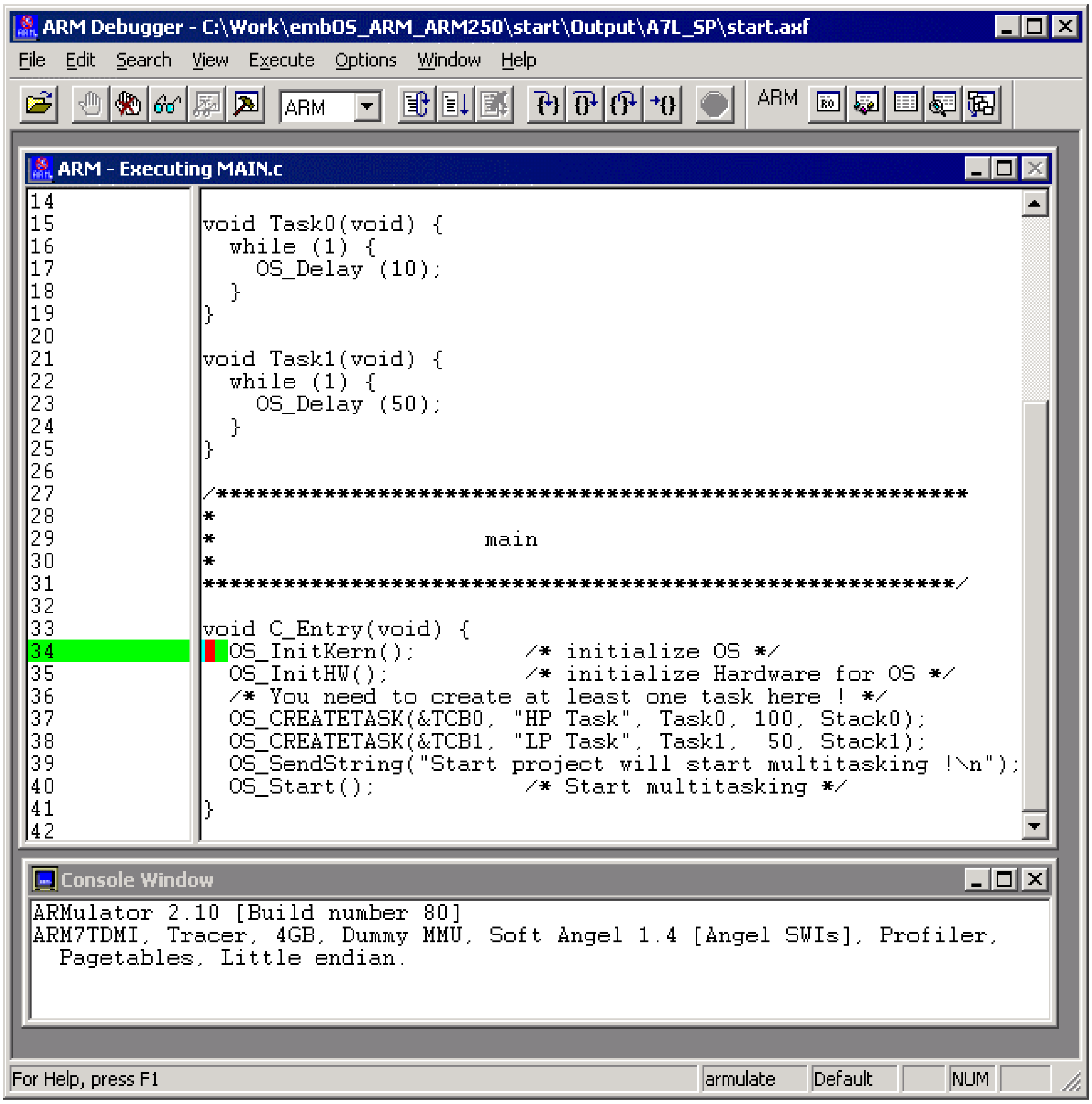
The following is a printout of the sample application main.c. It is a good starting-point for your application. (Please note that the file actually shipped with your port of ***embOS*** may look slightly different from this one)
What happens is easy to see:
After initialization of ***embOS;*** two tasks 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]; /* Stack-space */
OS_TASK TCB0, TCB1;               /* Task-control-blocks */


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

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

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

void C_Entry(void) {
  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 */
}
```

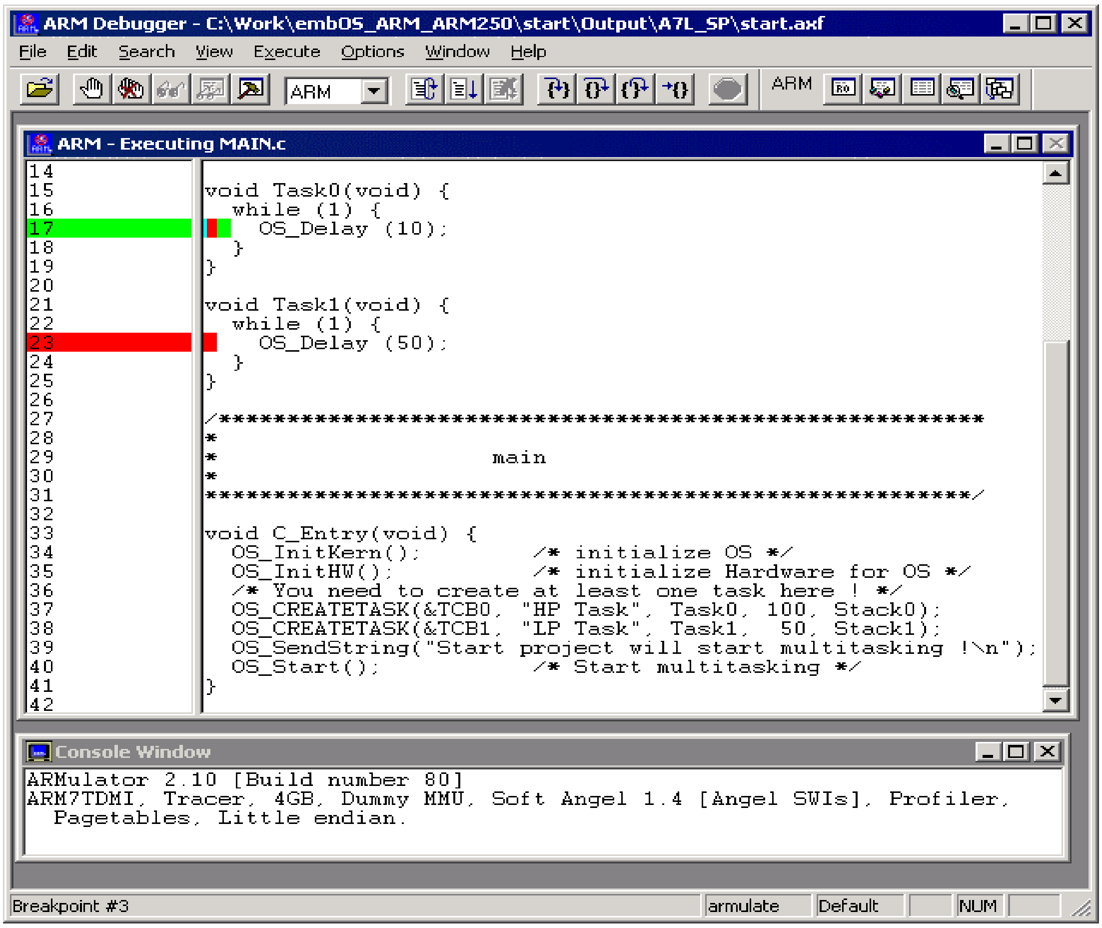
2.4. Stepping through the sample application Main.c using ARM Debugger

When starting the debugger, you will usually see the C_Entry function (very similar to the screenshot below). In some debuggers, you may look at the startup code and have to set a breakpoint at C_Entry. Now you can step thru the program.
`OS_InitKern()` is part of the ***embOS*** Library; you can therefore only step into it in disassembly mode. It initializes the relevant OS-Variables and enables interrupts. If you do not like this behavior, you are free to change it by incrementing the interrupt-disable counter using `OS_IncDI()` before the call to `OS_InitKern()`.

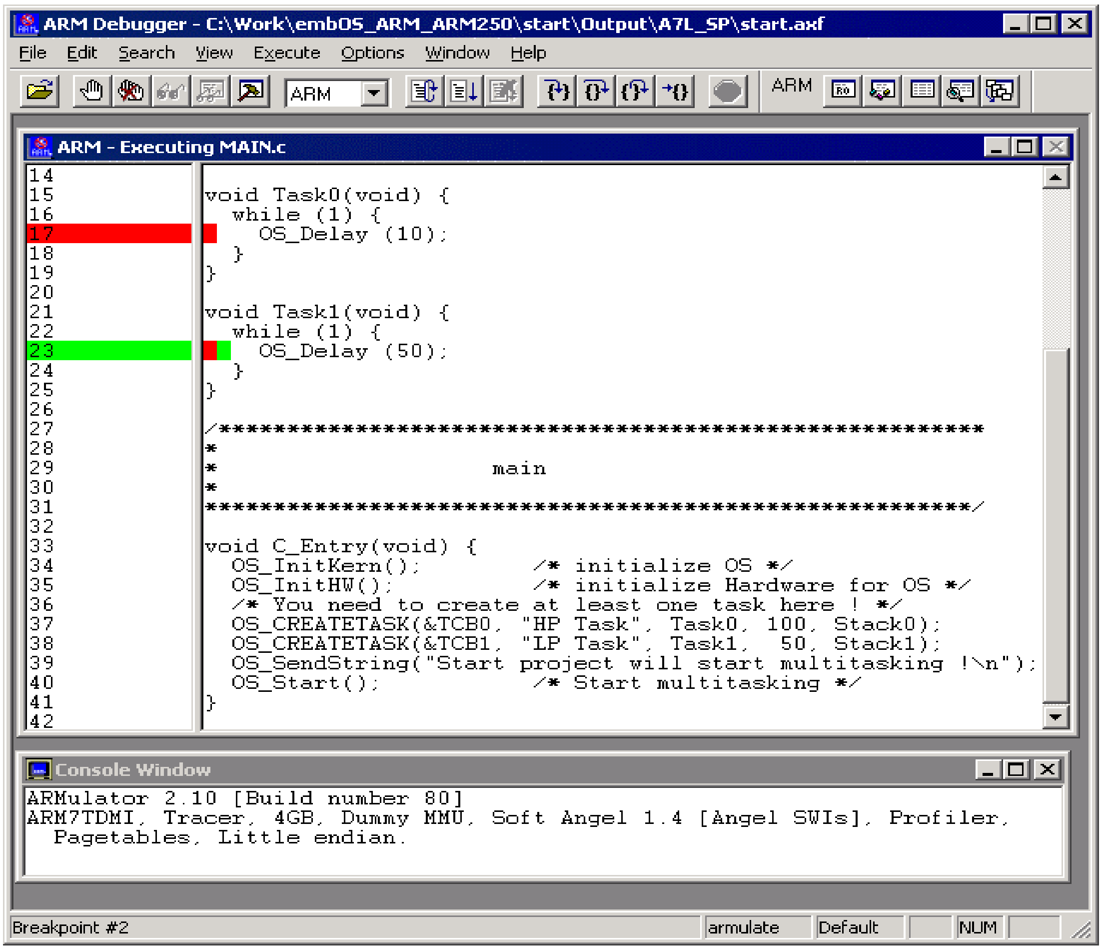
`OS_InitHW()` is part of RTOSINIT.c and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the timer-tick-interrupt for ***embOS.*** Step thru it to see what is done.
`OS_COM_Init()` is optional. It is required if embOSView shall be used. In this case it should initialize the UART used for communication.
`OS_Start()` should be the last line in C_Entry, since it starts multitasking and does not return.



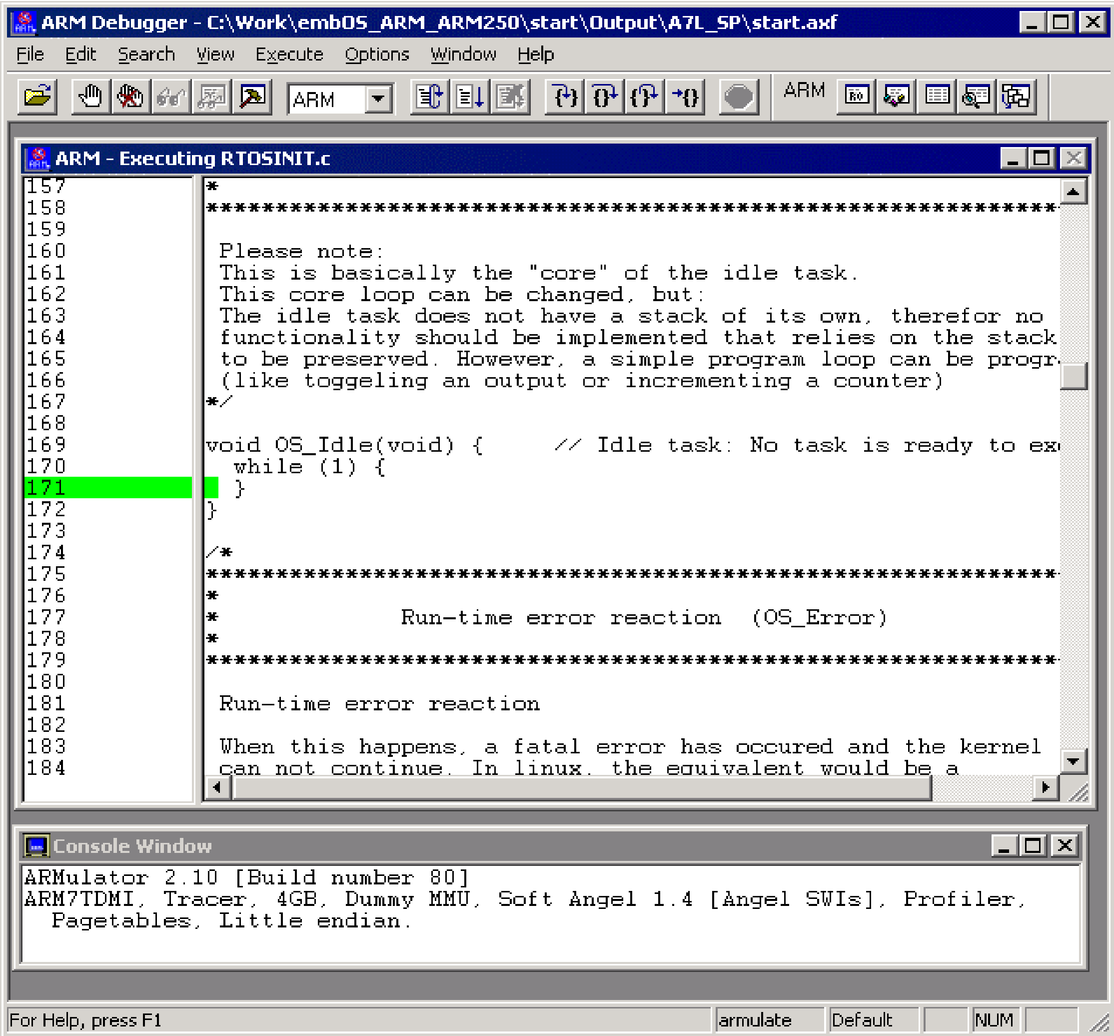
Before you step into OS_Start(), set one break point in Task0 and one in Task1. When you step into OS_Start(), you will only step into it in disassembly mode, because this function is part of the ***embOS*** library. However, you can press GO now or step in disassembly mode until you reach the highest priority task.



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



Continuing to step thru 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).



If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay. If you inspect system variable OS_Time, you can see how much time has expired in the target system. However, when using the ARMULATOR or any other simulator, OS_Time will not increment, because no timer interrupt is generated. As a result, the program will stick in the idle loop instead of stopping in one of the tasks again.

3. ARM specifics

3.1. CPU modes

embOS supports THUMB and ARM mode of the ARM 7/9 CPU. In THUMB mode, all OS modules have been compiled with option "-apcs /interwork" to enable an easy interface between ARM modules and THUMB modules of your application.

3.2. Available libraries

Core	Mode	Endianess	Library type	Library
ARM 7	ARM	little	Release	OsA7LR.alf
ARM 7	ARM	little	Stack-check	OsA7LS.alf
ARM 7	ARM	little	Stack-check + Profiling	OsA7LSP.alf
ARM 7	ARM	little	Debug	OsA7LD.alf
ARM 7	ARM	little	Debug + Profiling	OsA7LDP.alf
ARM 7	ARM	little	Debug + Trace	OsA7LDT.alf
ARM 9	ARM	little	Release	OsA9LR.alf
ARM 9	ARM	little	Stack-check	OsA9LS.alf
ARM 9	ARM	little	Stack-check + Profiling	OsA9LSP.alf
ARM 9	ARM	little	Debug	OsA9LD.alf
ARM 9	ARM	little	Debug + Profiling	OsA9LDP.alf
ARM 9	ARM	little	Debug + Trace	OsA9LDT.alf
ARM 7	THUMB	little	Release	OsT7LR.alf
ARM 7	THUMB	little	Stack-check	OsT7LS.alf
ARM 7	THUMB	little	Stack-check + Profiling	OsT7LSP.alf
ARM 7	THUMB	little	Debug	OsT7LD.alf
ARM 7	THUMB	little	Debug + Profiling	OsT7LDP.alf
ARM 7	THUMB	little	Debug + Trace	OsT7LDT.alf
ARM 9	THUMB	little	Release	OsT9LR.alf
ARM 9	THUMB	little	Stack-check	OsT9LS.alf
ARM 9	THUMB	little	Stack-check + Profiling	OsT9LSP.alf
ARM 9	THUMB	little	Debug	OsT9LD.alf
ARM 9	THUMB	little	Debug + Profiling	OsT9LDP.alf
ARM 9	THUMB	Little	Debug + Trace	OsT9LDT.alf

3.3. Entry point for C code

Due to the fact, that the ARM linker does link a special version of the C library in case a symbol *main* is detected, ***embOS*** does not use a *main* function. The function *C_Entry* is used instead, so that an ***embOS*** application does not rely on any debug environment and can execute in ROM also. For details, please see also

ARM Software Development Toolkit
USER GUIDE
Chapter 10 - Writing Code for ROM
10.3.11 Entering C code

4. Stacks

4.1. Task stack for ARM 7 and ARM 9

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 7/9, this minimum task stack size is about 56 bytes.

4.2. 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 128 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 also use the system-stack, the actual stack requirements depend on the application.
The size of the system stack can be changed by modifying value of SVC_STACK_SIZE in the file boot.s.

4.3. Interrupt stack for ARM 7 and ARM 9

If a normal hardware exception does occur, 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*** does switch 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 value of IRQ_STACK_SIZE in the file boot.s. We recommend at least 128 bytes.

4.4. Stack specifics of the ARM 7 and ARM 9 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.

5. Interrupts

5.1. What happens when an interrupt occurs?

- The CPU-core receives an interrupt request
- As soon as the interrupts are enabled, the interrupt is executed
- the CPU switches to the Interrupt stack
- the CPU saves PC and flags in registers LR_irq and SPSR_irq
- the CPU jumps to the vector address 0x18
- ***embOS*** OS_IRQ_SERVICE: save scratch registers
- ***embOS*** OS_IRQ_SERVICE: save LR_irq and SPSR_irq
- ***embOS*** OS_IRQ_SERVICE: switch to *supervisor mode*
- ***embOS*** OS_IRQ_SERVICE: execute OS_irq_handler (defined in RTOSINIT.C)
- ***embOS*** OS_irq_handler: check for interrupt source and execute timer interrupt, serial communication or user ISR (OS_USER_irq_func).
- ***embOS*** OS_IRQ_SERVICE: switch to *IRQ mode*
- ***embOS*** OS_IRQ_SERVICE: restore LR_irq and SPSR_irq
- ***embOS*** OS_IRQ_SERVICE: pop scratch registers
- return from interrupt

5.2. Defining interrupt handlers in "C"

The default C interrupt handler checks for all internal ***embOS*** related interrupts, such as timer and serial communication. In case none of these sources is responsible for the exception, a user defined function OS_USER_irq_func (usually defined in module UserIRQ.C) is called. Unless there are good reasons to do so, you should modify the code in OS_USER_irq_func only and leave the handler in RTOSINIT.C as it is. The advantage is an easier migration in case you get an update for ***embOS***; there might be modifications in the ***embOS*** module RTOSINIT.C.

Example

"Simple" interrupt-routine

```
void OS_USER_irq_func(void) {
  #if defined(CPU_KS32C50100)
    if (__INTPND&0x0800) {
      __INTPND = 0x0800;
      OSTEST_X_ISR0();
    }
  #elif defined(CPU_LH79531)
      if (IRQ_STATUS & OSTEST_TIMER_IRQ_MASK) {
        OSTEST_TIMER_IRQ_CLEAR = OSTEST_TIMER_IRQ_MASK;
        OSTEST_X_ISR0();
      }
  #else
    #error "Please define a CPU"
  #endif
}
```

5.3. Interrupt-stack

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.

5.4. Special considerations for the ARM 7 and ARM 9

None.

6. 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.

7. Technical data

7.1. 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	4
Mailbox	20
Software timer	20

8. Files shipped with ***embOS***

Directory	File	Explanation
INC	RTOS.H	Include file for RTOS, to be included in every "C"-file using RTOS-functions
LIB	`OS*.alf`	Libraries for all memory models and debug options
SRC	Boot.s	Low level assembler startup code
SRC	RtosVect.s	Assembler part of interrupt handler
SRC	UserIRQ.c	Frame for user interrupt function
SRC	RtosInit.c	Initializes the hardware, can be modified if required
SRC	Main.c	Frame program to serve as a start; C_Entry is the C level entry point.

Any additional file shipped as example.

9. Index