embOS

Real Time Operating System

CPU & Compiler specifics for
ARM cores using
ARM Developer Suite 1.2

Document revision 3
Date: January 17, 2008

SEGGER

A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Contact address

SEGGER Microcontroller GmbH & Co. KG

Heinrich-Hertz-Str. 5
D-40721 Hilden

Germany

Tel.+49 2103-2878-0
Fax.+49 2103-2878-28
Email: support@segger.com
Internet: http://www.segger.com

Manual versions

This manual describes the latest software version. If any error occurs, please inform us and we will try to assist you as soon as possible.

For further information on topics or routines not yet specified, please contact us.

Manual version	Date	By	Explanation
3.00	070928	OO	New initial version

Software versions

Refer to Release.html for information about the changes of the software versions.

About this document

Assumptions

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

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

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

How to use this manual

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

Typographic conventions for syntax

This manual uses the following typographic conventions:

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

Table 1.1: Typographic conventions

Table of Contents

1 Introduction 7

2 Using embOS with ARM Developer Suite..9

2.1 Installation 10
2.2 First steps 11
2.3 The sample application Main.c 12
2.4 Stepping through the sample application using ARM debugger 13

3 Build your own application 17

3.1 Required files for an embOS application 18
3.2 Change library mode... 19
3.3 Select an other CPU 19

4 ARM specifics 21

4.1 CPU modes 22
4.2 Available libraries 22

5 Stacks 23

5.1 Task stack for ARM 24
5.2 System stack for ARM 24
5.3 Interrupt stack for ARM 24
5.4 Stack specifics of the ARM family 24

6 Heap 25

6.1 Heap management 26

7 Interrupts...27

7.1 What happens when an interrupt occurs 28
7.2 Defining interrupt handlers in "C".. 28
7.3 Interrupt stack 28
7.4 Fast interrupt FIQ 28

8 STOP / WAIT mode 29

8.1 Saving power 30

9 Technical data..31

9.1 Memory requirements 32

10 Files shipped with embOS 33

10.1 Files included in embOS.. 34

Chapter 1

Introduction

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

How to use this manual

This manual describes all CPU and compiler specifics of ***embOS*** using ARM based controllers with ARM 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*** for ARM 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 Developer Suite.

Chapter 2

Using embOS with ARM 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 Developer Suite 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 heading *First steps* on page 11.

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 with the project manager, you should:
Copy either all or only the library-file that you need to your work-directory. Also copy the entire CPU specific subdirectory and the ***embOS*** header file `RTOS.h`. 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 Developer Suite project manager, it may be used without the project manager using batch files or a make utility without any problem.

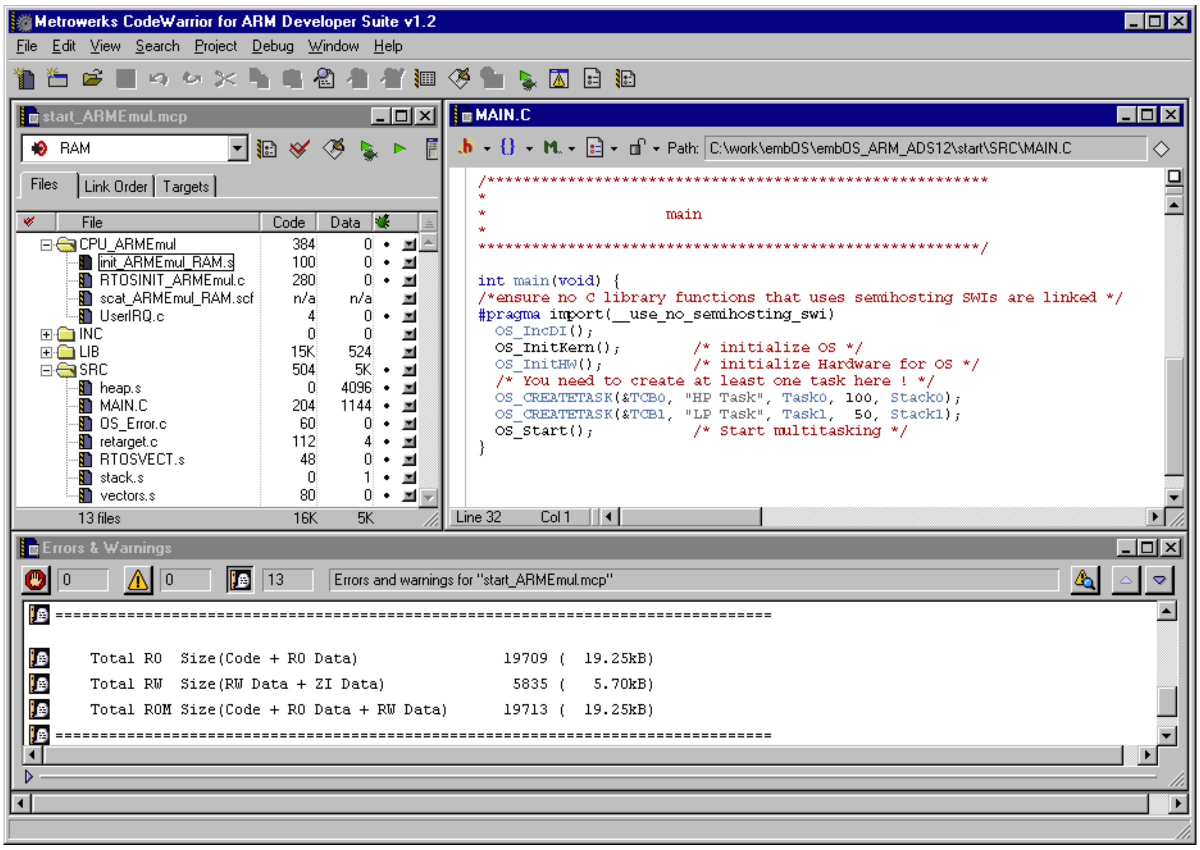
2.2 First steps

After installation of ***embOS*** (See “Installation” on page 10.) 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.

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
- For first steps, open the project for the ARMEmulator (`Start\Start_ARMEmul.mcp`) with ARM Developer Suite project manager (e.g. by double clicking it)
- Build the start project

After building the start project your screen should look like follows:



Metrowerks CodeWarrior for ARM Developer Suite v1.2
File Edit View Search Project Debug Window Help
start_ARMEmul.mcp
RAM
Files | Link Order | Targets
MAIN.C
Path: C:\work\embOS\embOS_ARM_ADS12\start\SRC\MAIN.C
Line 32 Col 1
Errors & Warnings
0 0 13
Errors and warnings for "start_ARMEmul.mcp"

File	Code	Data
CPU_ARMEmul	384	0
init_ARMEmul_RAM.s	100	0
RTOSINIT_ARMEmul.c	280	0
scat_ARMEmul_RAM.scf	n/a	n/a
UserIRQ.c	4	0
INC	0	0
LIB	15K	524
SRC	504	5K
heap.s	0	4096
MAIN.C	204	1144
OS_Error.c	60	0
retarget.c	112	4
RTOSVECT.s	48	0
stack.s	0	1
vectors.s	80	0
13 files	16K	5K

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

int main(void) {
/*ensure no C library functions that uses semihosting SWIs are linked */
#pragma import(__use_no_semihosting_swi)
  OS_IncDI();
  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_Start();             /* Start multitasking */
}
```

```
================================================================================

    Total RO  Size(Code + RO Data)                 19709 (  19.25kB)
    Total RW  Size(RW Data + ZI Data)               5835 (   5.70kB)
    Total ROM Size(Code + RO Data + RW Data)       19713 (  19.25kB)

================================================================================
```

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

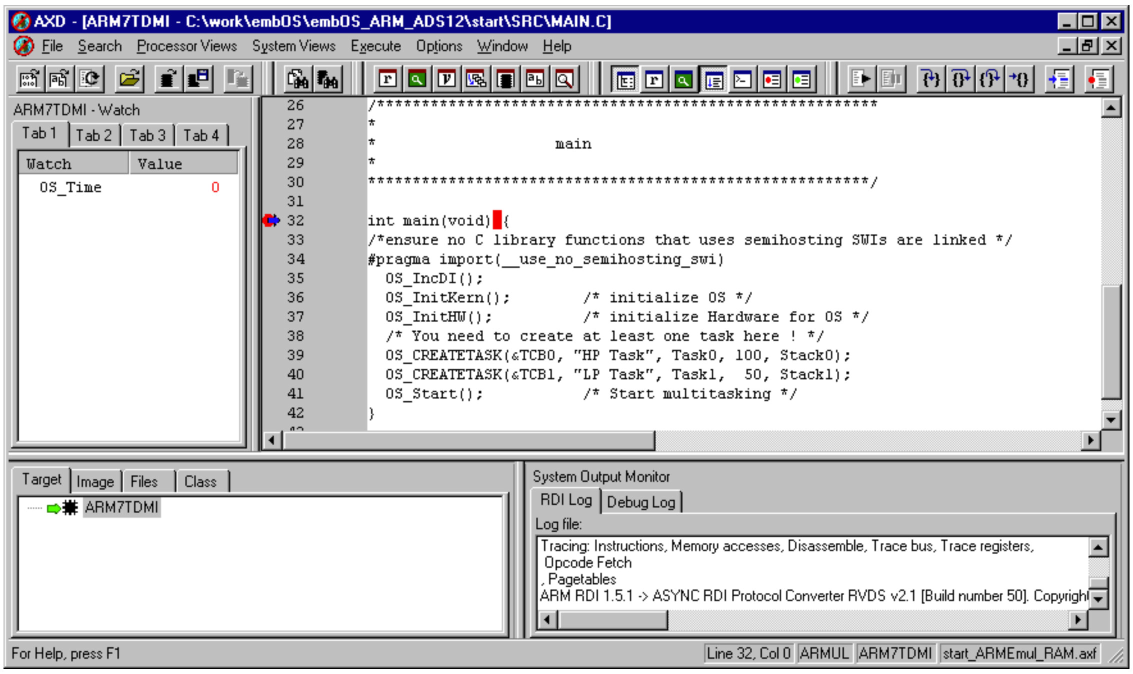
2.3 The sample application Main.c

The following is a printout of the sample application `main.c`. It is a good startingpoint 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 */

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

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 ARM 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.


AXD - [ARM7TDMI - C:\work\embOS\embOS_ARM_ADS12\start\SRC\MAIN.C]
File Search Processor Views System Views Execute Options Window Help
ARM7TDMI - Watch
Tab 1 Tab 2 Tab 3 Tab 4
Target Image Files Class
ARM7TDMI
System Output Monitor
RDI Log Debug Log
For Help, press F1

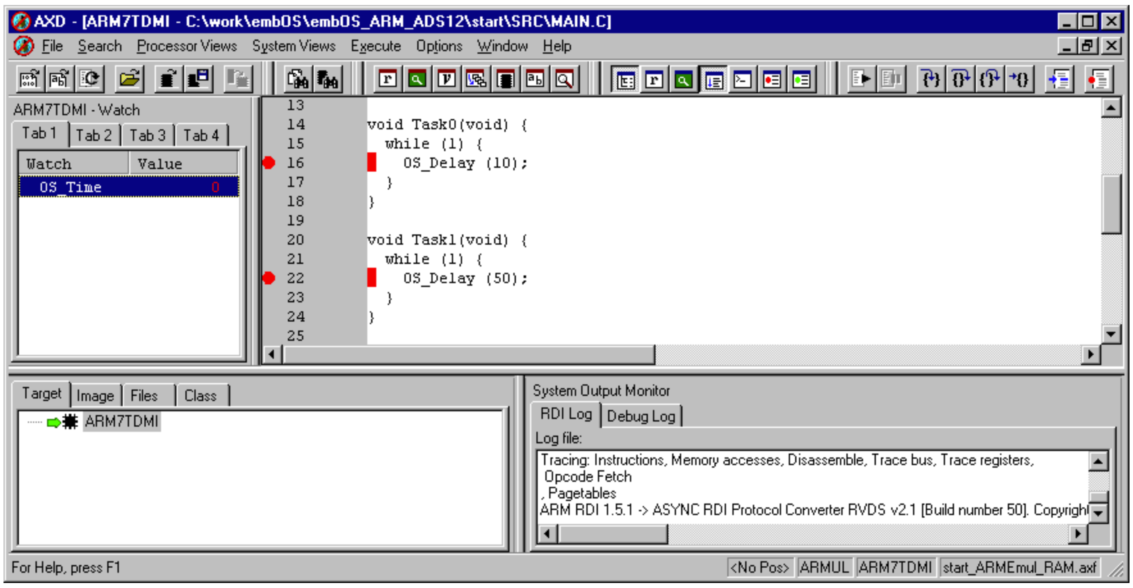
Watch	Value
OS_Time	0

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

int main(void) {
/*ensure no C library functions that uses semihosting SWIs are linked */
#pragma import(__use_no_semihosting_swi)
  OS_IncDI();
  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_Start();           /* Start multitasking */
}
```

```
Log file:
Tracing: Instructions, Memory accesses, Disassemble, Trace bus, Trace registers,
 Opcode Fetch
, Pagetables
ARM RDI 1.5.1 -> ASYNC RDI Protocol Converter RVDS v2.1 [Build number 50]. Copyright
```

Line 32, Col 0 | ARMUL | ARM7TDMI | start_ARMEmul_RAM.axf

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


AXD - [ARM7TDMI - C:\work\embOS\embOS_ARM_ADS12\start\SRC\MAIN.C]
File Search Processor Views System Views Execute Options Window Help
ARM7TDMI - Watch
Tab 1 Tab 2 Tab 3 Tab 4
Target Image Files Class
ARM7TDMI
System Output Monitor
RDI Log Debug Log
For Help, press F1

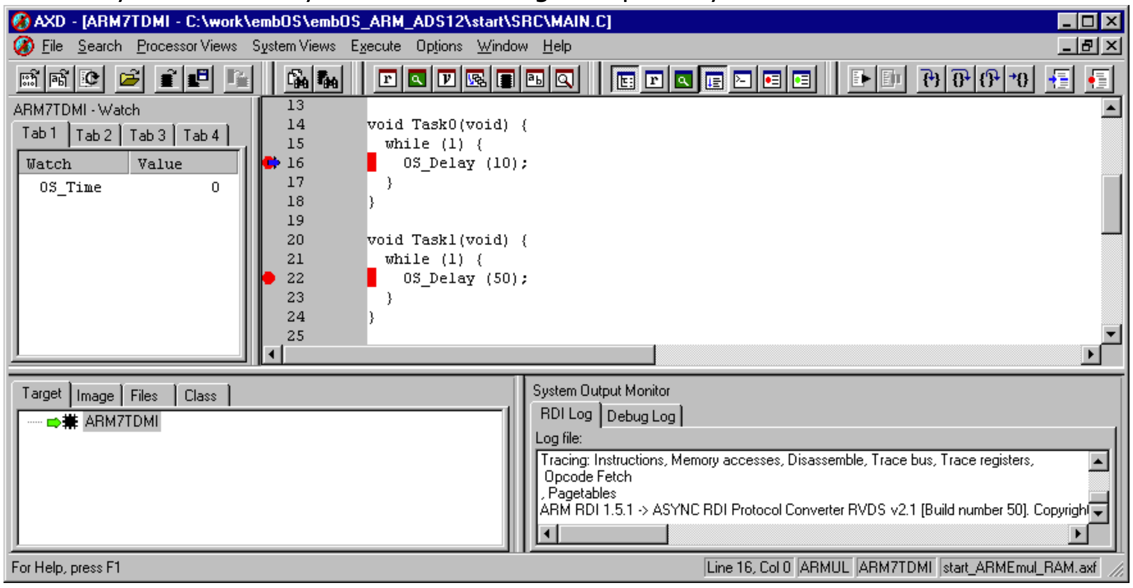
Watch	Value
OS_Time	0

```

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

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

```

```
Log file:
Tracing: Instructions, Memory accesses, Disassemble, Trace bus, Trace registers,
 Opcode Fetch
, Pagetables
ARM RDI 1.5.1 -> ASYNC RDI Protocol Converter RVDS v2.1 [Build number 50]. Copyright
```

<No Pos> | ARMUL | ARM7TDMI | start_ARMEmul_RAM.axf

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.


AXD - [ARM7TDMI - C:\work\embOS\embOS_ARM_ADS12\start\SRC\MAIN.C]
File Search Processor Views System Views Execute Options Window Help
ARM7TDMI - Watch
Tab 1 Tab 2 Tab 3 Tab 4
Target Image Files Class
ARM7TDMI
System Output Monitor
RDI Log Debug Log
For Help, press F1

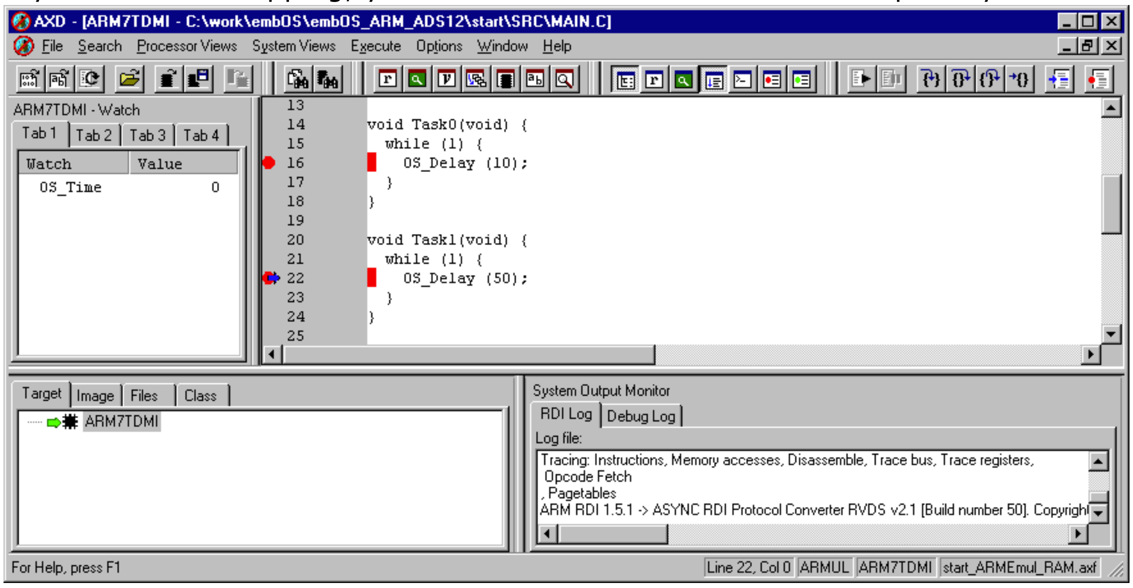
Watch	Value
OS_Time	0

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

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

```
Log file:
Tracing: Instructions, Memory accesses, Disassemble, Trace bus, Trace registers,
 Opcode Fetch
, Pagetables
ARM RDI 1.5.1 -> ASYNC RDI Protocol Converter RVDS v2.1 [Build number 50]. Copyrigh
```

```
Line 16, Col 0  ARMUL  ARM7TDMI  start_ARMEmul_RAM.axf
```

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


AXD - [ARM7TDMI - C:\work\embOS\embOS_ARM_ADS12\start\SRC\MAIN.C]
File Search Processor Views System Views Execute Options Window Help
ARM7TDMI - Watch
Tab 1 Tab 2 Tab 3 Tab 4
Target Image Files Class
ARM7TDMI
System Output Monitor
RDI Log Debug Log
For Help, press F1

Watch	Value
OS_Time	0

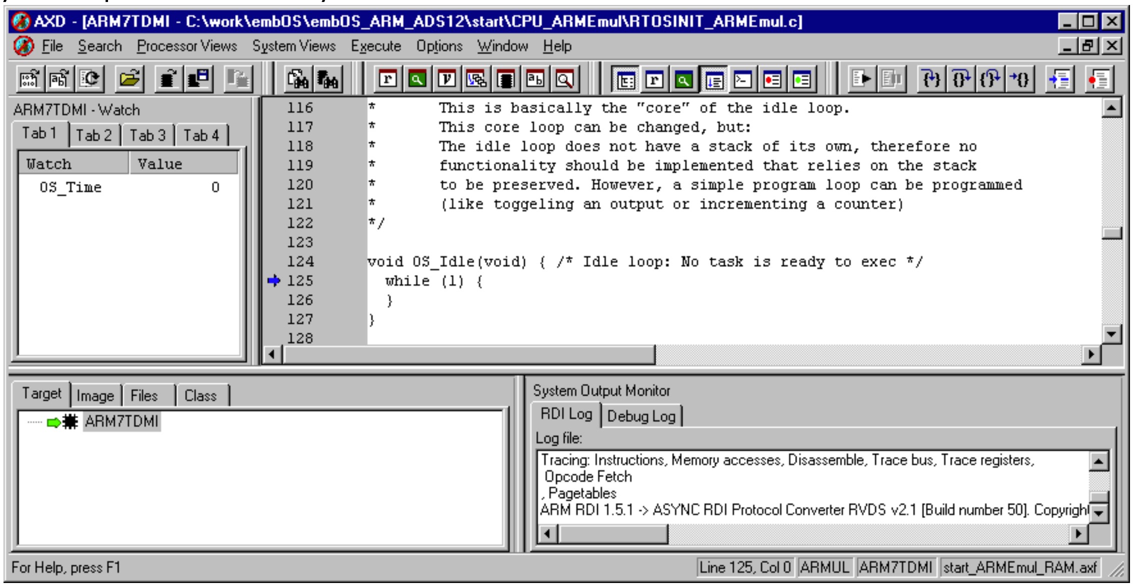
```
void Task0(void) {
  while (1) {
    OS_Delay (10);
  }
}

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

```
Log file:
Tracing: Instructions, Memory accesses, Disassemble, Trace bus, Trace registers,
 Opcode Fetch
, Pagetables
ARM RDI 1.5.1 -> ASYNC RDI Protocol Converter RVDS v2.1 [Build number 50]. Copyrigh
```

```
Line 22, Col 0  ARMUL  ARM7TDMI  start_ARMEmul_RAM.axf
```

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

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


AXD - [ARM7TDMI - C:\work\embOS\embOS_ARM_ADS12\start\CPU_ARMEmul\RTOSINIT_ARMEmul.c]

ARM7TDMI - Watch

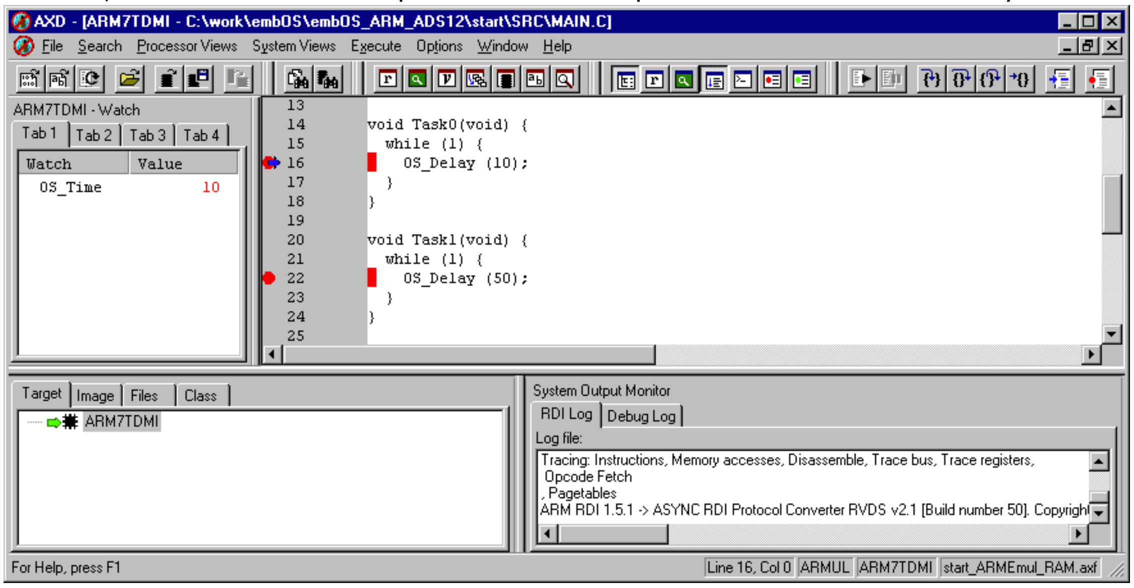
Watch	Value
OS_Time	0

```
*       This is basically the "core" of the idle loop.
*       This core loop can be changed, but:
*       The idle loop does not have a stack of its own, therefore no
*       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) {
  }
}
```

```
Tracing: Instructions, Memory accesses, Disassemble, Trace bus, Trace registers,
 Opcode Fetch
, Pagetables
ARM RDI 1.5.1 -> ASYNC RDI Protocol Converter RVDS v2.1 [Build number 50]. Copyright
```

For Help, press F1 | Line 125, Col 0 | ARMUL | ARM7TDMI | start_ARMEmul_RAM.axf

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, HPTask continues operation after expiration of the 10 ms delay.


AXD - [ARM7TDMI - C:\work\embOS\embOS_ARM_ADS12\start\SRC\MAIN.C]

ARM7TDMI - Watch

Watch	Value
OS_Time	10

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

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

```
Tracing: Instructions, Memory accesses, Disassemble, Trace bus, Trace registers,
 Opcode Fetch
, Pagetables
ARM RDI 1.5.1 -> ASYNC RDI Protocol Converter RVDS v2.1 [Build number 50]. Copyright
```

For Help, press F1 | Line 16, Col 0 | ARMUL | ARM7TDMI | start_ARMEmul_RAM.axf

Chapter 3

Build your own application

To build your own application, you should always start with a copy of the 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 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 subfolder `Inc\`
  This header file declares all ***embOS*** API functions and data types and has to be included in any source file using ***embOS*** functions.
- **RTOSInit_*.c** from one `of the CPU` subfolder.
  It contains the hardware dependent initialization code for the specific CPU, the ***embOS*** timer and optional UART for `embOSView`.
- **UserIRQ.c** from one of the 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 subfolders.
  It contains the low level initialization of the hardware and setup of the various stack pointer.
- **scat_*.scf** from one of the CPU subfolders.
  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 the "Src\"-subfolder.
  The error handler is used if any ***embOS*** library other than the Release build library is used in your project.
- **vector.s** from the "Src\"-subfolder.
  This file contains the ARM vector table.
- **heap.s** from the "Src\"-subfolder.
  It contains the heap size for dynamic memory allocation.
- **stack.s** from the "Src\"-subfolder.
  This file contains a dummy variable which tells the linker where to put the stack.
- **retarget.c** from the "Src\"-subfolder.
  Since you use your own linker file (scatter file) some functions need to be rewritten for the ARM ADS 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 ADS Compilers and Libraries Guide.
- 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 "Src\"-subfolder.

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 embOSLib groups or remove them from the configuration.
- Check and set the appropriate `OS_LIBMODE_*` define which you would like to use for debug and release builds.

3.3 Select an other CPU

embOS for ARM and ADS contains CPU specific code for various ARM CPUs. The sample start workspace contains projetcs for different 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 select 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()`, the interrupt service routines for ***embOS*** timer tick and communication to `embOSView`.

Chapter 4

ARM specifics

4.1 CPU modes

embOS supports nearly all memory and code model combinations that ARM's ADS 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 for ADS comes 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><e><LibMode>.alf

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.alf` is the library for a project using THUMB mode, ARM 7/9 core, little endian mode and release build library type.

Chapter 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()`), and because software-timers also use the systemstack, the actual stack requirements depend on the application.
The size of the system stack can be changed by modifying value of `Len_SVC_Stack` in the file `init_*.s`.

5.3 Interrupt stack for ARM

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 `Len_IRQ_Stack` in the file `init_*.s`. 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.

Chapter 6

Heap

6.1 Heap management

If you intend to use heap for dynamic memory allocation, the file `heap.s` needs to be modified. The variable `bottom_of_heap` defines how many bytes you want to allocate for heap:

Example of heap.s

```
        AREA    Heap, DATA, NOINIT

        EXPORT bottom_of_heap

; Create dummy variable used to locate bottom of heap

bottom_of_heap   SPACE   0x1000  ; reserve 4kb heap

        END
```

You may also need to modify your scatter file (`scat_*.scf`) according to the modification that was made to the `heap.s`.

Example

```
FLASH 0x00000000 0x1000000
{
  FLASH 0x00000000 0x1FFFF
  {
    vectors.o (Vect, +First)
    init*.o (Init)
    * (+RO)
  }

  XRAM 0x40000 0x1FFFF
  {
    * (+RW,+ZI)
  }

  HEAP +0x0 UNINIT 0x1000
  {
    heap.o (+ZI)
  }

  STACKS +0x0 UNINIT
  {
    stack.o (+RW, +ZI)
  }
}
```

Chapter 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 (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.

7.2 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 of a "simple" interrupt-routine

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

7.3 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.
The ARM interrupt stack is used for primary interrupt handler in `RTOSVect.s` only.

7.4 Fast interrupt FIQ

FIQ interrupt can not be used with ***embOS*** functions, it is reserved for high speed user functions.
`FIQ` is never disabled by ***embOS***.
Never call any ***embOS*** function from an `FIQ` handler.
Do not assign any ***embOS*** interrupt handler to `FIQ`.
When you decide to use `FIQ`, please ensure that `FIQ` stack is initialized during startup and an interrupt vector for `FIQ` handling is included in your application.

Chapter 8

STOP / WAIT mode

8.1 Saving power

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 `RTOSIinit_*.c`.

Chapter 9

Technical data

9.1 Memory requirements

These values are neither precise nor guaranteed but they give you a good idea of the memory-requirements. They vary depending on the current version of ***embOS***. 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 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	2
Mailbox	20
Software timer	20

Chapter 10

Files shipped with embOS

10.1 Files included in embOS

Directory	File	Explanation
root	*.pdf	Generic API and target specific documentation
root	embOSView.exe	Utility for runtime analysis, described in generic documentation
root	Release.html	Version control document
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*.alf	***embOS*** libraries
Start\Src	heap.s	Sample frame for heap definition
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	stack.s	Dummy stack placement file for linker
Start\Src	Vector.s	ARM vector table
Start\CPU_*	*.*	CPU specific hardware routines for various CPUs

Any additional files shipped serve as example.

Index

F
FIQ 28

H
Heap 25

I
Installation 10
Interrupt stack 24
Interrupts, FIQ 28

M
Memory models 22
Memory requirements 32

S
Stacks 23
  Interrupt stack 24
  System stack 24
STOP / WAIT mode 29
Syntax, conventions used 3
System stack 24