embOS

Real Time Operating System

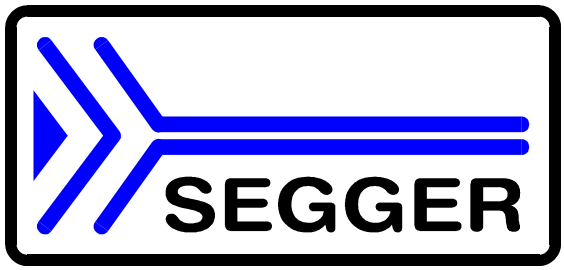
CPU & Compiler specifics for

ATMEL AVR with

IAR compiler and

Embedded Workbench

Document Rev. 6



A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Contents

Contents 3
1. About this document 4
    1.1. How to use this manual 4
2. Using ***embOS*** with IAR Embedded Workbench 5
    2.1. Installation 5
    2.2. First steps 6
    2.3. The sample application Main.c 7
    2.4. Stepping through the sample application Main.c using CSpy 7
3. Build your own application 11
    3.1. Required files for an ***embOS*** application 11
    3.2. Select a start project 11
    3.3. Add your own code 11
    3.4. Change library mode 11
4. AT90 / ATMega specifics 12
    4.1. Memory models 12
    4.2. Available libraries 12
    4.3. Distributed project files 13
5. Stacks 14
    5.1. Stack address range 14
    5.2. System stack 14
    5.3. Task stacks 14
6. Interrupts 15
    6.1. What happens when an interrupt occurs? 15
    6.2. Defining interrupt handlers in "C" 15
    6.3. Interrupt-stack 16
7. Idle Mode 17
8. Technical data 17
    8.1. Memory requirements 17
9. Files shipped with ***embOS*** 17
10. Index 18

1. About this document

This guide describes how to use ***embOS*** Real Time Operating System for the ATMEL AT90/ATmega series of microcontrollers using IAR compiler and Embedded Workbench.

1.1. How to use this manual

This manual describes all CPU and compiler specifics for ***embOS*** using ATMEL AT90 / ATmega 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*** 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 detailed information about functionality and fine-tuning of ***embOS*** for the ATMEL AT90 / ATmega based controllers using *IAR Embedded Workbench*.

2. Using ***embOS*** with 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* to develop your application, no further installation steps are required. You will find a prepared sample start project, 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 *IAR's Embedded Workbench* for your application development in order to become familiar with ***embOS.***

If for some reason you do not want to work with *IAR's Embedded Workbench*, 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's Embedded Workbench*, it may be used with batch files or a make utilities without any problem.

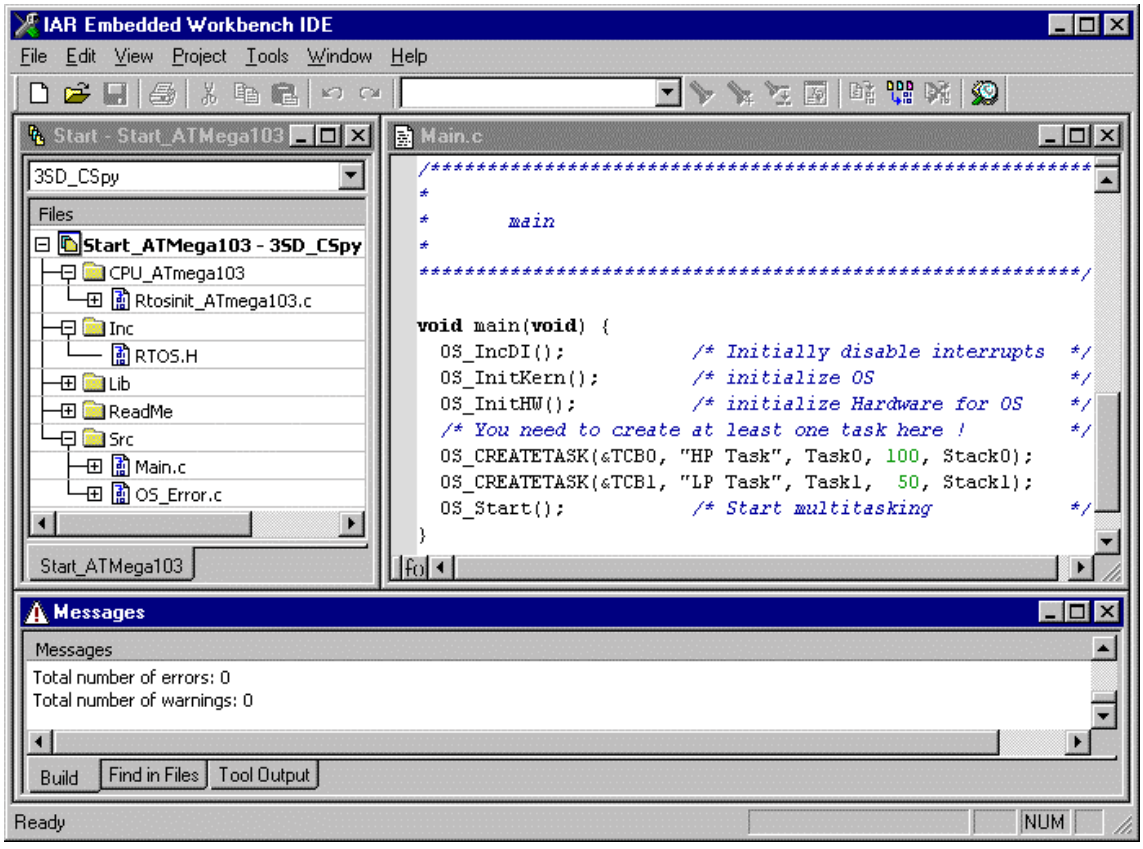
2.2. First steps

After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received a ready to go sample start workspace and project and it is a good idea to use this as a starting point of all your applications, as this project contains all compiler settings needed for ***embOS***.

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' from your ***embOS*** distribution into your work directory.
- Clear the read only attribute of all files in the new 'Start'-folder in your working directory.
- Open the folder 'Start'.
- Open the sample workspace 'Start.eww'. (e.g. by double clicking it)
- Select the configuration for CSpy simulator 3SD_CSpy
- 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

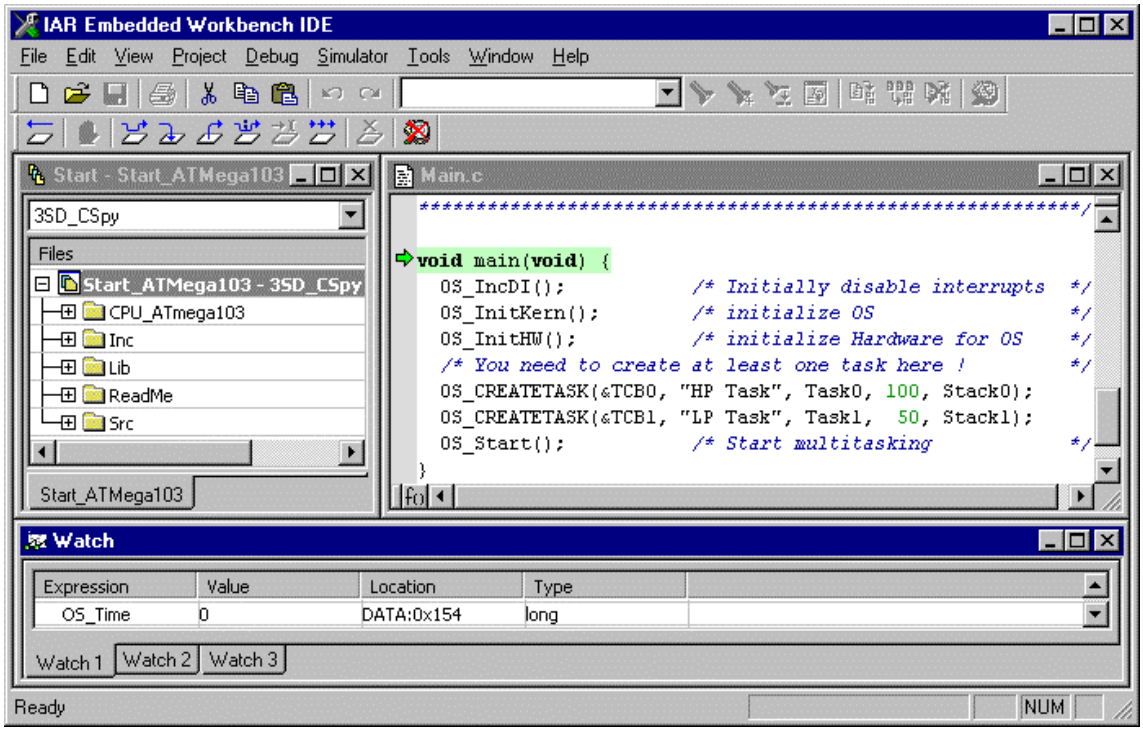
The following is a printout of the sample application main.c. It is a good starting-point for your application. (Please note that the file actually shipped with your port of ***embOS*** may look slightly different from this one)
What happens is easy to see:
After initialization of ***embOS;*** two tasks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

void 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(&TCB0, "HP Task", Task0, 100, Stack0);
  OS_CREATETASK(&TCB1, "LP Task", Task1,  50, Stack1);
  OS_Start();           /* Start multitasking            */
}
```

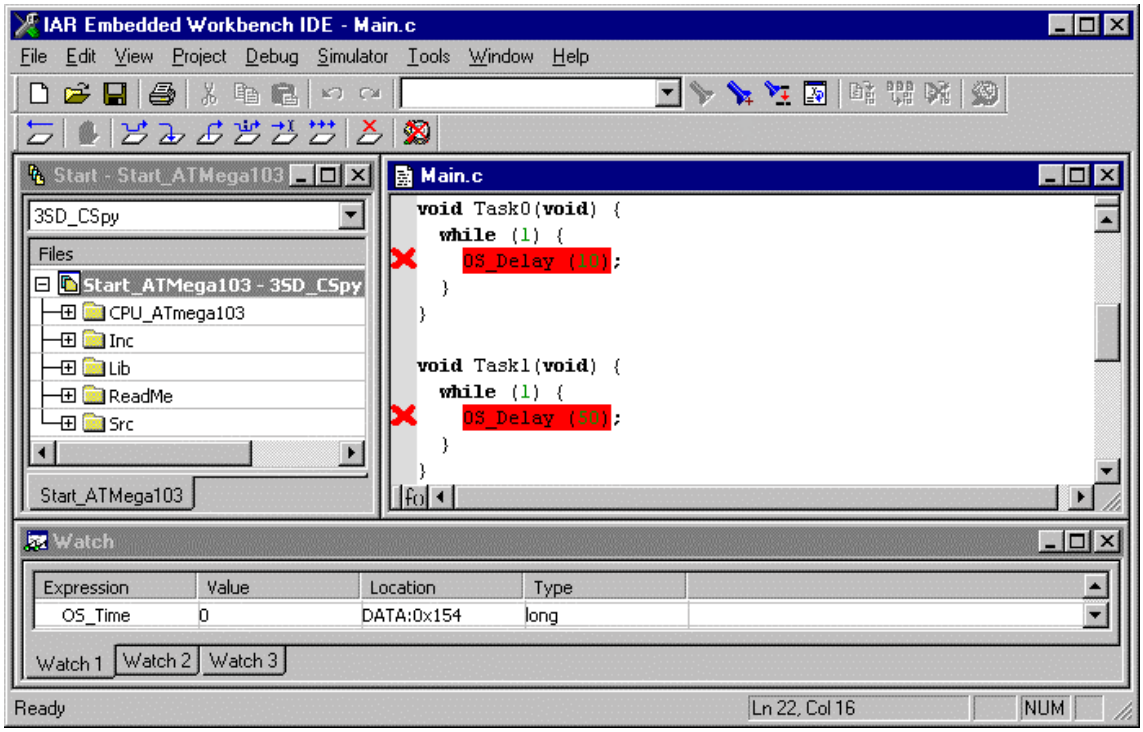
2.4. Stepping through the sample application Main.c using CSpy

When starting the debugger, you will usually see the *main* function (very similar to the screenshot below). Now you can step through the program.
`OS_IncDI()`initailly disables interrupts and inhibits re-enabling of interrupts during execution of `OS_InitKern()`.
`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 unless `OS_IncDI()` was not called before.
`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.

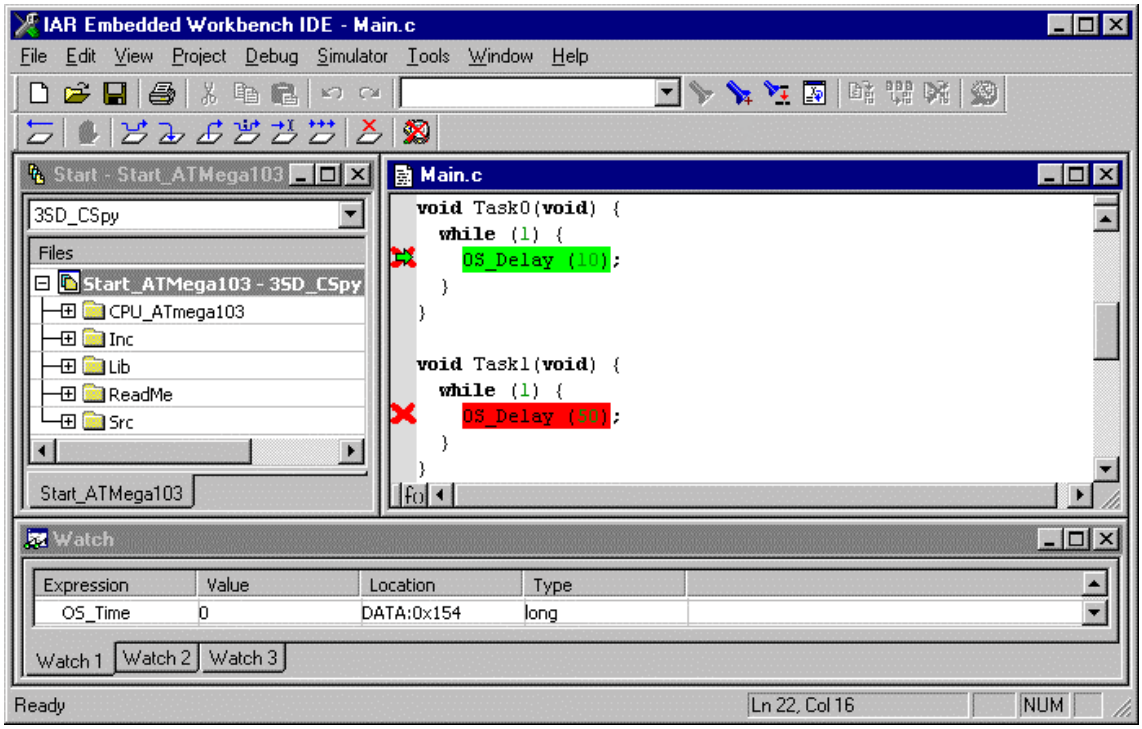
`OS_COM_Init()` called from `OS_InitHW()` is optional. It is required if embOSView shall be used. In this case it initializes the UART used for communication.
`OS_Start()` should be the last line in *main*, since it starts multitasking and does not return.



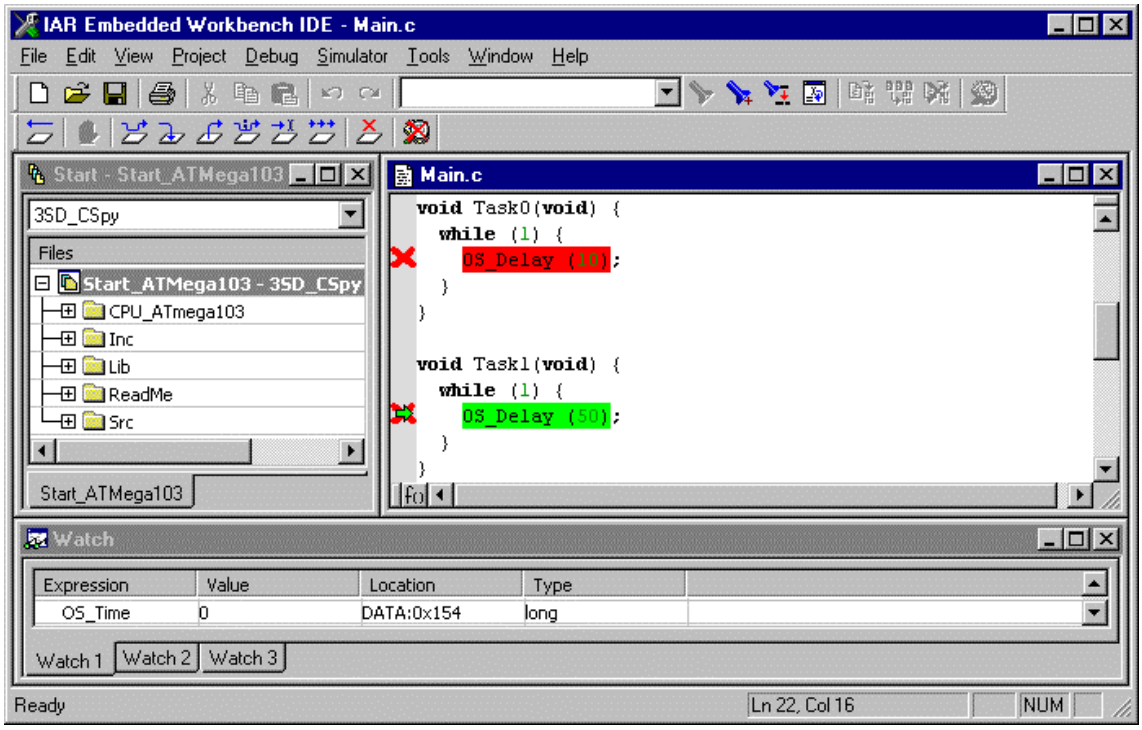
Before you step into `OS_Start()`, you should set breakpoints in the two tasks:



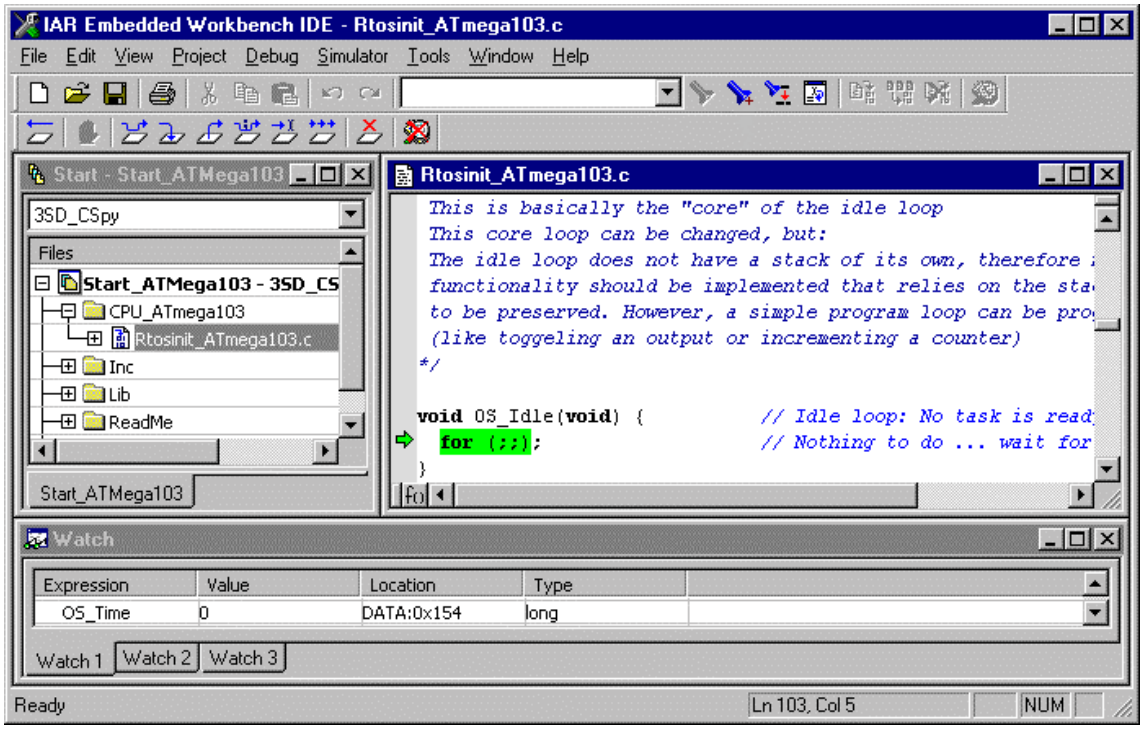
When you step over `OS_Start()`, the next line executed is already in the highest priority task created. (you may also step into `OS_Start()`, then stepping through the task switching process in disassembly mode). In our small start program, `Task0()` is the highest priority task and is therefore active.



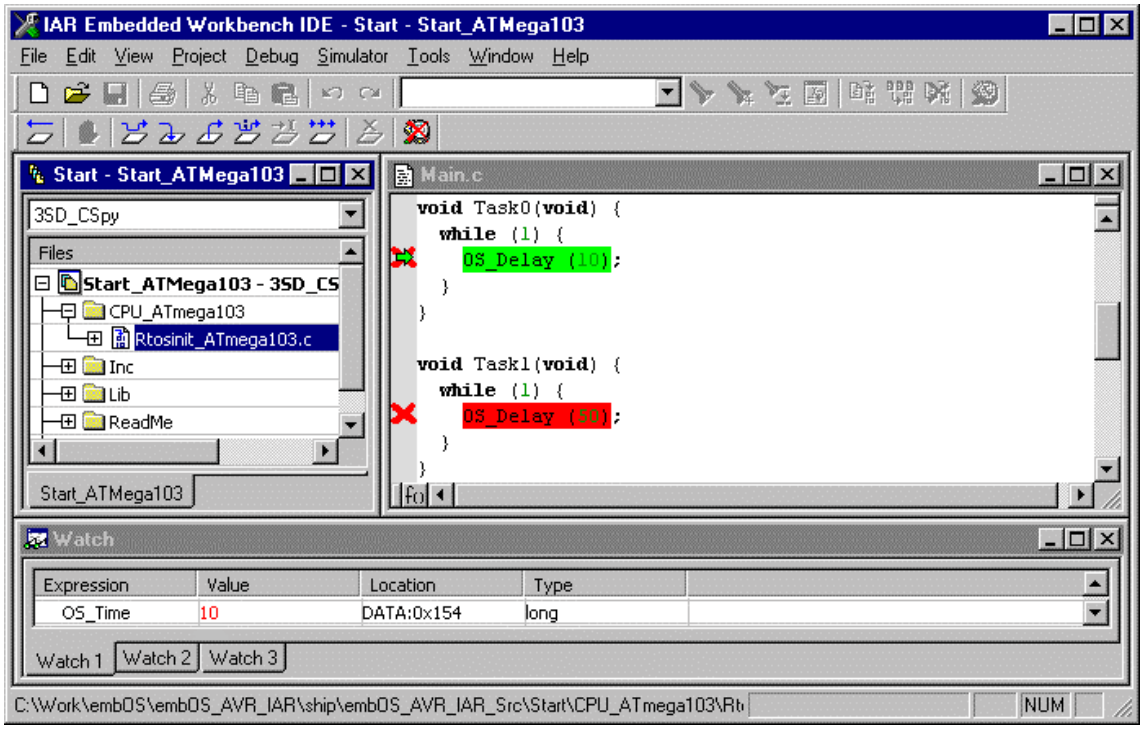
If you continue stepping, you will arrive in the task with the lower priority:



Continuing to step through the program, there is no other task ready for execution. ***embOS*** will suspend Task1 and switch to 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).
`OS_Idle()` is found in RTOSInit.c:



If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay. Coming from `OS_Idle()`, you should execute the 'Go' command to arrive at the highest priority task after its delay is expired. This can be seen at the system variable OS_Time:



3. Build your own application

To build your own application, you should start with the sample start project. 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 CPU specific subfolder CPU_*\.
  It contains hardware dependent initialization code for ***embOS*** timer and optional UART for embOSView.
- One ***embOS*** **library** from the Lib\ subfolder
- **OS_Error.c** from subfolder Src\
  The error handler is used if any library other than Release build library is used in your project.

When you decide to write your own startup code, please ensure that non initialized variables are initialized with zero, according to “C” standard. This is required for some ***embOS*** internal variables.
Your main() function has to initialize ***embOS*** by call of `OS_InitKern()` and `OS_InitHW()` prior any other ***embOS*** functions except `OS_IncDI()` are called.

3.2. Select a start project

embOS comes with one start project which includes different configurations for different output formats or debug tools. The start project was built and tested with ATmega103 CPU. For other ATMEL AVR or ATmega CPUs there may be modifications required in RTOSInit.c.

3.3. Add your own code

For your own code, you may add a new group to the project.
You should then modify or replace the main.c source file in the subfolder src\.

3.4. 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.
Therefore you have to select or replace the ***embOS*** library in your project or target:

- in the Lib group, exclude all libraries from build, except the one which should be used for your application.

Finally check project options about library mode setting according library mode used. Refer to chapter 4 about the library naming conventions to select the correct library and library mode specific define.

4. AT90 / ATMega specifics

4.1. Memory models

embOS was designed for the small memory model and CPU variants type 3 and type 5 that IAR's C-Compiler supports.
Smaller CPUs with a maximum of 64KB Flash (ATMega64 or smaller) are supported and require usage of different libraries.

4.2. Available libraries

embOS comes with different libraries. During development of your application you may use any of the debug type libraries, as those include error checks that are helpful during development.
Later you may change to the stack check version, which executes faster, as all other runtime error checks are disabled. The release library executes fastest as it does not include any error checks during runtime.

The files to use for ATmega CPUs variant 3 are:

Memorymodel	Library type	Library	define
Small	Release	rtos3S_R	OS_LIBMODE_R
Small	Stack-check	rtos3S_S	OS_LIBMODE_S
Small	Stack-check + Profiling	rtos3S_SP	OS_LIBMODE_SP
Small	Debug	rtos3S_D	OS_LIBMODE_D
Small	Debug + Profiling	rtos3S_DP	OS_LIBMODE_DP
Small	Debug + Profiling + Trace	rtos3S_DT	OS_LIBMODE_DT

The files to use for ATmega CPUs with 24bit code address range, variant 5 are:

Memorymodel	Library type	Library	define
Small	Release	rtos5S_R	OS_LIBMODE_R
Small	Stack-check	rtos5S_S	OS_LIBMODE_S
Small	Stack-check + Profiling	rtos5S_SP	OS_LIBMODE_SP
Small	Debug	rtos5S_D	OS_LIBMODE_D
Small	Debug + Profiling	rtos5S_DP	OS_LIBMODE_DP
Small	Debug + Profiling + Trace	rtos5S_DT	OS_LIBMODE_DT

The files to use for ATmega CPUs with 24bit code address range, variant 6 are:

Memorymodel	Library type	Library	Define
Small	Release	rtos6S_R	OS_LIBMODE_R
Small	Stack-check	rtos6S_S	OS_LIBMODE_S
Small	Stack-check + Profiling	rtos6S_SP	OS_LIBMODE_SP
Small	Debug	rtos6S_D	OS_LIBMODE_D
Small	Debug + Profiling	rtos6S_DP	OS_LIBMODE_DP
Small	Debug + Profiling + Trace	rtos6S_DT	OS_LIBMODE_DT

The files to use for ATmega CPUs with small Flash (ATmega64 or smaller) are:

Memorymodel	Library type	Library	define
Small	Release	rtos3SSF_R	OS_LIBMODE_R
Small	Stack-check	rtos3SSF_S	OS_LIBMODE_S
Small	Stack-check + Profiling	rtos3SSF_SP	OS_LIBMODE_SP
Small	Debug	rtos3SSF_D	OS_LIBMODE_D
Small	Debug + Profiling	rtos3SSF_DP	OS_LIBMODE_DP
Small	Debug + Profiling + Trace	rtos3SSF_DT	OS_LIBMODE_DT

When using IAR's workbench, please check the following points:

- The memory model and CPU variant is set as general project option
- One ***embOS*** library is part of your project (included in one group of your target)
- The appropriate define is set as compiler option for your project.

4.3. Distributed project files

The distribution of ***embOS*** contains one start workspace with different projects for the small memory model in the start subdirectory.
The projects contain configurations for any library type and one additional configuration that should be used for CSpy.
The configuration names reflect the selected library.
You should use these configurations to develop your application. Simply add new groups containing your own sources. This ensures, that all settings and files needed for ***embOS*** are always setup correctly.

5. Stacks

5.1. Stack address range

Because the ATMEL AT90 / ATmega has a 16-bit hardware stack-pointer, a stack has to be located in the lower 64kB of memory (0x0100 - 0xFFFF) The lowest addresses are reserved as special function registers and tiny area. You can not use memory outside this area as stack for a task.

5.2. System stack

The system stack is used for the following purposes:

- Normal stack during startup (until `OS_Start()` is called).
- ***embOS*** internal functions
- Software timer
- Stack for interrupt handler, when `OS_EnterIntStack()` is used.

The system stack is divided into two sections that are defined in the linker file or can be configured in the embedded workbench under “Project options | General | System”
RSTACK: The return stack is the segment where the CPUs stack pointer points to. This is used for storage of return addresses during subroutine call.
A good value for the size of the return stack is a minimum of 32 bytes, or 16 levels under project option settings.

CSTACK: Is the segment that is used for parameter passing. It is implemented by using CPUs Y-register as “stack pointer”.
A good value for the size of the CSTACK stack is a minimum of 100 bytes.

Bigger stacks are not a problem, of course.

5.3. Task stacks

Every task uses its own stack which has to be defined when the task is created. As the system stack, the task stack also is divided into two parts, using the CPUs stack pointer for return addresses and an additional emulated stack by using the Y-register as stack pointer for parameter and local variables.

6. Interrupts

6.1. What happens when an interrupt occurs?

- The CPU-core receives an interrupt request
- As soon as the interrupts are enabled and the processor finished a complete instruction, the interrupt is executed
- the CPU saves actual PC on the stack
- the CPU loads the specified instruction from the vector table. This should be a jump to the interrupt service routine (ISR)
- ISR: save registers
- ISR: user-defined functionality
- ISR: restore registers
- ISR: Execute RETI command, restoring PC, thus returning to the interrupted program
- For details, please refer to the Atmel / ATmega users manual.

6.2. Defining interrupt handlers in "C"

Routines preceded by the keywords `#pragma vector` save & restore the return address and return with RTI. ***embOS*** interrupt handler have to use `OS_CallISR()/ OS_CallNestableISR()`.

Example of an ***embOS*** interrupt handler

```
void OS_ISR_Tick_Handler(void);
void OS_ISR_Tick_Handler(void) {
  OS_TICK_Handle();
}

#pragma vector = TCC0_CCA_VECT
__interrupt void OS_ISR_Tick (void);
#pragma vector = TCC0_CCA_VECT
__interrupt void OS_ISR_Tick (void)  {
  OS_CallISR(OS_ISR_Tick_Handler);
}
```

The rules for an ***embOS*** interrupt handler are as follows:

- The ***embOS*** interrupt handler **must not define any local variables.**
- The ***embOS*** interrupt handler has to call `OS_CallISR()/ OS_CallNestableISR()`.
- **The interrupt handler must not perform any other operation, calculation or function call**. This has to be done by the local function called from `OS_CallISR()` or `OS_CallNestableISR()`.
- **For Atxmega embOS interrupt handlers must have the interrupt priority 1. All other interrupt handlers with higher interrupt priorities are zero latency interrupts which must not call any embOS API function.**

6.3. Interrupt-stack

Since the AT90 / ATmega has only one hardware stack pointer, every interrupt uses additional stack space on the stack of the current task.

IAR uses the hardware stack pointer and additional RAM pointed to by register Y as stack. As interrupts may occur any time, every task has to have enough stack for its own subroutine calls and all interrupts as well.
To reduce the stack space used for tasks, interrupt service routines may use the system stack for execution.
Stack switching is automatically done by `OS_CallISR()`/ `OS_CallNestableISR()` which calls internally `OS_EnterIntStack()`and `OS_LeaveIntStack()`.

7. Idle Mode

In Idle mode, the CPU is stopped, but UART, SPI, ADC, timers, comparators and interrupt system continue operation.
This may be used to save power consumption during idle times. Therefore you may place the SLEEP command in `OS_Idle()`. Any further interrupt will wake up the CPU and operation continues.

8. Technical data

8.1. Memory requirements

These values are neither precise nor guaranteed but they give you a good idea of the memory-requirements. They vary depending on the current version of ***embOS***. The minimum ROM requirement for the kernel itself is about 1904 bytes.
The table below shows 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	17
Resource semaphore	4
Counting semaphore	2
Mailbox	11
Software timer	9

9. Files shipped with ***embOS***

Directory	File	Explanation
root	`*.pdf`	Generic API and target specific documentation
root	`Release.html`	Release notes of ***embOS*** MSP430
root	`embOSView.exe`	Utility for runtime analysis, described in generic documentation
Start\	`Start.eww`	Sample workspace for IAR workbench
Start\	`Start_*.ewp`	Start project for various CPUs
Start\	`Start_*.ewd`	Setup for CSpy debugger
Start\Inc\	`RTOS.h`	To be included in any file using ***embOS*** functions
Start\lib\	`rtos*.r90`	***embOS*** libraries
Start\Src\	`main.c`	Frame program to serve as a start
Start\Src\	`OS_Error.c`	embOS error handler used in stack check and debug builds
Start\CPU_*\	`RtosInit_*.c`	Target CPU specific hardware initialization; can be modified
Start\CPU_*\	`CSpy_*.mac`	Target CPU specific interrupt simulation setup macro for CSpy simulator

Any additional files shipped serve as example.

10. Index

I
Idle mode 17
Installation 5
Interrupts 15
Interrupt-stack 16

M
memory models 12
memory requirements 17

S
Sleep 17
Stacks 14
system stack 14

T
target hardware 17