embOS

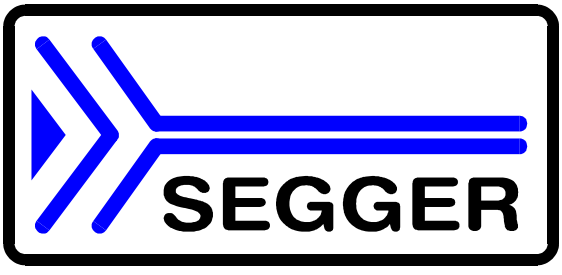
Real Time Operating System

CPU & Compiler specifics for

RENESAS M16C/80, M32C CPUs

and IAR compiler for M32C

Document Rev. 4



A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Contents

1. About this document

This guide describes how to use ***embOS*** for M16C/80 and M32C Real Time Operating System for the RENESAS M16C/80 and M32C series of microcontroller using IAR compiler for M32C and IAR's Embedded Workbench.

1.1. How to use this manual

This manual describes all CPU and compiler specifics for ***embOS*** using M16C/80 and M32C CPUs with IAR compiler. 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 workbench. If you have no experience using ***embOS***, you should follow this introduction, even if you do not plan to use C-SPY or IAR's Embedded Workbench, 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 M16C/80 and M32C using IAR compiler.

2. What's new?

- **Additional libraries delivered with *embOS***

Since version 3.50, libraries for 64bit floating point calculation are delivered with ***embOS*** for M32C.

- **Zero latency (fast) interrupts:**

Since version 3.82 of ***embOS*** for M32C, interrupt handling inside ***embOS*** was modified. Instead of disabling interrupts when ***embOS*** does atomic operations, the interrupt level of the CPU is set to a higher user definable level. Therefore all interrupts with a higher level can still be processed.

2.1. Update / Upgrade information

When you update / upgrade from an ***embOS*** version prior 3.82, you may have to change your interrupt handlers because of the new *Fast interrupt* support. All interrupt handlers using ***embOS*** functions have to run on priorities below the user definable priority limit which is initially set to 5. This limit may be changed by a cll of the new embOS function `OS_SetFastIntPriorityLimit()`.
Please read the chapter "Interrupts" in this manual.

3. Using *embOS* with IAR's Embedded Workbench

3.1. 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, keep all files in their respective sub directories. Make sure the files are not read only after copying.
If you received a zip-file, extract it to any folder of your choice, preserving the directory structure of the zip-file.

Assuming that you are using IAR's Embedded Workbench to develop your application, no further installation steps are required. You will find prepared sample start projects for M16C80 and M32C CPUs, 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 will not 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 without the workbench using batch files or a make utility without any problem.

3.2. First steps

After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received ready to go sample start workspaces and projects for M16C/80 and M32C CPUs and it is a good idea to use this as a starting point of all your applications.

Your ***embOS*** distribution contains two different folders with start projects:

- 'Start_MC80' for M16C/80 CPUs
- 'Start_M32C' for M32C CPUs.

Every folder contains everything you need for the specific CPU. As long as only one CPU is used for your applications, there is no need to copy both of them.

To get your new application running, you should proceed as follows.

For M16C/80 targets you should:

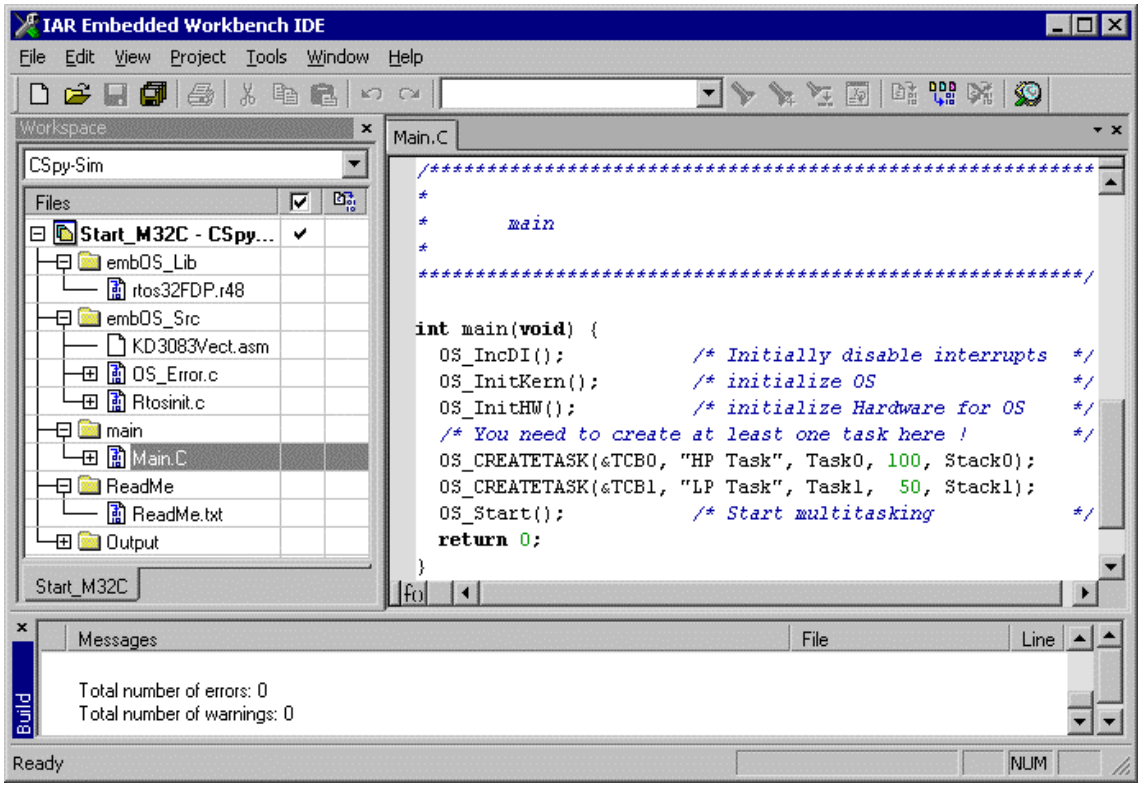
- Create a work directory for your application, for example c:\work
- Copy the whole folder 'Start_MC80' from your ***embOS*** distribution into your work directory.
- Clear the read only attribute of all files in the new 'Start_MC80'-folder in your working directory.
- Open the folder 'Start_MC80'.
- Open the start workspace 'Start_MC80.eww'. (e.g. by double clicking it)
- Select a configuration
- Build the start project

For M32C targets you should:

- Create a work directory for your application, for example c:\work
- Copy the whole folder 'Start_M32C' from your ***embOS*** distribution into your work directory.
- Clear the read only attribute of all files in the new 'Start_M32C'-folder in your working directory.
- Open the folder 'Start_M32C'.
- Open the start workspace 'Start_M32C.eww'. (e.g. by double clicking it)
- Select a configuration
- Build the start project

Further examples in this manual show the start project for M32C CPU which is found in the 'Start_M32C' folder. The M16C80 start project is similar and looks the same.

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



For latest information you should open the ReadMe.txt which is part of your project.

3.3. 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.
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.

```
/*---------------------------------------------------------------------
File    : Main.c
Purpose : Skeleton program for OS
--------  END-OF-HEADER  ---------------------------------------------
*/

#include "RTOS.h"

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

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

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

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

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

4. Using debugging tools to debug the application

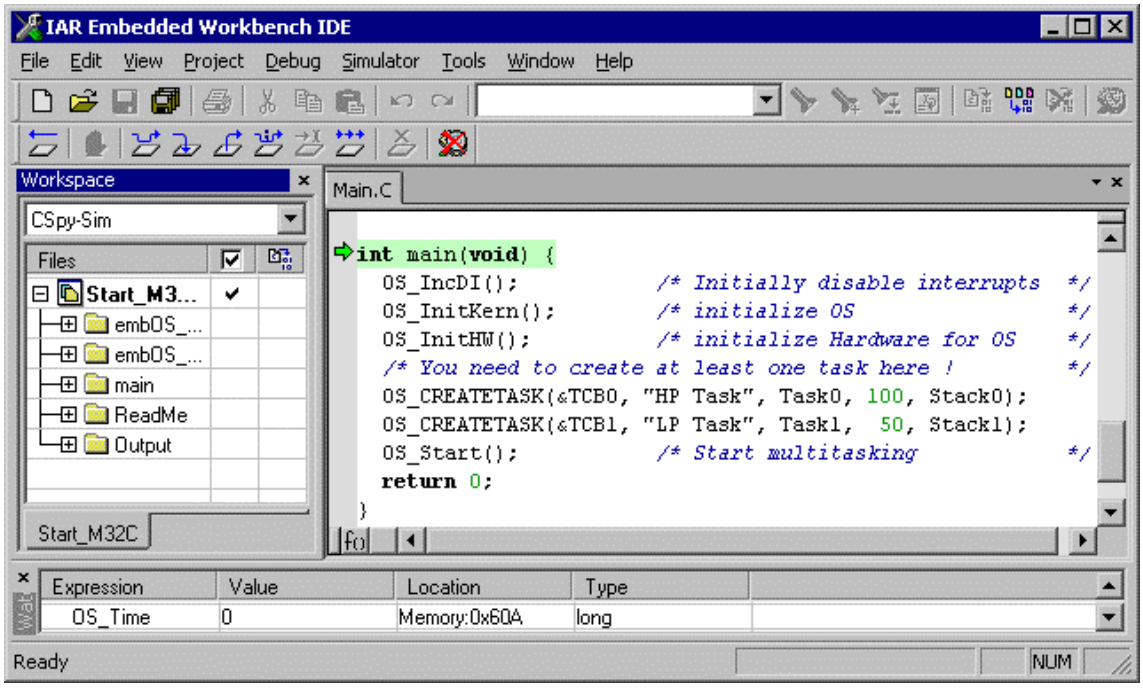
The *embOS* start project contains configurations which are already setup for the following debugging tools:

- IAR's simulator CSpy. This configuration is named "CSpy-Sim".
- RENESAS's ROM Monitor driver KD308/KD3083 or IARs CSpy. This configuration is named "ROM-Monitor" and may also be used with IAR CSpy in serial or USB ROM monitor mode or RENESAS KD308/KD3083.
- RENESAS's in circuit emulator PD308 or newer. This configuration is named "Emulator"

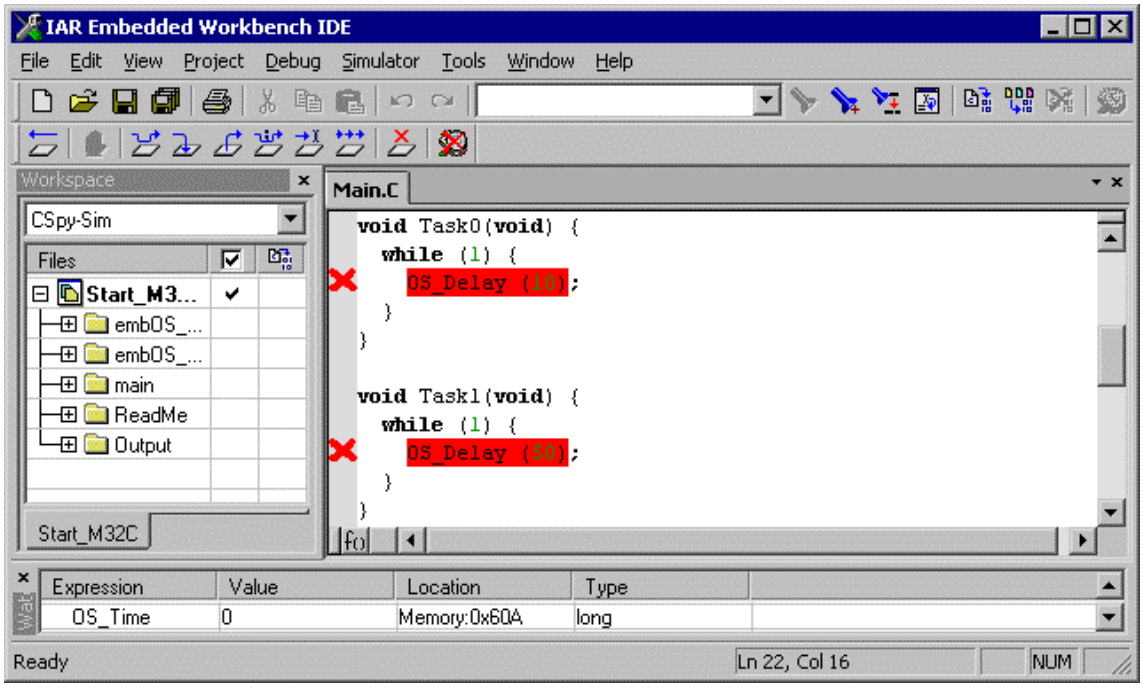
All these configurations are prepared to produce the appropriate output files required by your debugger.
The E8a and E8 debugging tool from RENESAS can be used without any problem. The linker files may have to be modified when used with E8a or E8, because the debugger occupies 256bytes of RAM and 2KBytes of ROM.
The following chapters describe a sample session based on our sample application main using CSpy Simulator.

4.1. Using IAR's C-Spy simulator

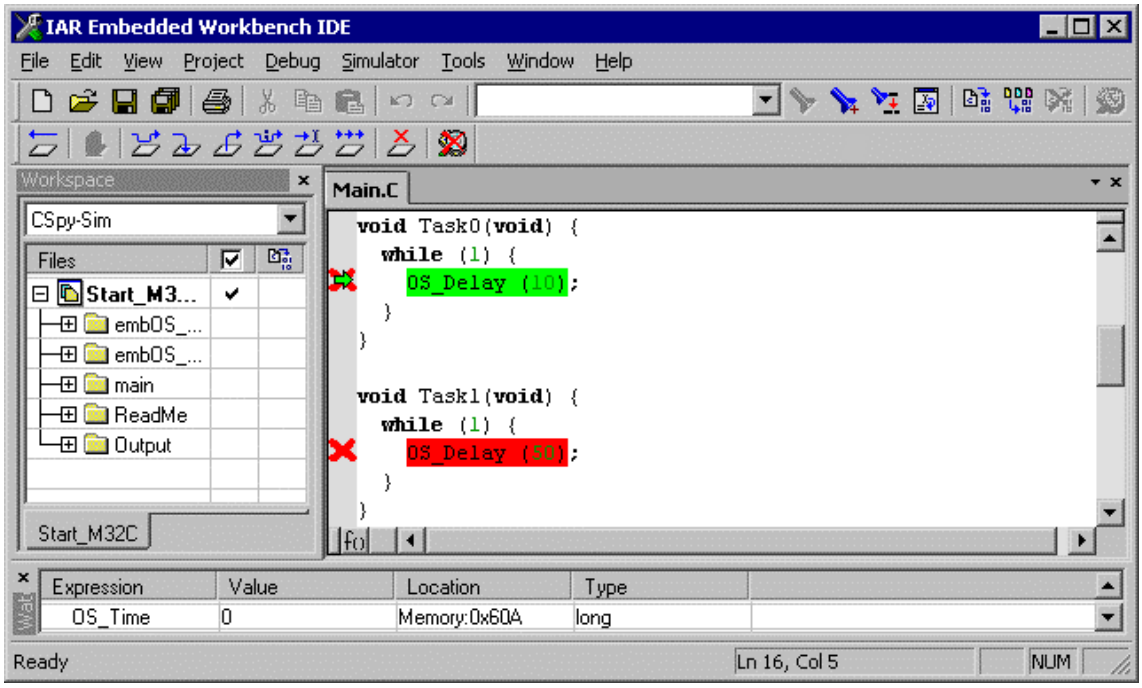
When starting C-Spy simulator after building the C-Spy target, you will usually see the main function, or you may look at the startup code and have to set a breakpoint at main. Now you can step through the program.
`OS_IncDI()` initially disables interrupts and prevents OS_InitKern() from reenabling them.
`OS_InitKern()` initializes *embOS* -Variables. As this function is part of the *embOS* library, you may step into it in disassembly mode only.
`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. As simulators usually can not simulate UART operations, OS_UART may be defined as (-1) to disable UART initialization and communication.
`OS_Start()` should be the last line in main, since it starts multitasking and does not return.



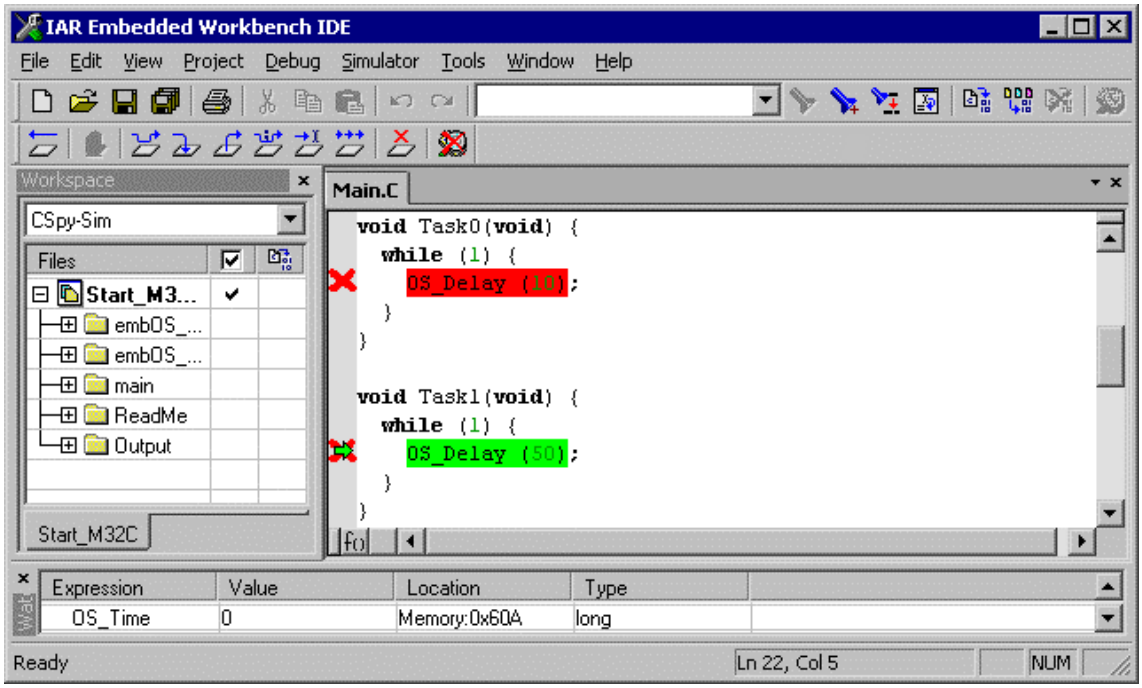
Before you step into `OS_Start()`, you should set breakpoints in the two tasks:



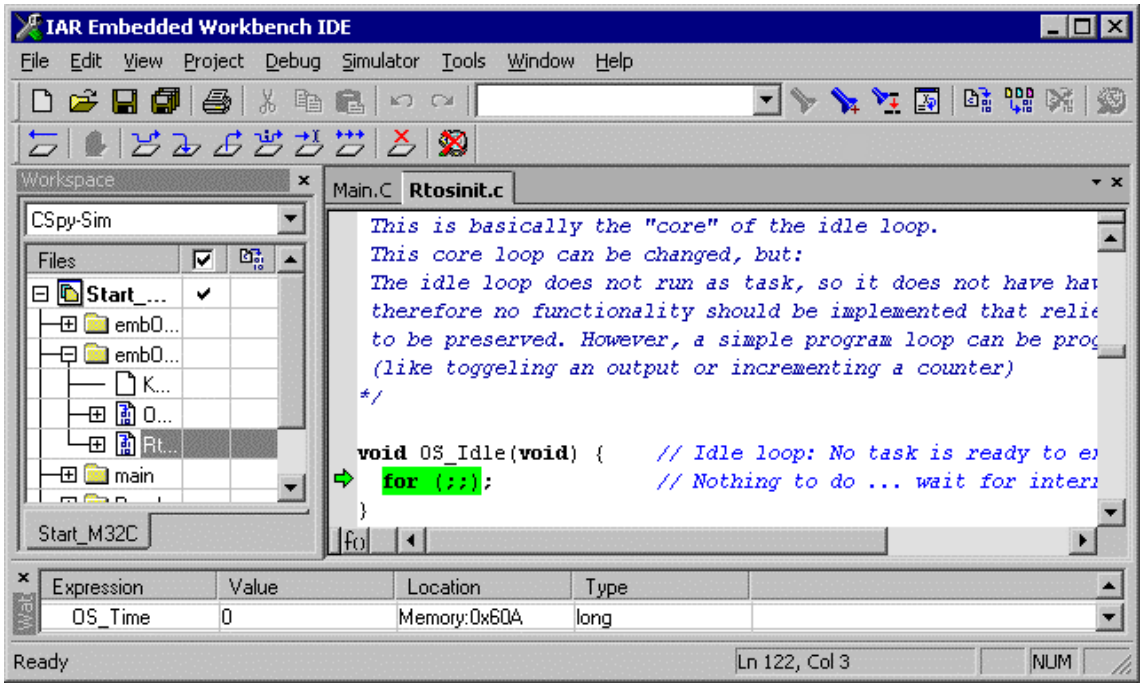
When you step over `OS_Start()`, the next line executed is already in the highest priority task created. (you may also step into `OS_Start()`, then stepping through the task switching process in disassembly mode). In our small start program, `Task0()` is the highest priority task and is therefore active.



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

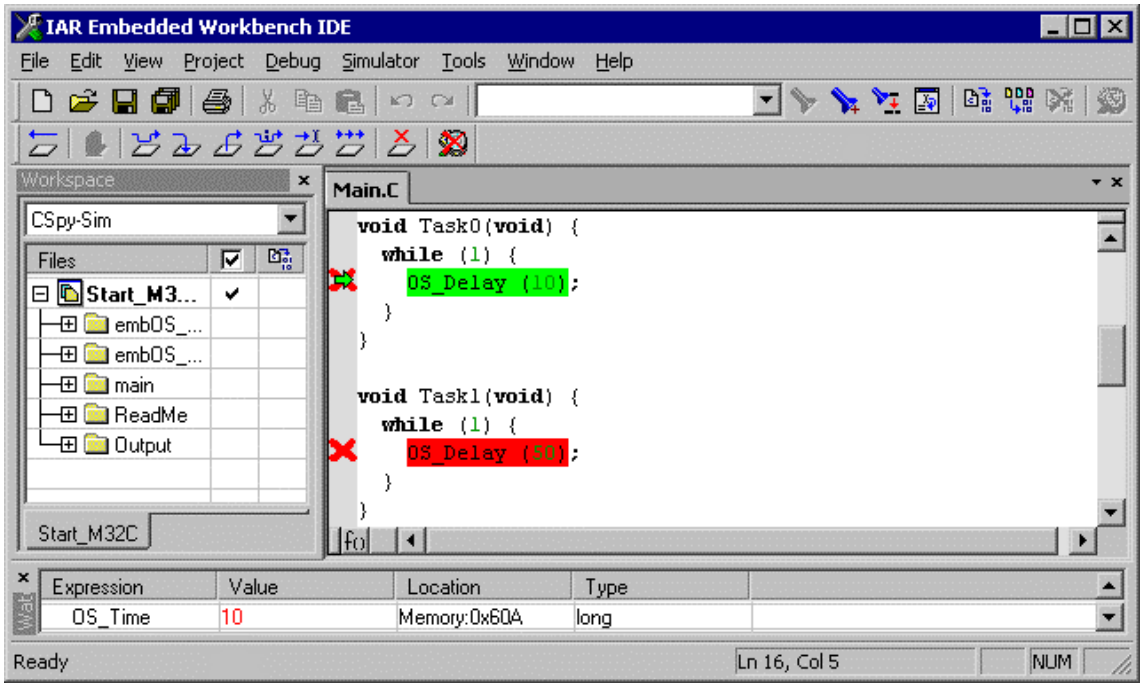


Continuing to step through the program, there is no other task ready for execution. ***embOS*** will suspend Task1 and switch to the idle-loop, which is an endless loop which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).
`OS_Idle()` is found in RTOSInit.c
You will arrive there, when you step through the task switching process in disassembly mode:



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.

The watch window shows the system variable `OS_Time`, which shows how much time has expired in the target system.



4.2. Using ROM Monitor configuration

The distribution of ***embOS*** for M16C80 / M32C is prepared for usage of KD308/KD3083 debugging software as well as IAR's CSpy ROM monitor driver. Depending on the ROM monitor and CPU used, the ROM monitor needs UART interrupt vectors which point to ROM monitor internal interrupt functions.

For M16C80 CPUs, these vectors are defined in '`KD308Vect.asm`' which is enabled in the configuration "ROM-Monitor" in the Start_MC80 project. It is setup for KD308 or CSpy ROM monitor usage.
For M32C CPUs, these vectors are defined in '`KD3083Vect.asm`' which is enabled in the configuration "ROM-Monitor" in the Start_M32C project which is setup for KD3083 or CSpy ROM monitor usage.
As ROM monitors usually communicate via UART1 of the target CPU, this UART can not be selected as communication port for embOSView or for your application.
Note also, that the variable interrupt vector table has to be located below the highest target CPUs flash sector, which is used by the ROM Monitor.
Default linker files should be set up accordingly.
Depending on the ROM monitor used on M32C CPUs, the interrupt vectors for UART might not be required, because some CPUs may be configured to use the debug interrupt for communication.

Problem with ROM monitor running ***embOS*** application:

When ROM monitor stopped at a breakpoint, it may happen, that any interrupt activates a task switch while stepping through the program, as interrupts are enabled during stepping. This task switch can not be handled by ROM monitor and it crashes.
To overcome this problem, you may open the register window and set interrupt priority (IPL) to 6 immediately after the breakpoint was reached. This enables stepping without any task switches, as all ***embOS*** interrupts normally run with lower priorities.

4.3. Interrupt vector definition file KD308Vect.asm

This file is used for M16C80 CPUs which are already programmed with a RENESAS ROM monitor. The file defines two interrupt vectors for UART1 used by ROM Monitor. When not using KD308 or CSpy in ROM monitor mode, this interrupt vector definition file is not required, as ***embOS*** interrupts are defined in 'C'-source code.
Both vectors for Rx- and Tx- interrupt point to the same address.

Important:

Ensure, that this file is linked to your application, when needed for ROM-Monitor.
Check the project options for assembler AM32C, code generation:
'Make a LIBRARY module' option has to be unchecked. Otherwise the linker may optimize those vectors away, as they are not referenced by your application.

4.4. Interrupt vector definition file KD3083Vect.asm

This file is used for M32C CPUs which are already programmed with a RENESAS ROM monitor that requires debug communication interrupts on UART. The file defines two interrupt vectors for UART1 used by ROM Monitor. When not using KD3083 or CSpy in ROM monitor mode, this interrupt vector definition file is not required, as ***embOS*** interrupts are defined in 'C'-source code.
Both vectors for Rx- and Tx- interrupt point to the same address.

Important:

Ensure, that this file is linked to your application, when needed for ROM-Monitor.
Check the project options for assembler AM32C, code generation:
'Make a LIBRARY module' option has to be unchecked. Otherwise the linker may optimize those vectors away, as they are not referenced by your application.

4.5. Using PD308 or other in circuit emulator

The standard distribution of ***embOS*** for M16C80 / M32C and IAR compiler contains a configuration for RENESAS's in circuit emulator.
This configuration is named "Emulator" and it produces an 'X30' output file with debug information which may be loaded into RENESAS's in circuit emulator to debug the application.

4.6. Using E8 or E8a debugging tool

RENESAS's E8a debugging tool can be used for M32C CPUs without problems.
The standard distribution of ***embOS*** for M16C80 / M32C and IAR compiler does not contain a configuration for the E8a, but an existing configuration can easily be changed to use E8a as debugging tool.
The linker file may have to be modified for E8a support, because E8a occupies 256bytes of RAM and 2KBytes of ROM in the target CPUI.

4.7. Common debugging hints

For debugging your application, you should use a debug build, e.g. use the debug build libraries in your projects if possible. The debug build contains additional error check functions during runtime.
When an error is detected, the debug libraries call `OS_Error()`, which is defined in the separate file `OS_Error.c`.
Using an emulator you should set a breakpoint there. The actual error code is assigned to the global variable `OS_Status`. The program then waits for this variable to be reset. This allows to get back to the program-code that caused the problem easily: Simply reset this variable to 0 using your in circuit-emulator, and you can step back to the program sequence causing the problem. Most of the time, a look at this part of the program will make the problem clear.
How to select an other library with debug code for your projects is described later on in this manual.

5. 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.

5.1. 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 subfolder Src\.
  It contains hardware dependent initialization code for ***embOS*** timer and optional UART for embOSView.
- **OS_Error.c** from subfolder Src\.
  It contains the ***embOS*** runtime error handler `OS_Error()` which is used in stack check or debug builds.
- One ***embOS*** **library** from the Lib\ subfolder
- **KD30*Vect.asm** from subfolder Src\ for ROM monitor targets
  If configuration should be built for ROM monitor usage, interrupt vectors for debug UART may have to be defined in your project. This is done in KD30*Vect.asm.

When you decide to write your own startup code, 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.

5.2. 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 various M16C80 and M32C CPUs. For your specific CPU variant there may be modifications required.

5.3. 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\.

5.4. Change memory model or library mode

For your application you may have to choose an other memory-model. 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 replace the ***embOS*** library in your project or target:

- Build a new group for the library an add it to the selected target.
- Add the appropriate library from the Lib-subdirectory to your new group.
- Remove the previous library group from your target.

Finally check the project options about target CPU data / memory model settings and compiler settings according the selected *embOS* library mode used. Refer to chapter 6 about the library naming conventions to select the correct library.

6. M16C/80 and M32C specifics

6.1. Memory models

embOS supports all the memory models that IAR' s C-Compiler supports.
For the M16C/80 and M32C, 3 memory models are available:

Model	Code	Data
Near	far (24 bits always)	near (16 bits)
Far	far (24 bits always)	far (24 bits, 64K segments)
Huge	far (24 bits always)	huge (24 bits)

6.2. Available libraries

The files to use depend on target CPU, memory model and library type used.
The library files are located in the subfolder 'Lib' in the CPU specific start project folder.
The CPU type selection and memory model settings for your target application have to confirm to the library used in your application.

The naming convention for library files is as follows:
RTOS<CPUFAMILY><MEMORYMODEL><FLOAT><LIBRARYTYPE>.r48

<CPUFAMILY> specifies the CPU family:
- **80** for M16C/80 CPUs
- **32** for M32C CPUs

<MEMORYMODEL> specifies the memory model:
- **N** for Near memory model
- **F** for Far memory model
- **H** for Huge memory model

<FLOAT> specifies the memory model:
- **D** for 64bit floating point calculation.
- **left blank** for standard 32bit floating point calculation.

<LIBRARYTYPE> specifies the type of *embOS* -library:
- **R** stands for Release build library.
- **S** stands for Stack check library, which performs stack checks during runtime.
- **SP** stands for Stack check and Profiling library, which performs stack checking and additional runtime (Profiling) calculations
- **D** stands for Debug library which performs error checking during runtime.
- **DP** stands for Debug and Profiling library which performs error checking and additional Profiling during runtime.
- **DT** stands for Debug and Trace library which performs error checking and additional Trace functionality during runtime.

Example:

RTOS80NSP.r48 is the *embOS* library for **M16C/80** CPU family in **N**ear memory model, 32bit (standard) floating point calculation, with **S**tack check and **P**rofiling functionality. It is located in the Start_MC80\lib\ subdirectory.

6.3. CPU specific settings

embOS may be used with any M16C80 / M32C CPU variant. Our start projects are set up for “generic” CPU. You should select your specific CPU as project option.

7. Stacks

7.1. Task stack for M16C/80 and M32C

Every *embOS* task has to have its own stack. Task stacks can be located in any RAM memory location.
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 M16C/80 and M32C, this minimum stack size is about 50 bytes in the far memory model.

7.2. System stack for M16C/80 and M32C

The system stack size required by *embOS* is about 30 bytes (60 bytes in. profiling builds) 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. We recommend at least a minimum of 128 bytes.
embOS uses IARs CSTACK as system stack.
The size of the system stack may be set up as project option or can be defined in the link file as _CSTACK_SIZE.

7.3. Interrupt stack for M16C/80 and M32C

The M16C/80 and M32C has been designed with multitasking in mind; it has 2 stack-pointers, the USP and the ISP. The U-Flag selects the active stack-pointer. During execution of a task or timer, the U-flag is set thereby selecting the user-stack-pointer. If an interrupt occurs, the M16C/80 and M32C clears the U-flag and switches to the interrupt-stack-pointer automatically this way. The ISP is active during the entire ISR (interrupt service routine). This way, the interrupt does not use the stack of the task and the stack-size does not have to be increased for interrupt-routines. Additional stack-switching as for other CPUs is therefore not necessary for the M16C/80 and M32C.
IAR defines the interrupt stack as ISTACK in the linker files. The size of the interrupt stack is defined in the link-file as _ISTACK_SIZE or may be set up as project option. (We recommend at least 192 bytes)

7.4. Stack specifics of the RENESAS M16C/80 and M32C family

Because the stack-pointer of M16C/80 and M32C CPUs can address the entire memory area, stacks can be located anywhere in RAM. For performance reasons you should try to locate stacks in fast RAM.

8. Interrupts

8.1. What happens when an interrupt occurs?

- The CPU-core receives an interrupt request
- As soon as the interrupts are enabled and the processor interrupt priority level is below or equal to the interrupt priority level, the interrupt is executed
- the CPU switches to the Interrupt stack
- the CPU saves PC and flags on the stack
- the IPL is loaded with the priority of the interrupt and interrupts are disabled
- the CPU jumps to the address specified in the vector table for the interrupt service routine (ISR)
- ISR : save registers
- ISR : user-defined functionality
- ISR : restore registers
- ISR: Execute REIT command, restoring PC, Flags and switching to User stack
- For more details, refer to the RENESAS users manual.

8.2. Defining interrupt handlers in "C"

Routines defined with the keyword interrupt automatically save & restore the registers they modify and return with REIT.
For a detailed description on how to define an interrupt routine in "C", refer to the IAR C-Compiler's user's guide.

Example

"Simple" interrupt-routine

```
#pragma vector= (13)
__interrupt void IntHandlerTimerA1(void) {
  IntCnt++;
}
```

8.3. Interrupt-stack

Since the M16C/80 and M32C 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.

8.4. Zero latency interrupts with M16C80/M32C CPUs

Instead of disabling interrupts when ***embOS*** does atomic operations, the interrupt level of the CPU is set to a higher user definable level. Therefore all interrupts with higher levels can still be processed.
These interrupts are named ***Zero latency interrupts***.
The default level limit for zero latency interrupts is set to 5, meaning, any interrupt with level 6 or 7 is never disabled and can be accepted anytime.
You must not execute any *embOS* function from within a *Zero latency interrupt* function.

8.5. Interrupt priorities with *embOS* for M16C80/M32C CPUs

With introduction of *Zero latency interrupts*, interrupt priorities useable by the application are divided into two groups:

- Low priority interrupts with priorities from 1 to a user definable priority limit. These interrupts are called ***embOS*** interrupts.
- High priority interrupts with priorities above the user definable priority limit. These interrupts are called ***Zero latency interrupts***.

Interrupt handler functions for both types have to follow the coding guidelines described in the following chapters.
The priority limit between ***embOS*** interrupts and Zero latency interrupts can be set at runtime by a call of the function `OS_SetFastIntPriorityLimit()`.

8.6. Interrupt latency

With ***embOS*** for M32C, the interrupt latencies are kept as small as possible, because high priority interrupts are never locked by the operating system.
Because the CPU automatically disables all interrupts when accepting an interrupt, the interrupt latency for interrupts with higher priority can not be zero. The interrupt handler has to re-enable interrupts by setting the I-Flag. Using ***embOS***, this is done by a call of the function `OS_EnterNestableInterrupt()` or `OS_EnterInterrupt()`.

Differences between OS_EnterInterrupt() and OS_EnterNestableInterrupt()

OS_EnterInterrupt() shall be used for an interrupt that may use ***embOS*** functions and runs on low priority (below the zero latency priority limit), but shall not be interrupted by other low priority interrupts.
OS_EnterInterrupt() sets the IPL of the CPU up to the zero latency priority limit and the re-enables interrupts.
OS_EnterNestableInterrupt() shall be used for an interrupt that may use ***embOS*** functions and runs on low priority (below the zero latency priority limit), but may be interrupted by other interrupts with higher priority. On entry, the IPL remains unchanged and interrupts are re-enabled.

8.7. OS_SetFastIntPriorityLimit(): Set the interrupt priority limit for Zero latency interrupts

The interrupt priority limit for Zero Latency interrupts is set to 5 by default. This means, all interrupts with higher priority from 6 to 7 will never be disabled by ***embOS***.

Description

OS_SetFastIntPriorityLimit() is used to set the interrupt priority limit between Zero latency interrupts and lower priority ***embOS*** interrupts.

Prototype

```
void OS_SetFastIntPriorityLimit(unsigned int Priority)
```

Parameter	Meaning
`Priority`	The highest value useable as priority for ***embOS*** interrupts. All interrupts with higher priority are never disabled by ***embOS***. Valid range: 1 <= `Priority` <= 7

Return value

NONE.

Add. information

To disable Zero latency interrupts at all, the priority limit may be set to 7 which is the highest interrupt priority for interrupts.
To modify the default priority limit, `OS_SetFastIntPriorityLimit()` should be called before ***embOS*** was started.

9. STOP / WAIT Mode

Usage of the wait instruction is one possibility to save power consumption during idle times. If required, you may modify the `OS_Idle()` routine, which is part of the hardware dependent module RtosInit.c.

The stop-mode works without a problem; however the real-time operating system is halted during the execution of the stop-instruction if the timer that the scheduler uses is supplied from the internal clock. With external clock, the scheduler keeps working.

10. Technical data

10.1. Memory requirements

These values are neither precise nor guaranteed but they give you a good idea of the memory-requirements. They vary depending on the ***embOS*** library. The values in the table are for the far memory model and release build library.

Short description	ROM [byte]	RAM [byte]
Kernel	approx.1600	37
Event-management	< 200	---
Mailbox management	< 550	---
Single-byte mailbox management	< 300	---
Resource-semaphore management	< 250	---
Timer-management	< 250	---
Add. Task	---	28
Add. Counting Semaphore	---	6
Add. Mailbox	---	16
Add. Timer	---	14
Power-management	---	---

11. Files shipped with *embOS* for IAR M32C compiler

Directory	File	Explanation
root	`*.pdf`	Generic API and target specific documentation
root	`Release.html`	Release notes of ***embOS*** M32C
root	`embOSView.exe`	Utility for runtime analysis, described in generic documentation
Start_MC80\	`Start_MC80.eww`	Start project for M16C/80 CPUs
Start_M32C\	`Start_M32C.eww`	Start project for M32C CPUs

Each start project folder contains the following files:

Directory	File	Explanation
	`CSpy.mac`	CSpy macro for hardware timer simulation
	`Readme.txt`	Latest information about ***embOS*** M32C
	`*.ewp`	Start project file for embedded workbench
Inc\	`RTOS.h`	To be included in any file using ***embOS*** functions
	`*.ewp`	Start project file for embedded workbench
Lib\	`Rtos*.r48`	***embOS*** libraries
Src\	`main.c`	Frame program to serve as a start
Src\	`RtosInit.c`	To be compiled & linked with your program, initializes the hardware, can be modified
Src\	`OS_Error.c`	***embOS*** error handler used in stack check or debug builds

12. Index