embOS

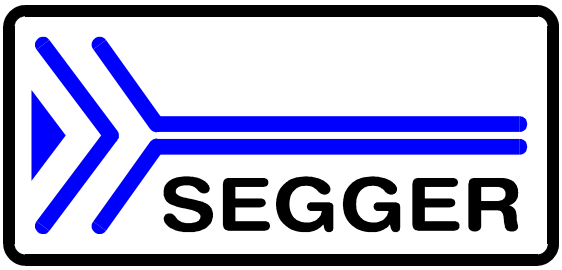
Real Time Operating System

CPU & Compiler specifics for

Renesas K/0, K/0S & K0R cores using

IAR Embedded Workbench

Document Rev. 6



A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Contents

1. About this document

This guide describes how to use the ***embOS*** Real Time Operating System for the Renesas K/0, K/0S and K0R series of microcontroller using *IAR Embedded Workbench*.

1.1. How to use this manual

This manual describes all CPU and compiler specifics for ***embOS*** using Renesas K/0 & K/0S or K0R 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 Renesas K/0, K/0S and K0R 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* 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 *IAR Embedded Workbench* for your application development in order to become familiar with ***embOS.***

If for some reason you will not work with the *IAR Embedded Workbench*, you should:
Copy either all or only the library-file that you need to your work-directory. Also copy the hardware initialization file RTOSInit_*.c found in the CPU specific subdirectories 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 *IAR Embedded Workbench*, it may be used without the project manager using batchfiles or a make utility without any problem.

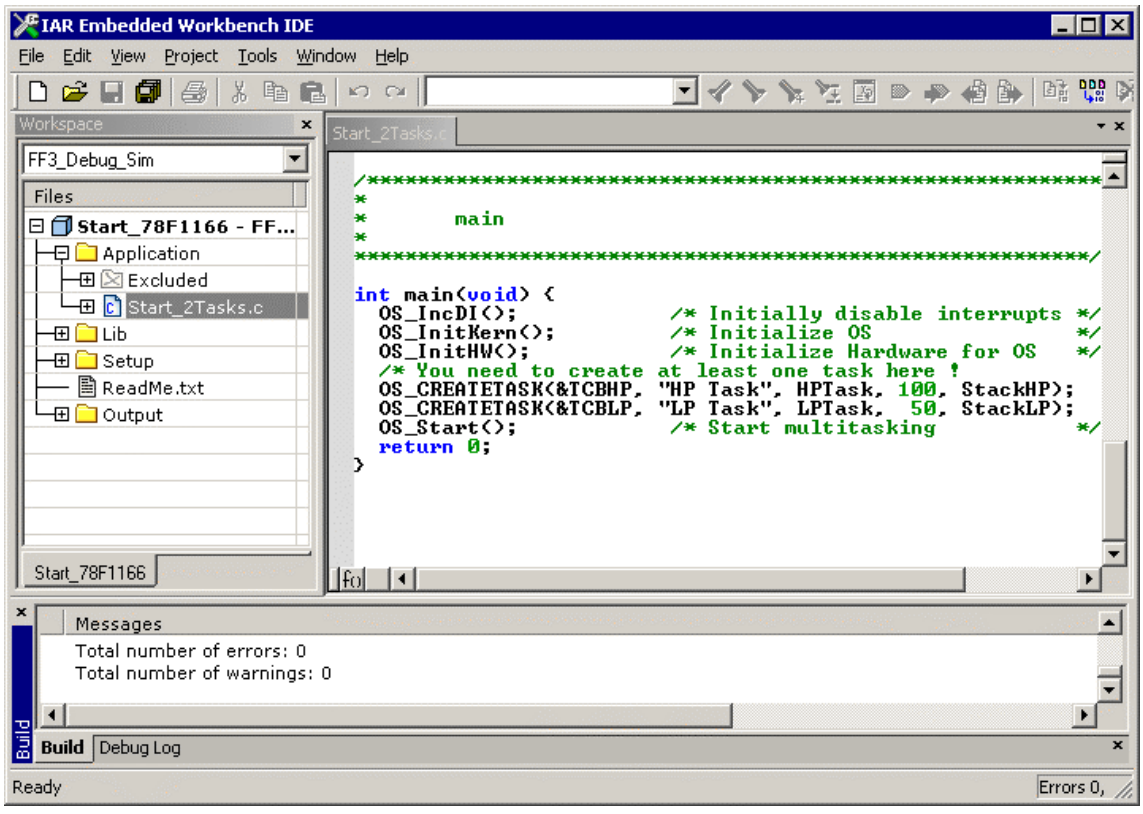
2.2. First steps

After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received a ready to go sample workspace which contains two start projects 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 one sample workspace found in any of the CPU specific sub-directories of the Start\Boardsupport\ folder with your *IAR Embedded Workbench* (e.g. by double clicking it)
Select a configuration and then build he start project
Further documentation describes the configuration "FF3_Debug_Sim" form the project workspace "Start\BoardSupport\CPU_78F1166\Start_78F1166.eww" which is set up for the CSpy simulator.

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



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

2.2.1. The sample application Start_2Tasks.c

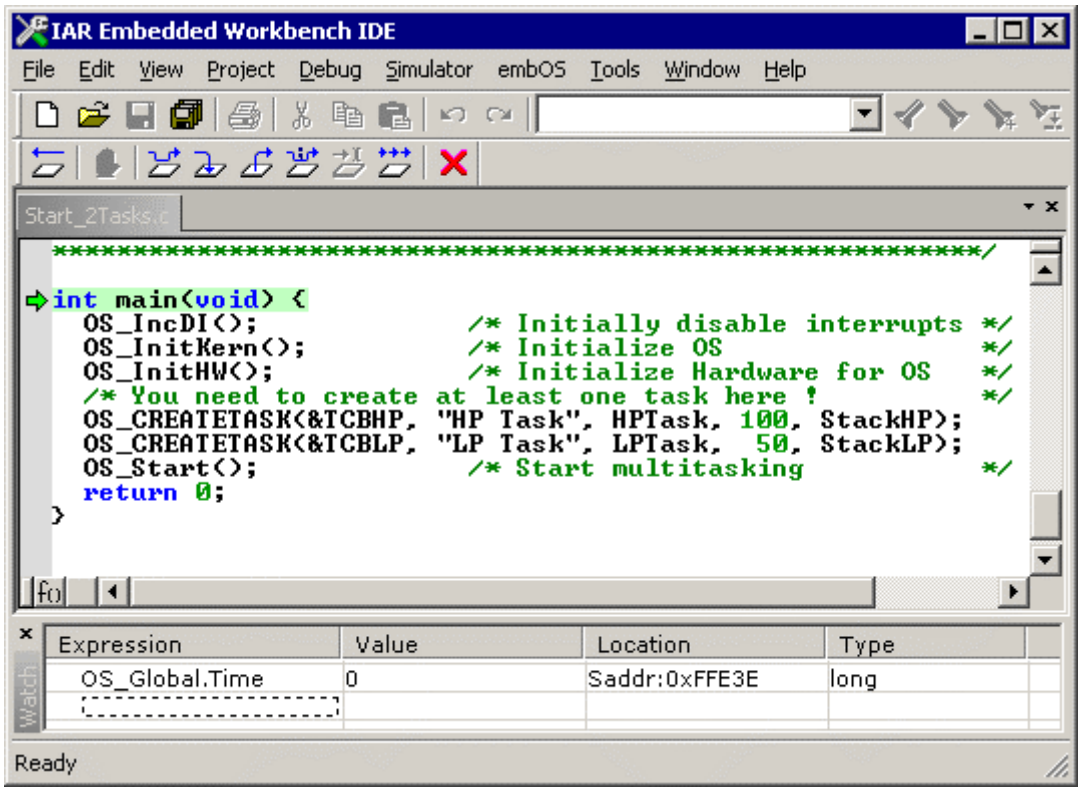
The following is a printout of the sample application Start_2Tasks.c. It is a good starting-point for your application. (Please note that the file actually shipped with your port of ***embOS*** may look slightly different from this one)
What happens is easy to see:
After initialization of ***embOS,*** two tasks are created and started.
The 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    : Start_2Tasks.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;
}
```

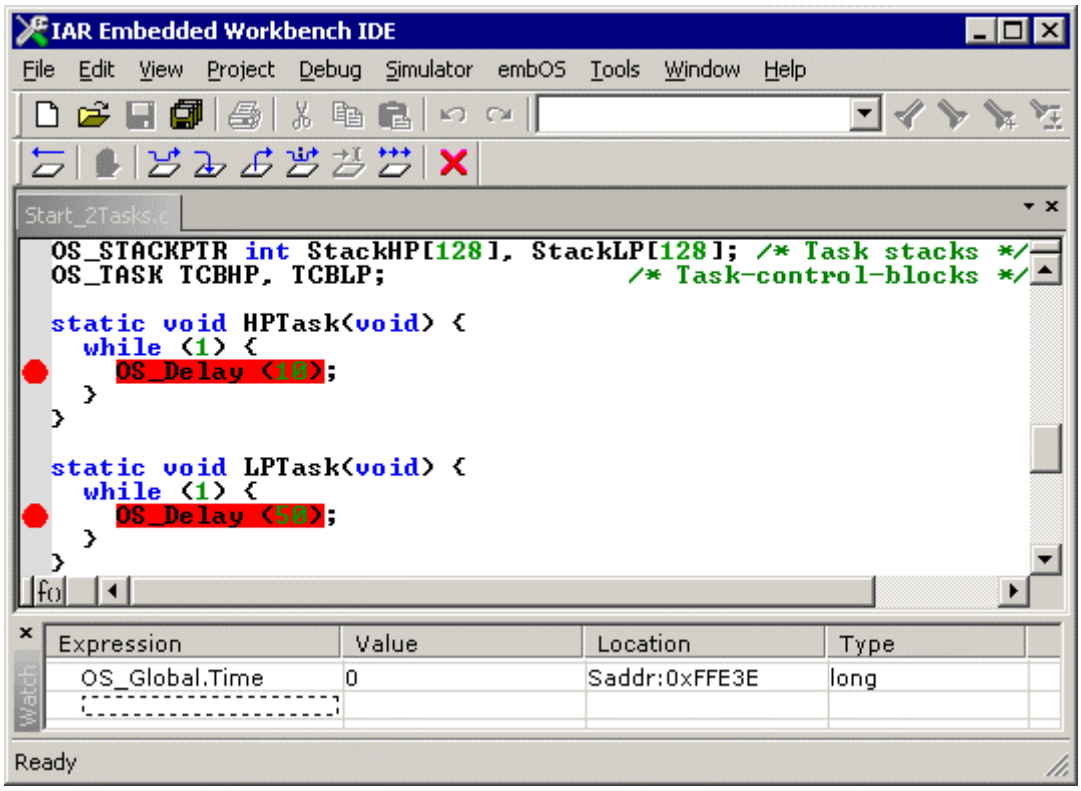
2.3. Stepping through the sample application using CSpy

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()` initially disables interrupts and prevents `OS_InitKern()` from re-enabling them. Interrupts are automatically re-enabled when `OS_Start()` is called.
`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 they were blocked by a previous call of `OS_IncDI()`.
`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.

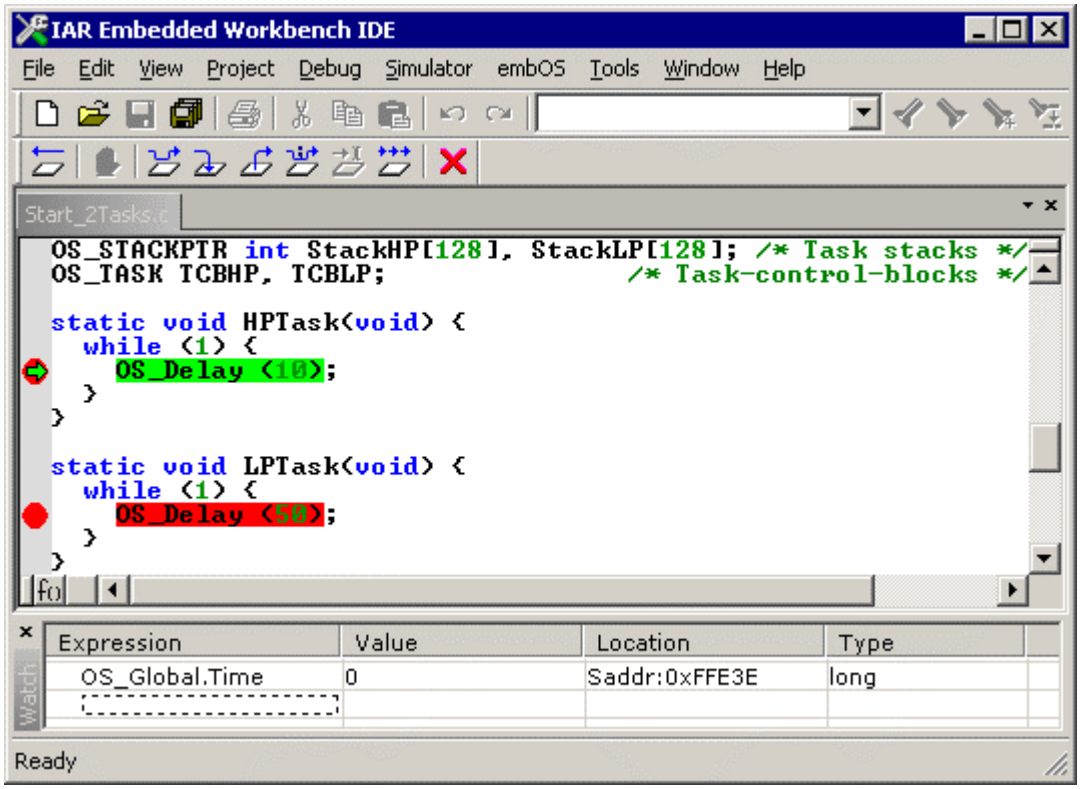
`OS_Start()` should be the last line in `main()`, since it starts multitasking and does not return. `OS_Start()` automatically enables interrupts.



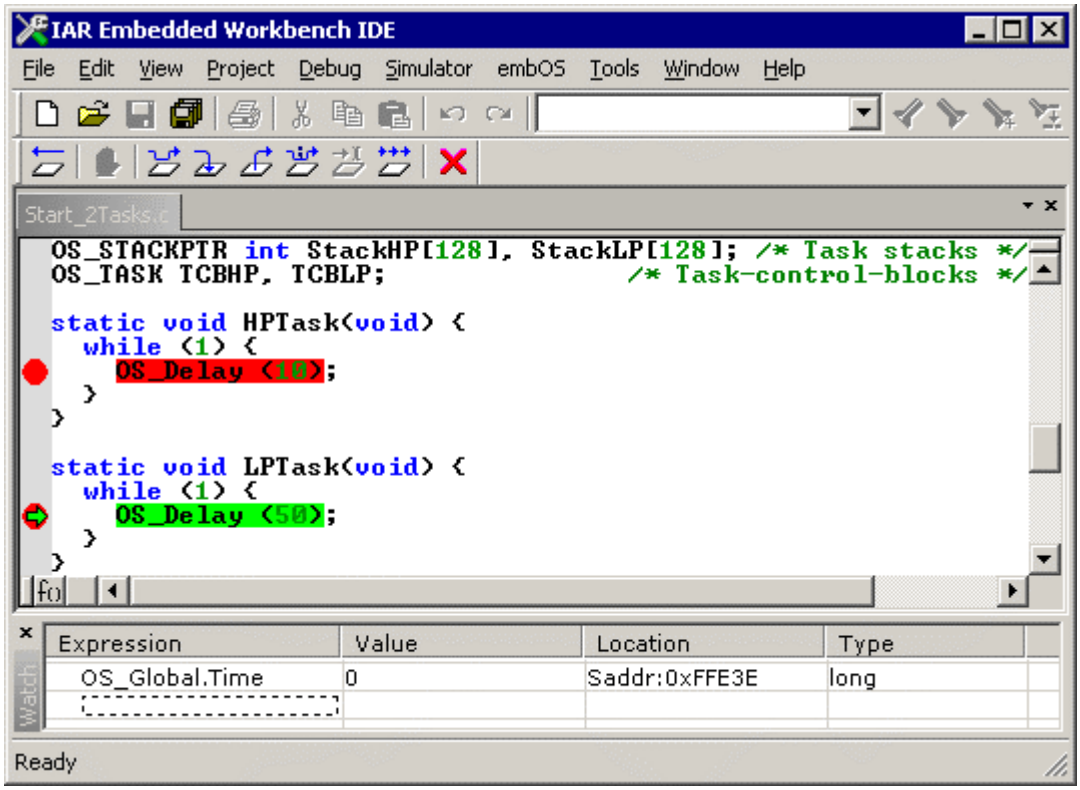
Before you execute `OS_Start()`, you should set break points in HPTask and LPTask:



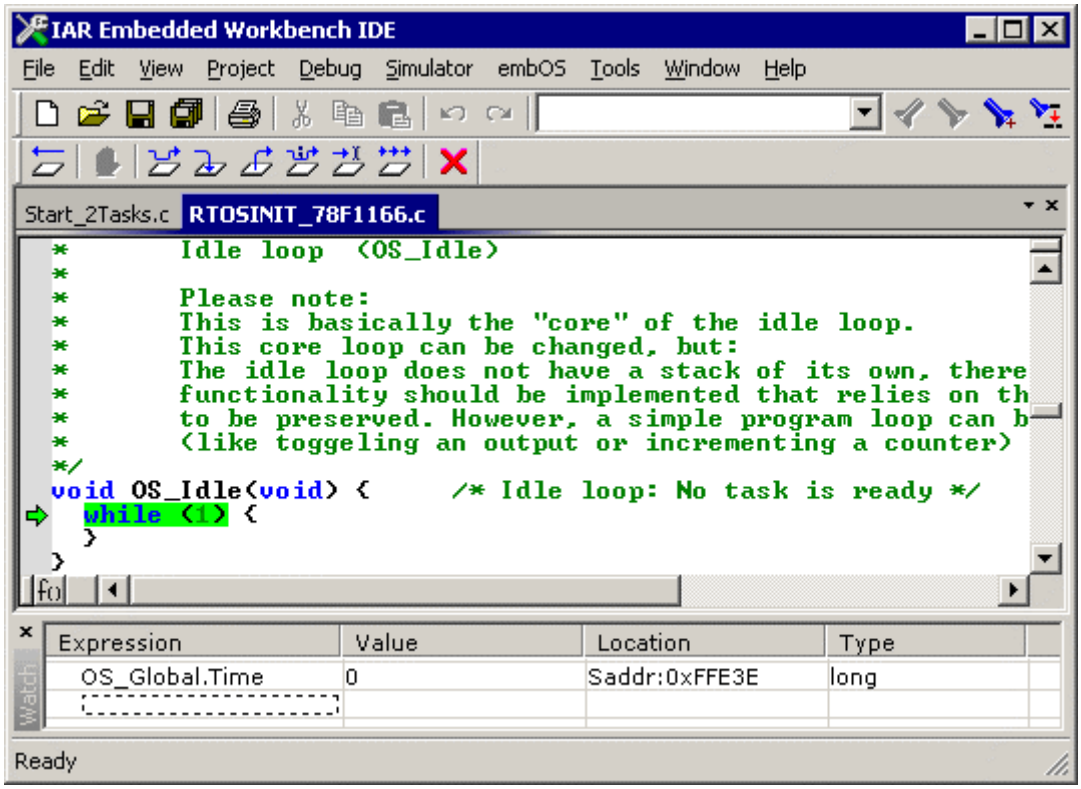
When step into `OS_Start()`, you can only step into it in disassembly mode, because this function is part of the ***embOS*** library. However, you can press GO now or step in disassembly mode until you reach the highest priority task.



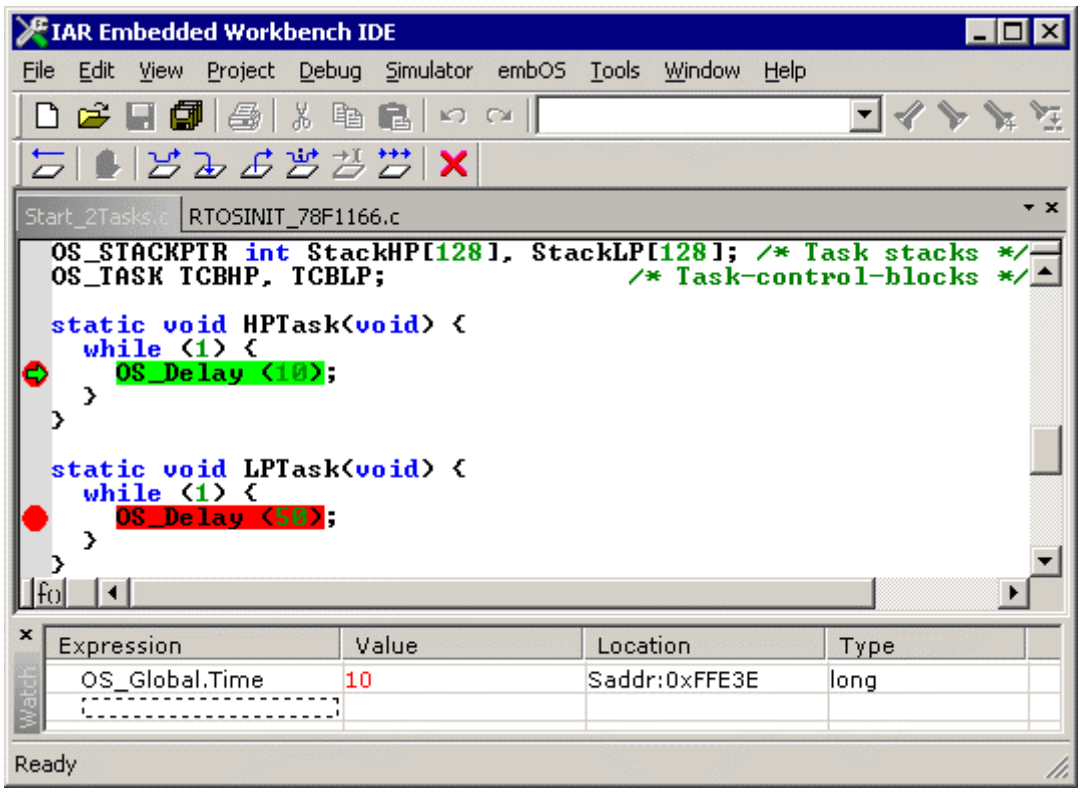
If you continue program execution, you will arrive in the task with the second highest priority:



Continuing the program 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:



As can be seen by the value of ***embOS*** timer variable, `HPTask()` continues operation after the given delay of 10 ***embOS*** timer tick cycles.

3. Renesas K/0, K/0S and K0R specifics

3.1. Processor configuration

embOS supports 78K/0 (7800X, 780XXX) and 78K/0S (789XXX) processors using the standard memory model.
For K0R CPUs, all code model- / data model-combinations are supported.

3.2. Available libraries for K0 CPUs, used with CLIB

Core	CPU variant	Memory model	Library type	Library
7800x	78K0 basic	Standard	Extreme Release	rtosCS0XR.r26
7800X	78K0 basic	Standard	Release	rtosCS0R.r26
7800X	78K0 basic	Standard	Stack-check	rtosCS0S.r26
7800X	78K0 basic	Standard	Stack-check + Profiling	rtosCS0SP.r26
7800X	78K0 basic	Standard	Debug	rtosCS0D.r26
7800X	78K0 basic	Standard	Debug + Profiling	rtosCS0DP.r26
7800X	78K0 basic	Standard	Debug + Trace	rtosCS0DT.r26
7801x	78K0	Standard	Extreme Release	rtosCS1XR.r26
7801X	78K0	Standard	Release	rtosCS1R.r26
7801X	78K0	Standard	Stack-check	rtosCS1S.r26
7801X	78K0	Standard	Stack-check + Profiling	rtosCS1SP.r26
7801X	78K0	Standard	Debug	rtosCS1D.r26
7801X	78K0	Standard	Debug + Profiling	rtosCS1DP.r26
7801X	78K0	Standard	Debug + Trace	rtosCS1DT.r26

3.3. Available libraries for K0 CPUs, used with DLIB

Core	CPU variant	Memory model	Library type	Library
7800x	78K0 basic	Standard	Extreme Release	rtosDS0XR.r26
7800X	78K0 basic	Standard	Release	rtosDS0R.r26
7800X	78K0 basic	Standard	Stack-check	rtosDS0S.r26
7800X	78K0 basic	Standard	Stack-check + Profiling	rtosDS0SP.r26
7800X	78K0 basic	Standard	Debug	rtosDS0D.r26
7800X	78K0 basic	Standard	Debug + Profiling	rtosDS0DP.r26
7800X	78K0 basic	Standard	Debug + Trace	rtosDS0DT.r26
7801x	78K0	Standard	Extreme Release	rtosDS1XR.r26
7801X	78K0	Standard	Release	rtosDS1R.r26
7801X	78K0	Standard	Stack-check	rtosDS1S.r26
7801X	78K0	Standard	Stack-check + Profiling	rtosDS1SP.r26
7801X	78K0	Standard	Debug	rtosDS1D.r26
7801X	78K0	Standard	Debug + Profiling	rtosDS1DP.r26
7801X	78K0	Standard	Debug + Trace	rtosCS1DT.r26

3.4. Available libraries for K0S CPUs, used with CLIB

Core	CPU variant	Memory model	Library type	Library
789XXX	78K0S	Standard	Extreme Release	rtosCS2XR.r26
789XXX	78K0S	Standard	Release	rtosCS2R.r26
789XXX	78K0S	Standard	Stack-check	rtosCS2S.r26
789XXX	78K0S	Standard	Stack-check + Profiling	rtosCS2SP.r26
789XXX	78K0S	Standard	Debug	rtosCS2D.r26
789XXX	78K0S	Standard	Debug + Profiling	rtosCS2DP.r26
789XXX	78K0S	Standard	Debug + Trace	rtosCS2DT.r26

3.5. Available libraries for K0S CPUs, used with DLIB

Core	CPU variant	Memory model	Library type	Library
789XXX	78K0S	Standard	Extreme Release	rtosDS2XR.r26
789XXX	78K0S	Standard	Release	rtosDS2R.r26
789XXX	78K0S	Standard	Stack-check	rtosDS2S.r26
789XXX	78K0S	Standard	Stack-check+ Profiling	rtosDS2SP.r26
789XXX	78K0S	Standard	Debug	rtosDS2D.r26
789XXX	78K0S	Standard	Debug + Profiling	rtosDS2DP.r26
789XXX	78K0S	Standard	Debug + Trace	rtosDS2DT.r26

3.6. Available libraries for K0R CPUs, used with CLIB

Code model	Data model	Library type	Library mode #define	Library
Near	Near	Extreme Release	OS_LIBMODE_XR	rtosCNN3XR.r26
Near	Near	Release	OS_LIBMODE_R	rtosCNN3R.r26
Near	Near	Stack-check	OS_LIBMODE_S	rtosCNN3S.r26
Near	Near	Stack-check + Profiling	OS_LIBMODE_SP	rtosCNN3SP.r26
Near	Near	Debug	OS_LIBMODE_D	rtosCNN3D.r26
Near	Near	Debug + Profiling	OS_LIBMODE_DP	rtosCNN3DP.r26
Near	Near	Debug + Trace	OS_LIBMODE_DT	rtosCNN3DT.r26
Near	Far	Extreme Release	OS_LIBMODE_XR	rtosCNF3XR.r26
Near	Far	Release	OS_LIBMODE_R	rtosCNF3R.r26
Near	Far	Stack-check	OS_LIBMODE_S	rtosCNF3S.r26
Near	Far	Stack-check + Profiling	OS_LIBMODE_SP	rtosCNF3SP.r26
Near	Far	Debug	OS_LIBMODE_D	rtosCNF3D.r26
Near	Far	Debug + Profiling	OS_LIBMODE_DP	rtosCNF3DP.r26
Near	Far	Debug + Trace	OS_LIBMODE_DT	rtosCNF3DT.r26
Far	Near	Extreme Release	OS_LIBMODE_XR	rtosCFN3XR.r26
Far	Near	Release	OS_LIBMODE_R	rtosCFN3R.r26
Far	Near	Stack-check	OS_LIBMODE_S	rtosCFN3S.r26
Far	Near	Stack-check + Profiling	OS_LIBMODE_SP	rtosCFN3SP.r26
Far	Near	Debug	OS_LIBMODE_D	rtosCFN3D.r26
Far	Near	Debug + Profiling	OS_LIBMODE_DP	rtosCFN3DP.r26
Far	Near	Debug + Trace	OS_LIBMODE_DT	rtosCFN3DT.r26
Far	Far	Extreme Release	OS_LIBMODE_XR	rtosCFF3XR.r26
Far	Far	Release	OS_LIBMODE_R	rtosCFF3R.r26
Far	Far	Stack-check	OS_LIBMODE_S	rtosCFF3S.r26
Far	Far	Stack-check + Profiling	OS_LIBMODE_SP	rtosCFF3SP.r26
Far	Far	Debug	OS_LIBMODE_D	rtosCFF3D.r26
Far	Far	Debug + Profiling	OS_LIBMODE_DP	rtosCFF3DP.r26
Far	Far	Debug + Trace	OS_LIBMODE_DT	rtosCFF3DT.r26

3.7. Available libraries for K0R CPUs, used with DLIB

Code model	Data model	Library type	Library mode #define	Library
Near	Near	Extreme Release	OS_LIBMODE_XR	rtosDNN3XR.r26
Near	Near	Release	OS_LIBMODE_R	rtosDNN3R.r26
Near	Near	Stack-check	OS_LIBMODE_S	rtosDNN3S.r26
Near	Near	Stack-check + Profiling	OS_LIBMODE_SP	rtosDNN3SP.r26
Near	Near	Debug	OS_LIBMODE_D	rtosDNN3D.r26
Near	Near	Debug + Profiling	OS_LIBMODE_DP	rtosDNN3DP.r26
Near	Near	Debug + Trace	OS_LIBMODE_DT	rtosDNN3DT.r26
Near	Far	Extreme Release	OS_LIBMODE_XR	rtosDNF3XR.r26
Near	Far	Release	OS_LIBMODE_R	rtosDNF3R.r26
Near	Far	Stack-check	OS_LIBMODE_S	rtosDNF3S.r26
Near	Far	Stack-check + Profiling	OS_LIBMODE_SP	rtosDNF3SP.r26
Near	Far	Debug	OS_LIBMODE_D	rtosDNF3D.r26
Near	Far	Debug + Profiling	OS_LIBMODE_DP	rtosDNF3DP.r26
Near	Far	Debug + Trace	OS_LIBMODE_DT	rtosDNF3DT.r26
Far	Near	Extreme Release	OS_LIBMODE_XR	rtosDFN3XR.r26
Far	Near	Release	OS_LIBMODE_R	rtosDFN3R.r26
Far	Near	Stack-check	OS_LIBMODE_S	rtosDFN3S.r26
Far	Near	Stack-check + Profiling	OS_LIBMODE_SP	rtosDFN3SP.r26
Far	Near	Debug	OS_LIBMODE_D	rtosDFN3D.r26
Far	Near	Debug + Profiling	OS_LIBMODE_DP	rtosDFN3DP.r26
Far	Near	Debug + Trace	OS_LIBMODE_DT	rtosDFN3DT.r26
Far	Far	Extreme Release	OS_LIBMODE_XR	rtosDFF3XR.r26
Far	Far	Release	OS_LIBMODE_R	rtosDFF3R.r26
Far	Far	Stack-check	OS_LIBMODE_S	rtosDFF3S.r26
Far	Far	Stack-check + Profiling	OS_LIBMODE_SP	rtosDFF3SP.r26
Far	Far	Debug	OS_LIBMODE_D	rtosDFF3D.r26
Far	Far	Debug + Profiling	OS_LIBMODE_DP	rtosDFF3DP.r26
Far	Far	Debug + Trace	OS_LIBMODE_DT	rtosDFF3DT.r26

3.8. Profiling

Cycle accurate time measurement needs a 32bit X 32bit operation. Because 78K/0 and 78K/0S cannot execute 32 bit multiplications in a fast way, we do not recommend to use the profiling libraries (SP, DP or DT). If you are going to use profiling libraries, the interrupt latency time may increase drastically.

3.9. Changing the tick frequency

Normally, the initialization code in RTOSInit.c is set up to generate an ***embOS*** timer interrupt every millisecond.
For very slow CPUs such as some of the K0/K0S CPUs, it might be better to generate interrupts at larger periods.
Different (lower or higher) interrupt rates are possible. If you choose an interrupt frequency different from 1 kHz, the value of the time variable will no longer be equivalent to multiples of 1 ms. However, if you use a multiple of 1ms as tick time, the basic time unit can be made 1 ms by using the (optional) configuration function `OS_TICK_Config()` and calling the special ***embOS*** tick handler function `OS_TICK_HandleEx()` instead one used in the ***embOS*** timer interrupt handler. The basic time unit does not have to be 1 ms; it might just as well be 10 ms or any other value.

3.9.1.OS_TICK_Config()

OS_TICK_Config() can be used to configure ***embOS*** in situations where the basic timer-interrupt interval (tick) is a multiple of 1 ms and the time values for delays still should represent 1 ms as the time base. `OS_CONFIG()` tells ***embOS*** how many system time units expire per ***embOS*** tick and what the system frequency is.

Example

The following code example will instruct ***embOS*** to increment the time variable OS_Time by 2 per ***embOS*** timer-interrupt.

```
OS_TICK_Config(2,1); /* Configure OS : System-frequency, ticks/int */
```

If, for example, the basic timer was initialized to 500 Hz, which would result in an ***embOS*** timer-interrupt every 2 ms, a call of `OS_Delay(10)` would result in a delay of 20 ms, because all timing values are interpreted as ticks. A call of `OS_TICK_Config()` with the parameters shown in example 2 would compensate for the difference, resulting in a delay of 10 ms when calling `OS_Delay(10)`.

Note:

The default *embOS* timer tick handler does not handle the settings which were made by a call of `OS_TICK_Config()`.
The alternate ***embOS*** timer tick handler `OS_TICK_HandleEx()` has to be called from the timer interrupt service routine to use the correction which was set by `OS_TICK_Config()`.
To use `OS_TICK_Config()`, the timer interrupt service routine in `RTOSInit.c` has to be modified as shown below:

```
#pragma vector=OS_INTTM00_vect
__interrupt void OS_ISR_Tick (void) {
  OS_EnterNestableInterrupt();
  OS_EnterIntStack();
  OS_TICK_HandleEx();
  OS_LeaveIntStack();
  OS_LeaveNestableInterrupt();
}
```

4. Stacks

4.1. Task stack for Renesas K/0, K/0S and K0R

The stack pointer can point to any location of the K/0 & K/0S address space. The stack pointer of K0R is a 16bit register which can address the upper 64KB of the address space. However, usually only the internal high speed RAM is suitable for stacks; please check also the hardware manual of your device. If you use one of our sample *.xcl files and the storage modifier for your stack declaration, the stack will be placed automatically into the internal high speed RAM.
We defined a section `K0_STACKS` in our linker control files for the K0 CPUs.
To ensure that stacks are located in this section, you have to use a #pragma statement or the “@” operator to place the task stacks into internal high speed RAM. Please refer to our XCL-files and sample main application:

```
/* Task stacks need to be located in segment K0_STACKS   */
/* which resides in internal high speed RAM.             */
__no_init int Stack0[64] @ "K0_STACKS";
__no_init int Stack1[64] @ "K0_STACKS";
```

4.2. System and Interrupt stack for Renesas K/0, K/0S and K0R

The IAR CSTACK is used as system stack. Your application uses this stack before executing `OS_Start()`, during internal functions and during the timer tick routines. Also software timers use the system stack. If your interrupt service routines use `OS_EnterIntStack()`, they will also use the system stack.
The CSTACK segment also has to be located in internal high speed RAM on most CPUs.

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 corresponding interrupt service routine (ISR) is started
- The first thing you should do in the ISR, is to call `OS_EnterInterrupt()` `or OS_EnterNestableInterrupt()`. This tells ***embOS***, that you are executing an ISR. In case of calling `OS_EnterNestableInterrupt()` ***embOS*** will re-enable interrupts again to allow nesting.
- The ISR does store all registers which are modified by the ISR on the current stack. Current stack is either a task stack or the system stack,
- If your are using `OS_EnterIntStack()` in the ISR, it will switch the stack pointer to the system stack. Please be aware, that a function calling `OS_EnterIntStack()` is not allowed to have local variables.
- If you used `OS_EnterIntStack()` at the beginning of your ISR, you have to call `OS_LeaveIntStack()` at the end of this function. The stack pointer will be restored to its original value.
- Depending on which function you have called at the beginning of your ISR, you will have to call `OS_LeaveInterrupt()` or `OS_LeaveNestableInterrupt()` and the ISR will return from interrupt. If the ISR caused a task switch, it will take place immediately when leaving the ISR.

5.2. Defining interrupt handlers in "C"

The definition of an interrupt function using ***embOS*** calls is very much the same as for a normal interrupt service routine (ISR). If your ISR will use ***embOS*** system calls, or if you enable interrupts again in your ISR, you will have to call `OS_EnterInterrupt() or OS_EnterNestableInterrupt()` at the start and `OS_LeaveInterrupt()` or `OS_LeaveNestableInterrupt()` at the end of your ISR. In case you want to execute the ISR on the system stack, you will have to call `OS_EnterIntStack()` right after e.g. `OS_EnterInterrupt()` and `OS_LeaveIntStack()` right before e.g. `OS_LeaveInterrupt()`.

Example

"Simple" interrupt-routine

```
#pragma vector= INTTM00_vect
__interrupt void OS_ISR_Tick (void) {
  OS_EnterNestableInterrupt();
  OS_EnterIntStack();
  OS_TICK_Handle();
  OS_LeaveIntStack();
  OS_LeaveNestableInterrupt();
}
```

5.3. Interrupt-stack

The routines `OS_EnterIntStack()` and `OS_LeaveIntStack()` can be used to switch the stack pointer to the system stack during execution of the ISR. If you are not using these routines, the ISR uses the active staks. The active stack is either a task stack or the system stack.

5.4. Interrupt stack switching

Since the Renesas K0/K0S/K0R CPUs do not have a separate stack pointer for interrupts, every interrupt runs on the current stack. To reduce stack load of tasks, ***embOS*** offers its own interrupt stack which is located in the system stack.
To use the ***embOS*** interrupt stack, call `OS_EnterIntStack()` at the beginning of an interrupt handler just after the call of `OS_EnterInterrupt()` and call `OS_LeaveIntStack()` at the end just before `OS_LeaveInterrupt()`.

Please note, that an interrupt handler using interrupt stack switching must not use local variables.

Interrupt-routine using ***embOS*** interrupt stack:

```
static void OS_ISR_Rx_Handler(void) {
  int Dummy;
  if (ASIS0 & 0x07) {           /* Check any reception error        */
    Dummy = RXB0;               /* Reset error, discard Byte        */
  } else {
    OS_OnRx(RXB0);              /* Process data                     */
  }
}

__interrupt [INTSR0_vect] void OS_ISR_rx(void) {
  OS_EnterNestableInterrupt();  /* We will enable interrupts        */
  OS_EnterIntStack();           /* We will use interrupt stack      */
  OS_ISR_Rx_Handler();          /* A call to a handler is required !*/
  OS_LeaveIntStack();           /* Interrupt stack switching does   */
  OS_LeaveNestableInterrupt();  /* not allow local variables in ISR */
}
```

6. STOP / WAIT Mode

In case your controller supports 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 the function `OS_Idle()`, which is implemented in the source file 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***. The minimum ROM requirement for the kernel itself is about 1.500 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	18
Resource semaphore	4
Counting semaphore	2
Mailbox	12
Software timer	10

8. Files shipped with *embOS*

Directory	File	Explanation
root	`*.pdf`	Generic API- and target specific documentation
root	`Release.html`	Release notes of ***embOS*** for K0/K0S/K0R
root	`embOSView.exe`	Utility for runtime analysis, described in generic documentation
Start\Inc\	`Rtos.h`	Include file for ***embOS,*** to be included in every “C”-file using ***embOS*** functions.
Start\Lib\	`rtos*.r26`	Libraries for all memory models
Start\BoardSupport\CPU_*\	`Start*.eww`	CPU specific sample workspace
Start\BoardSupport\CPU_*\	`Start*.ewp`	CPU specific sample project
Start\BoardSupport\CPU_*\	`Start*.ewd`	CPU and project specific debug configuration file.
Start\BoardSupport\CPU_*\Application\	`*.*`	Sample application programs.
Start\BoardSupport\CPU_*\Setup\	`RTOSInit*.*`	Target CPU specific init functions. May be modified if required.
Start\BoardSupport\CPU_*\Setup\	`OS_Error.c`	***embOS*** error handler, used in stack check or debug builds.
Start\BoardSupport\CPU_*\Setup\	`*.mac`	Target CPU specific simulation macro files for C-SPY simulator.
Start\BoardSupport\CPU_*\Setup\	`*.*`	Target CPU specific linker files and others required for the specific CPU variant

embOSView and the manuals are found in the root directory of the distribution. Any additional files are shipped as example.

9. Index