embOS

Real Time Operating System

CPU & Compiler specifics for

RENESAS H8/300H / H8S CPUs

and IAR compiler

Document Rev. 4

SEGGER

A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Contents

1. About this document

This guide describes how to use ***embOS*** for H8/H8S Real Time Operating System for the RENESAS H8/300H and H8S series of microcontroller using IAR compiler and IARs Embedded Workbench.

1.1. How to use this manual

This manual describes all CPU and compiler specifics for ***embOS*** using H8 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 C compiler and IAR's Embedded Workbench. If you have no experience using ***embOS***, you should follow this introduction, even if you do not plan to use 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 H8 CPUs and IAR compiler.

Naming convention

As ***embOS*** for H8 supports two different CPU cores, some aspects described in the following chapters may be specific for H8/300H or H8S CPU cores and others may be valid for both. We use the following CPU naming convention:

- **H8** is used when description covers both CPU cores.
- **H8/300H** is used for information specific to H8/300H CPU core.
- **H8S** is used for information specific to H8S CPUs.

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

The following chapter describes how to install and work with ***embOS*** for H8 CPUs and IAR's 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's Embedded Workbench to develop your application, no further installation steps are required. You will find prepared sample start projects for H8/300H and H8S 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.***

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.

2.2. First steps

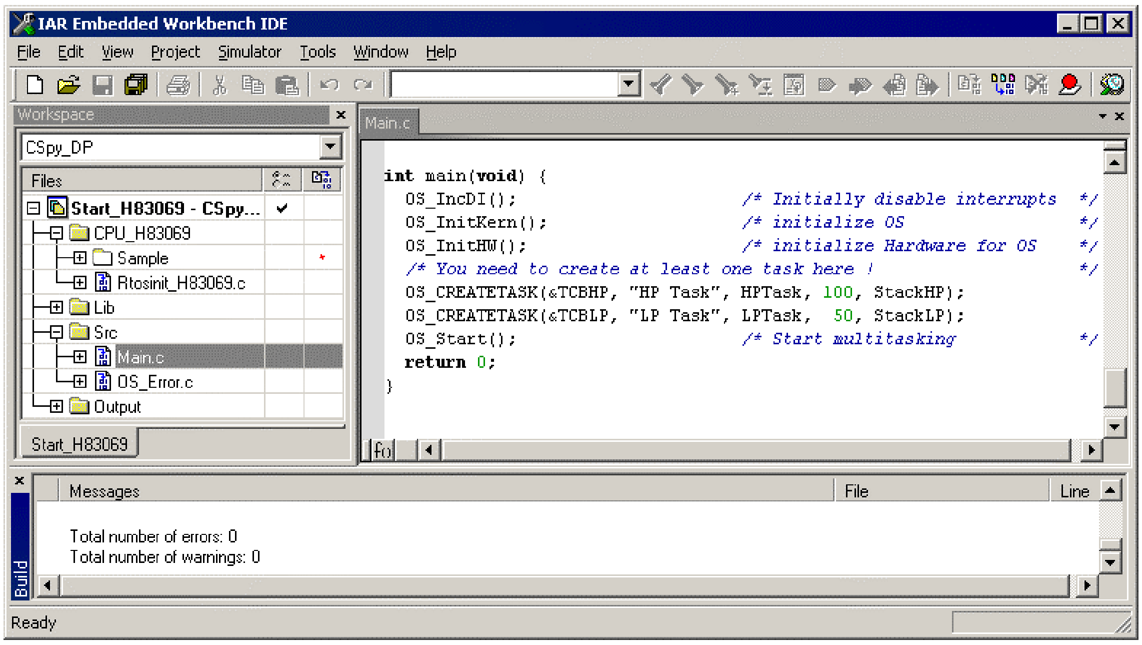
After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received a ready to go sample start workspace and projects for H8/300H and H8S CPUs and it is a good idea to use this as a starting point of all your applications.

Your ***embOS*** distribution contains one folder ‘Start’ which contains the sample start workspace and projects and every additional files used to build your application.

To get your application running, you should proceed as follows:

- Create a work directory for your application, for example c:\work
- Copy all files and subdirectories from the ***embOS*** distribution disk 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’ in your work directory.
- Open the start workspace ‘Start.eww’. (e.g. by double clicking it)
- Select and build one the start project, preferably a configuration for CSpy

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



```
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;
}
```

```
Total number of errors: 0
Total number of warnings: 0
```

Initially the configuration for huge memory model for IAR’s simulator / debugger CSpy is selected.
If you do not have CSpy installed, you may select an other target which is use-able for your simulator / debugger.

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.

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.

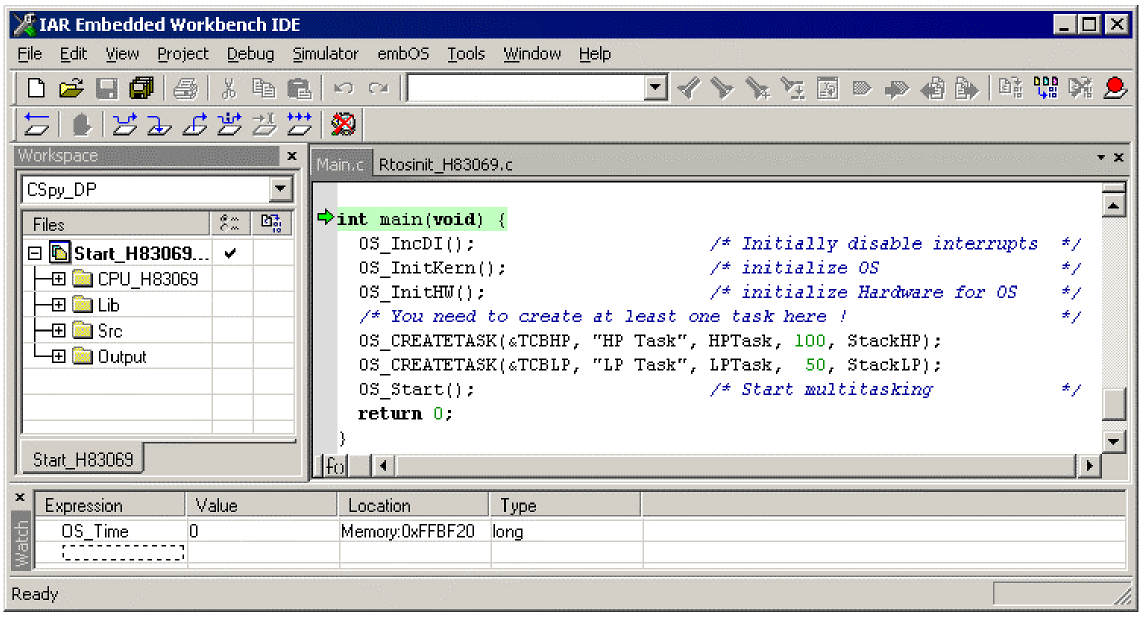
```
/*********************************************************************
*          SEGGER MICROCONTROLLER SYSTEME GmbH
*   Solutions for real time microcontroller applications
**********************************************************************
----------------------------------------------------------------------
File    : Main.c
Purpose : Skeleton program for embOS
--------  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;
}
```

3. Using debugging tools to debug the application

The ***embOS*** start projects contain configurations which are already setup for IAR's debugger / simulator C-Spy. These targets are named "DP_CSpy".
The CSpy configurations are prepared to produce the appropriate output files required by IAR's CSpy debugger.
The following chapter describes a sample session based on our sample application main and target DP_CSpy.

3.1. Using IAR's CSpy simulator

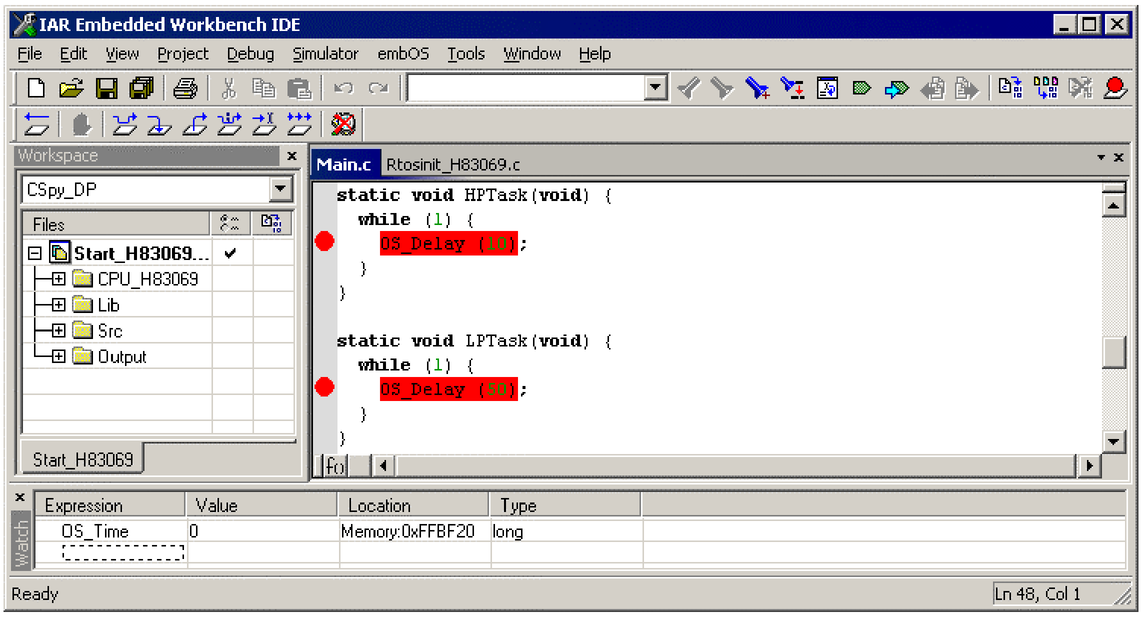
When starting CSpy simulator after building the CSpy configuration, you will usually see the main function, or you may look at the startup code and have to set a breakpoint at main. Now you can step through the program.
`OS_IncDI()` disables interrupts and tells ***embOS***, that interrupts should not be enabled during `OS_InitKern()`.
`OS_InitKern()`initializes ***embOS*** –Variables. If `OS_incDI()` was not called before, interrupts will be enabled. 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()` in `OS_InitHW()` is optional. It is required if embOSView shall be used. As simulators usually can not simulate UART operations, OS_UART should be defined as (-1) to disable UART initialization and communication.
`OS_Start()` is the last line executed in main, since it starts multitasking and does not return.



```
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;
}
```

Expression	Value	Location	Type
OS_Time	0	Memory:0xFFBF20	long

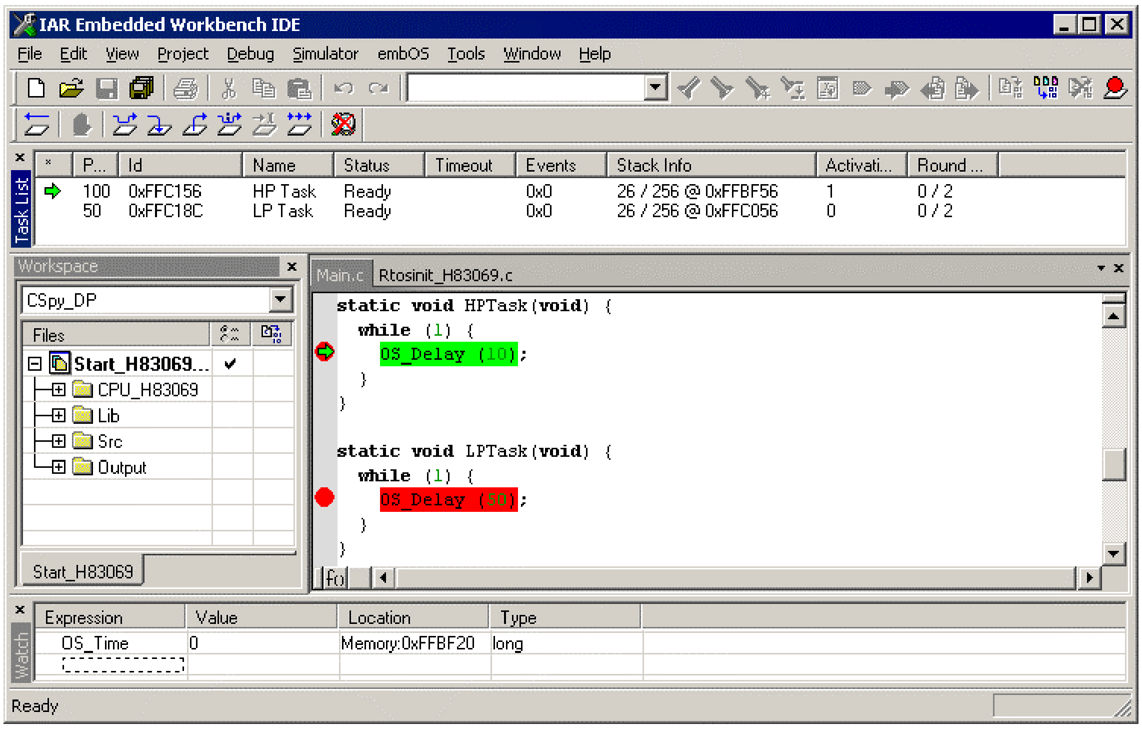
Before you step into OS_Start(), you should set two break points in the two tasks as shown below



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

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

Expression	Value	Location	Type
OS_Time	0	Memory:0xFFBF20	long

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.

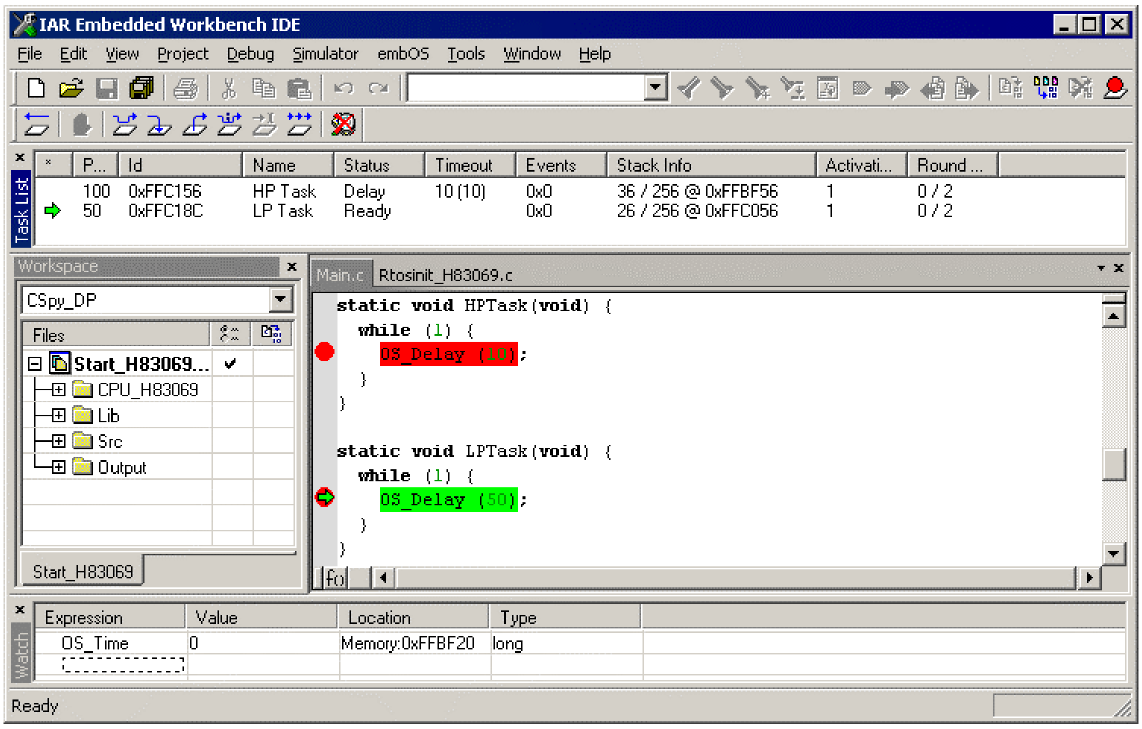


*	P...	Id	Name	Status	Timeout	Events	Stack Info	Activati...	Round ...
➔	100	0xFFC156	HP Task	Ready		0x0	26 / 256 @ 0xFFBF56	1	0 / 2
	50	0xFFC18C	LP Task	Ready		0x0	26 / 256 @ 0xFFC056	0	0 / 2

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

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

Expression	Value	Location	Type
OS_Time	0	Memory:0xFFBF20	long

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

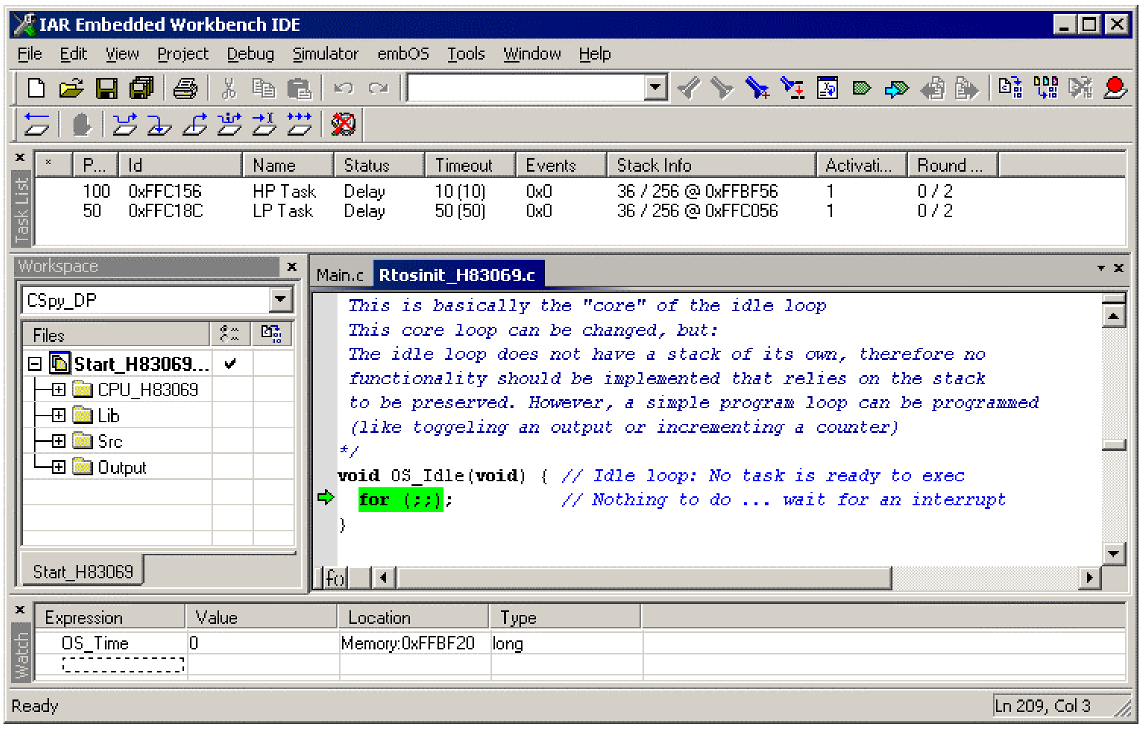


*	P...	Id	Name	Status	Timeout	Events	Stack Info	Activati...	Round ...
	100	0xFFC156	HP Task	Delay	10 (10)	0x0	36 / 256 @ 0xFFBF56	1	0 / 2
	50	0xFFC18C	LP Task	Ready		0x0	26 / 256 @ 0xFFC056	1	0 / 2

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

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

Expression	Value	Location	Type
OS_Time	0	Memory:0xFFBF20	long

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 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:

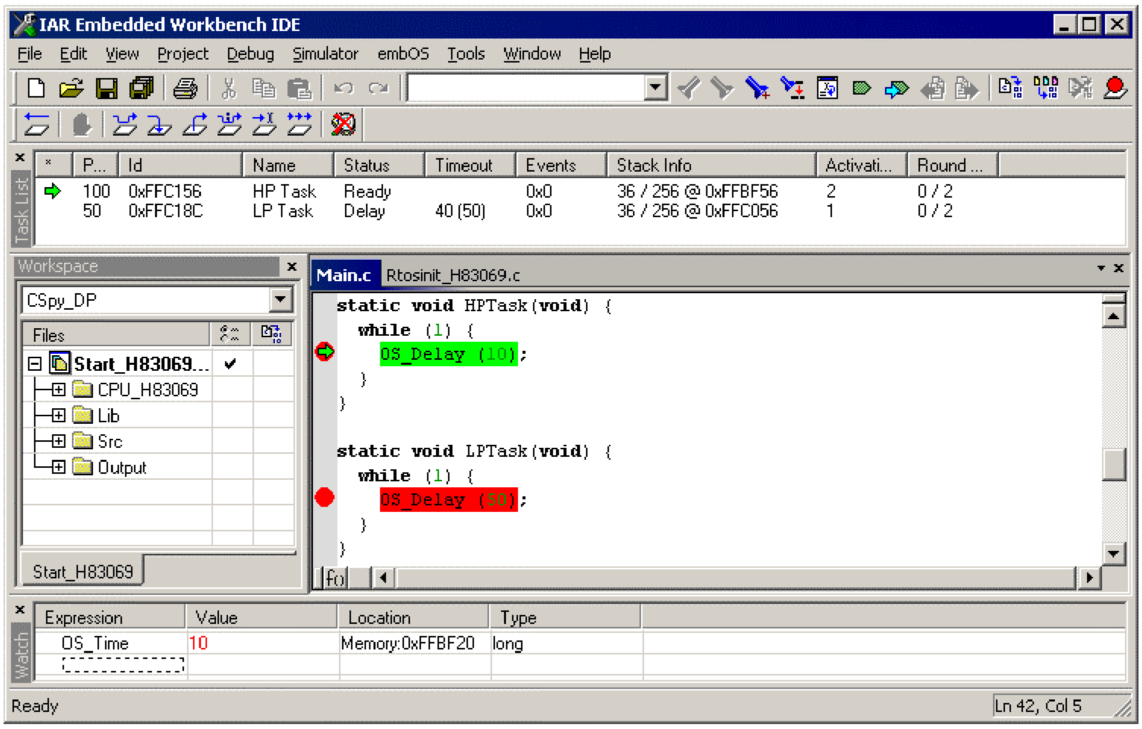


*	P...	Id	Name	Status	Timeout	Events	Stack Info	Activati...	Round ...
	100	0xFFC156	HP Task	Delay	10 (10)	0x0	36 / 256 @ 0xFFBF56	1	0 / 2
	50	0xFFC18C	LP Task	Delay	50 (50)	0x0	36 / 256 @ 0xFFC056	1	0 / 2

```
  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
  for (;;);          // Nothing to do ... wait for an interrupt
}
```

Expression	Value	Location	Type
OS_Time	0	Memory:0xFFBF20	long

If you set a breakpoint in 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:



*	P...	Id	Name	Status	Timeout	Events	Stack Info	Activati...	Round ...
➡	100	0xFFC156	HP Task	Ready		0x0	36 / 256 @ 0xFFBF56	2	0 / 2
	50	0xFFC18C	LP Task	Delay	40 (50)	0x0	36 / 256 @ 0xFFC056	1	0 / 2

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

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

Expression	Value	Location	Type
OS_Time	10	Memory:0xFFBF20	long

As can be seen by the value of ***embOS*** timer variable `OS_Time`, shown in the watch window, Task0 continues operation after expiration of the 10 ms delay.

3.2. 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()`.
Using an emulator or simulator 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 or simulator, 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.

4. Build your own application

To build your own application, you should start with one of the sample start projects. This has the advantage, that all necessary files are included and all settings for the project are already done.

4.1. 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 one CPU subfolder.
  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\ 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 are called.

4.2. Select a start project

embOS comes with start projects for H8/300H and H8S CPUs, which include different configurations for different output formats or debug tools. The start project for H8/300H CPU was built and tested for H83069 CPU. For various CPU variants there may be modifications required as described later in this manual.

4.3. 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\.

4.4. Change memory model or library mode

For your application you may have to choose a different 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:
- Add the appropriate library from the Lib-subdirectory to your Lib group.
- Exclude the previous library from build.
- Set the appropriate OS_LIBMODE_* define as project option.

Finally check project options about target CPU, memory model settings and compiler settings according library mode used. Refer to chapter 5 about the library naming conventions to select the correct library.

5. IAR compiler specifics

5.1. Memory models, compiler options

embOS for H8 for IAR compiler is delivered with libraries for the all combinations of CPU operating modes and data models supported by IAR compiler.

5.2. Available libraries

embOS is shipped with libraries for two different CPU variants and all memory models.
The library name is composed as follows:

Rtos <CPU> <operating mode> <data model> <float>_<LibMode>. r37

Parameter	Meaning	Values
CPU	CPU variant	H8H: H8/300H CPU type
		H8S: H8S CPU type
operating mode	Operating mode of CPU	l: advanced mode
		s: normal mode
data model	Data model for variable addressing	s: small data model
		h: huge data model
float	Size of “double” variabales	f: 32bit
		d: 64bit
LibMode	Library mode	R: Release
		S: Stack check
		SP: Stack check + profiling
		D: Debug
		DP: Debug + profiling
		DT: Debug + profiling + Trace

For each operating mode / data model / float size combinaton, all ***embOS*** library modes are available:

Library mode	Meaning	define
R	Release	OS_LIBMODE_R
S	Stack check	OS_LIBMODE_S
SP	Stack check + Profiling	OS_LIBMODE_SP
D	Debug + stack check	OS_LIBMODE:_D
DP	Debug + stack check + Profiling	OS_LIBMODE_DP
DT	Debug + stack check + profiling + Trace	OS_LIBMODE_DT

This results in 72 different libraries delivered with ***embOS***.

When using IAR workbench, please check the following points:

- Operating mode and data model are set as general project option
- One ***embOS*** library is part of your project (included in one group of your target). The CPU type and memory model of the library used has to fit to the project options.
- The appropriate define according to ***embOS*** library mode is set as compiler preprocessor option for your project.

5.3. Distributed project files

The distribution of ***embOS*** for H8 and IAR compiler comes with start projects for H8/300H CPUs and H8S CPUs:

- **Start_H83069** is a sample start project, prepared for starterkit EDK3069
- **Start_H8S2239** is a start project file for H8S CPUs

The start projects contain configurations for different ***embOS*** library modes and debug options.

5.4. Distributed configurations

Different configurations are included in the sample start projects. The configurations are named according the output format they were built for:

- **Target_***: Produces an Motorola output file
- **CSpy_***: Is set up for CSPY Simulator, produces the correct output file and starts ***embOS*** timer interrupt simulation.

6. H8/300H and H8S CPU specifics

All hardware specific functions required for ***embOS*** are located in the CPU specific RTOSInit_*.c files.
Settings for CPU clock speed and UART settings for embOSView are defined with most common defaults. According to your specific hardware, these settings may have to be changed to ensure proper timer tick and UART communication with embOSView..
As far as possible, you should not modify RTOSInit.c, as this has the disadvantage, that this modifications have to be tracked when you update to a newer version of ***embOS***.
Various CPU derivates may be equipped with different peripherals. It may be necessary to write your own initialization code for your specific CPU derivate.
You may therefore copy one RTOSInit_*.c file which is closest to your CPU variant and modify this new created file to handle your CPU.

6.1. Clock settings and corrections for ***embOS*** timer interrupt

`OS_InitHW()` routine in `RTOSInit.c` derives timer init values from the constant define `OS_PCLK_TIMER`. Per default, the value of `OS_PCLK_TIMER` equals `OS_FSYS`, which defines the CPU clock of the target system. Wrong settings would result ***embOS*** timer ticks unequal to 1 ms.
To adapt the ***embOS*** timer tick frequency to your CPU, you may:

- Define `OS_FSYS` as project option. `OS_FSYS` should equal your CPU clock frequency in Hertz. Note that modification of `OS_FSYS` may also affect the UART initialization for embOSView.
- You may alternatively define `OS_PCLK_TIMER` as project option (compiler preprocessor option). This value is used to calculate the timer compare value used for ***embOS*** timer.

6.2. Clock settings and corrections for UART used for embOS-View

`OS_COM_Init()` routine in `RTOSInit.c` derives baudrate generator init values from the constant define `OS_PCLK_UART`. Per default, the value of `OS_PCLK_UART` equals `OS_FSYS`, which defines the CPU clock of the target system.

To correct the ***embOS*** UART baudrate for embOSView, you may:

- Define `OS_FSYS` as project option. `OS_FSYS` should equal your CPU clock frequency in Hertz. Note that modification of `OS_FSYS` may also affect the timer initialization for ***embOS***.
- You may alternatively define `OS_PCLK_UART` as project option (compiler preprocessor option). This value is used to calculate values used to initialize UART used for communication with embOSView.

6.3. Conclusion about clock settings

- **OS_FSYS** has to be defined according to your CPU clock frequency. This should be defined as compiler preprocessor option in your project.
- **OS_PCLK_TIMER** has to be defined to fit the frequency used as peripheral clock for the ***embOS*** timer. The value defaults to OS_FSYS. It should be

modified and defined as compiler preprocessor option if modification is required.

- **OS_PCLK_UART** has to be defined to fit the frequency used as peripheral clock for the UART used for communication with embOSView. The value defaults to OS_FSYS. It should be modified and defined as compiler preprocessor option if modification is required.

7. Stacks

7.1. Task stack for H8 CPUs

Every ***embOS*** task has to have its own stack. Task stacks can be located in any RAM memory location that can be used as stack by the H8 CPU.
As H8 CPUs have a 32 bit stack pointer, the whole memory area can be used as task stack.
Using Small data model, the available memory is limited to 64 KBytes which are accessible in near memory range.
Please note, that the task stacks have to be aligned at EVEN addresses. To ensure proper alignment, implement task stack as array of int.
The stack-size required for tasks 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 H8, this minimum task stack size is about 42 bytes in the near memory model.
As current version of ***embOS*** does not support a separate interrupt stack, all interrupts may run on the task stacks as well. Therefore we recommend at least a minimum of 128 bytes for task stacks.

7.2. System stack for H8 CPUs

The system stack size required by ***embOS*** is about 40 bytes (65 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.
The stack used as system stack is the one defined as CSTACK in the linker command file (*.xcl) and normally its size is defined as a project option.
A good value for the system stack is typically about 128 to 256 bytes.

7.3. Interrupt stack for H8 CPUs

The H8 CPUs do not support a hardware interrupt stack. All interrupts run on the current stack.
Therefore the size of task stacks and the system stack have to be large enough to handle all nested interrupts and subroutine calls.

7.4. Reducing the stack size

The stack check libraries check the used stack of every task and the system stack also. Using embOSView the total size and used size of any stack can be examined. This may be used to reduce the stack sizes, if RAM space is a problem in your application.

8. Interrupts with H8/300H CPUs

The following chapter describes interrupt specifics of H8/300H CPUs and interrupt modes used with ***embOS***.

8.1. Interrupt handling process with H8/300H CPUs

H8/300H CPUs support two different interrupt modes. For ***embOS***, interrupt mode 1 is used. This mode supports the following features:

- Interrupt priority registers to assign two priority levels to peripheral interrupts.
- Three level masking by I and UI bit in the CPU condition code register.

Interrupt mode 1 has to be initialized by clearing the `UE` bit in the CPUs System Control Register `SYSCR`. This is normally done in `OS_InitHW()` in RTOSInit.c

Interrupt processing in interrupt mode 1 is as follows:

- The CPU-core receives an interrupt request
- If interrupts are enabled for the priority of the interrupting device, the interrupt is executed.
- All interrupts are masked by setting the I and UI bit.
- 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 RTE command, restoring PC and condition code register.
- For details, please refer to the RENESAS users manual.

8.2. Fast interrupts with H8/300H CPUs

Instead of disabling interrupts when ***embOS*** does atomic operations, the I-Flag (interrupt disable flag) of condition code register is set. This results in disabling interrupts with low priority. All interrupts with high priority can still be processed. These interrupts are named *Fast interrupts*.
You must not execute any *embOS* function from within a *fast interrupt* function.

8.3. Interrupt priorities with H8/300H CPUs

Interrupt priorities useable for interrupts using ***embOS*** API functions are limited.

- Any interrupt handler using ***embOS*** API functions has to run with low interrupt priority. These ***embOS*** interrupt handlers have to start with `OS_EnterInterrupt()` or `OS_EnterNestableInterrupt()` and must end with `OS_LeaveInterrupt()` or `OS_LeaveNestableInterrupt()`.
- Any *Fast interrupt* (running at high priority) must not call any ***embOS*** API function. Even `OS_EnterInterrupt()` and `OS_LeaveInterrupt()` must not be called.
- Interrupt handler running at low priority not calling any ***embOS*** API function are allowed, but must not re-enable interrupts!

8.4. Nested interrupts with H8/300H CPUs

After entering the interrupt service routine, interrupts are automatically disabled by the CPU. As long as interrupts are not re-enabled by software, the interrupt service routine can not be interrupted by any other interrupt regardless of priority.
Interrupt service-routines using ***embOS*** functions may be made nestable by use of the following two functions:

- **OS_EnterInterrupt() :** tells ***embOS***, that interrupt code is running and enables high priority interrupts by resetting the UI-Flag.
- **OS_EnterNestableInterrupt() :** tells ***embOS***, that interrupt code is running and enables all interrupts by resetting the I- and UI-Flag. Thus interrupt service routines running at low priority may also be interrupted by other interrupts running at low priority. The interrupt service routine may nest itself!

Please be careful using nestable interrupt service routines.

- Always reset the interrupt pending condition before re-enabling interrupts.
- To re-enable interrupts in an interrupt service routine using ***embOS*** functions, always use OS_EnterInterrupt() or OS_EnterNestableInterrupt() at the beginning of the ISR.
- Never call OS_EI() or any other interrupt enabling function from inside an interrupt service routine which uses ***embOS*** functions.
- From an interrupt service routine running at high priority, never re-enable low priority interrupts.

8.5. Defining interrupt handlers for H8/300H CPUs in "C"

Routines preceded by the keyword `interrupt` save & restore the registers they modify and return with RTE.
The corresponding interrupt vector number has to be written after the keyword `interrupt` using brackets.
For a detailed description on how to define an interrupt routine in "C", refer to the IAR C-Compiler's user's guide.

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

embOS interrupt handler can be used for interrupt sources running at low priority.

```
void interrupt [TPU_INTVEC] TimerInt (void) {
  _RESET_INT_PENDING(TPU_TISR, TPU_INT_PENDING_BIT);
  OS_EnterNestableInterrupt();
  _HandleTimer();
  OS_LeaveNestableInterrupt();
}
```

Please ensure that interrupt pending condition is reset before re-enabling interrupts by `OS_EnterNestableInterrupt()`.
If re-enabling of interrupts should not be performed in an Interrupt handler, use `OS_EnterInterrupt()` and `OS_LeaveInterrupt()`.
Every interrupt handler using ***embOS*** functions has to start with `OS_EnterInterrupt()` or `OS_EnterNestableInterrupt()` and has to end with `OS_LeaveInterrupt()` or `OS_LeaveNestableInterrupt()`. Inside the interrupt handler, any other function may be called.

9. Interrupts with H8S CPUs

The following chapter describes interrupt specifics of H8S CPUs and the interrupt modes used with ***embOS***.

9.1. Interrupt handling process with H8S CPUs

H8S CPUs support two different interrupt modes. For ***embOS***, interrupt mode 2 is used. This mode supports the following features:

- Interrupt priority registers to assign 8 priority levels to peripheral interrupts.
- Priority level controlled masking.
- Interrupts with higher priority are never disabled by entering an interrupt service routine with lower priority

Interrupt mode 2 has to be initialized during CPU setup. This is normally done in `OS_InitHW()` in RTOSInit.c

Interrupt processing in interrupt mode 2 is as follows:

- The CPU-core receives an interrupt request
- If interrupts are enabled for the priority of the interrupting device, the interrupt is executed.
- The CPU store PC, CCR and EXR onto the current stack.
- The interrupt mask level of the CPU is updated from the level of the interrupting device.
- 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 RTE command, restoring PC, condition code register and EXR.
- For details, please refer to the RENESAS users manual.

9.2. Fast interrupts with H8S CPUs

Instead of disabling interrupts when ***embOS*** does atomic operations, the interrupt level of the CPU is set to 5. Therefore all interrupts with level 6 or above can still be processed.
These interrupts are named ***Fast interrupts***.
You must not execute any *embOS* function from within a *fast interrupt* function.

9.3. Interrupt priorities with ***embOS*** for H8S CPUs

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

- Low priority interrupts with priorities from 1 to 5. These interrupts are called ***embOS*** interrupts.
- High priority interrupts with priorities of 6 and 7. These interrupts are called ***Fast interrupts***.

Interrupt handler functions for both types have to follow the coding guidelines described in the following chapters.

9.4. Defining interrupt handlers for H8S CPUs in "C"

Routines preceded by the keyword `__interrupt` save & restore the registers they modify and return with RTE.
The corresponding interrupt vector number has to be assigned immediately before the interrupt function using a `#pragma` definition:
`I#pragma vector=number`.
For a detailed description on how to define an interrupt routine in "C", refer to the IAR C-Compiler's user's guide.

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

embOS interrupt handler have to be used for interrupt sources running at priorities from 1 to 5

```
#pragma vector=TPU_INTVEC
__interrupt void UserInterrupt (void) {
  OS_EnterInterrupt();    // Tell embOS that interrupt code is running.
  HandleUserInterrupt();  // Call interrupt handler function
  OS_LeaveInterrupt();    // Tell embOS that interrupt code ends.
}
```

Any interrupt handler running at priorities from 1 to 5 has to be written according the code example above, regardless any other ***embOS*** API function is called.
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_EnterInterrupt()` as first function call, when interrupts should not be nested. It has to call `OS_EnterNestableInterrupt()`, when the related interrupt may be interruptible be higher priority interrupts.
- The interrupt handler has to call a user defined function which handles the interrupt. This function may use local variables and should clear the interrupt pending condition of the interrupting source if necessary. This user handler function may call other functions and may also call any other ***embOS*** function, but must not modify the interrupt priority.
- Finally the ***embOS*** interrupt handler has to call `OS_LeaveInterrupt()` when `OS_EnterInterrupt()` was called initially, or has to end with `OS_LeaveNestableInterrupt()` when a nestable interrupts was entered by `OS_EnterNestableInterrupt()`.

Differences between OS_EnterInterrupt() and OS_EnterNestableInterrupt()

`OS_EnterInterrupt()` should be used as entry function in an ***embOS*** interrupt handler, when the corresponding interrupt should not be interrupted by an other ***embOS*** interrupt. `OS_EnterInterrupt()` sets the interrupt priority of the CPU to 5 thus locking any other ***embOS*** interrupt, Fast interrupts are not disabled.
Interrupt handlers started with `OS_EnterInterrupt()` have to end with `OS_LeaveInterrupt()`.

`OS_EnterNestableInterrupt()` should be used as entry function in an ***embOS*** interrupt handler, when interruption by higher prioritized ***embOS*** interrupts should be allowed. `OS_EnterNestableInterrupt()` does not alter the interrupt priority of the CPU thus keeping all interrupts with higher priority enabled.

Interrupt handlers started with `OS_EnterNestableInterrupt()` have to end with `OS_LeaveNestableInterrupt()`.

Example of a *Fast interrupt* handler

Fast interrupt handler have to be used for interrupt sources running at priorities from 6 to 7 only

```
#pragma vector=TPU_INTVEC
__interrupt void FastUserInterrupt (void) {
  unsigned long Count;  // local variables are allowed
  Count = TPU_TCNT0;
  HandleCount(Count);   // Any function call except embOS functions is allowed
}
```

The rules for a *Fast interrupt* handler are as follows:

- Local variables may be used.
- Other functions may be called.
- ***embOS*** functions must not be called, nor direct, neither indirect.

9.5. Defining interrupt handlers for H8S CPUs in assembler

Even though not recommended, you may write interrupt handlers in assembler. These interrupt handlers have to follow the same rules as interrupt handlers written in “C”.
All interrupts running at low priority from 1 to 5 also have to start with `OS_EnterInterrupt()` or `OS_EnterNestableInterrupt()`. These macros are defined in RTOS.h.
This is required, because interrupts with low priorities may be interrupted by other interrupts calling ***embOS*** functions. The task switch from interrupt will only work if every ***embOS*** interrupt uses the same stack layout and starts with `OS_EnterInterrupt()` or `OS_EnterNestableInterrupt()`.
To get an idea how to write an interrupt handler in assembler, you may look at the list files of any module containing an ***embOS*** interrupt handler.

Example of an interrupt handler using OS_EnterInterrupt() written in assembler:

```
        ; ***** void __interrupt OS_ISR_Tick (void)
OS_ISR_Tick:
        STM.L   (ER4-ER6), @-ER7       ; Initially save registers ER4..ER6
                                       ; DO NOT SAVE ANY OTHER REGISTERS
                                       ; because task switch from int relies
                                       ; on this stack frame !
        ; ***** OS_EnterInterrupt();
        JSR     @OS_EnterIntFunc:24

        ; ***** _ISR_TickHandler();
        JSR     @_ISR_TickHandler:24   ; call interrupt handler function

        ; ***** OS_LeaveInterrupt();
        JSR     @OS_LeaveIntFunc:24

        LDM.L   @ER7+, (ER4-ER6)       ; Finally restore registers
        RTE                            ; return from interrupt
```

Example of an interrupt handler using OS_EnterNestableInterrupt():

```
        ; ***** void interrupt [TPU_INTVEC] OS_ISR_Tick (void)
OS_ISR_Tick:
        STM.L   (ER4-ER6), @-ER7       ; Initially save registers ER4..ER6
                                       ; DO NOT SAVE ANY OTHER REGISTERS
                                       ; because task switch from int relies
                                       ; on this stack frame !
        ; ***** OS_EnterInterrupt();
        JSR     @OS_EnterNestableIntFunc:24

        ; ***** _ISR_TickHandler();
        JSR     @_ISR_TickHandler:24   ; call interrupt handler function

        ; ***** OS_LeaveInterrupt();
        JSR     @OS_LeaveNestableIntFunc:24

        LDM.L   @ER7+, (ER4-ER6)       ; Finally restore registers
        RTE                            ; return from interrupt
```

9.6. Interrupt vector table

Normally there is no need to define a separate interrupt vector table when using IAR compiler for H8, as interrupt routines may be written in “C”-source and the interrupt vector table is generated automatically. If for some reason, you have to define a vector table as assembler file, please refer to IAR documentation.

10. Sleep / Standby Mode

Usage of the Sleep 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 Sleep mode works without any problems, because the ***embOS*** scheduler is activated on any timer interrupt.
The Software Standby-Mode may be used, if scheduling depends on those interrupts, which may release Software Standby-Mode. The real-time operating system is halted during the execution of the Software-Standby mode if the timer that the scheduler uses is supplied from internal clock. With external clock, the scheduler keeps working. ***embOS*** timer may be realized with external hardware which triggers one of the interrupt inputs of the CPU.
Hardware standby mode can not be used, as this mode can not be suspended by any interrupt.

11. Technical data

11.1. 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 values in the table are for the Large memory model and release build library.

Short description	ROM [byte]	RAM [byte]
Kernel	approx.1582	34
Add. Task	---	28
Add. Semaphore	---	6
Add. Mailbox	---	14
Add. Timer	---	14
Power-management	---	---

12. Files shipped with *embOS* H8 for IAR compiler

embOS for H8 and IAR compiler is shipped with documentation in PDF format and release notes as html.
The start project, source files, all libraries and additional files required for linker or emulator / simulator are located in the sub folder 'Start'. The distribution of ***embOS*** contains the following files:

Directory	File	Explanation
Start\	`Start*.ew*`	Start workspace and projects projects for IAR Embedded Workbench.
Start\CPU*\	`*.mac`	***embOS*** timer Interrupt simulation macro for IAR C-Spy simulator
Start\CPU*\	`RTOSInit*.c`	CPU specific hardware setup functions used for ***embOS***
Start\Inc\	`RTOS.h`	***embOS*** API header file. To be included in any file using ***embOS*** functions
Start\Lib\	`*.r37`	***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 or debug builds.
CPU*\Sample\	`*.*`	Sample applications.
GenOsSrc\	`*.*`	***embOS*** sources (Source version only)
	`*.Bat`	Batch files to build ***embOS*** libraries from sources (Source version only)

embOSView and the manuals are found in the root directory of the distribution.

13. Index