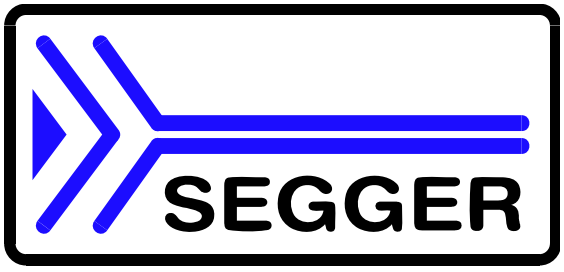
embOS

Real Time Operating System

CPU & Compiler specifics for H8/H8S/H8SX cores using Renesas Tools for H8

Document revision 2

Date: March 14, 2008



A product of SEGGER Microcontroller Systeme GmbH

www.segger.com

Disclaimer

Specifications written in this document are believed to be accurate, but are not guaranteed to be entirely free of error. The information in this manual is subject to change for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, SEGGER MICROCONTROLLER SYSTEME GmbH (the manufacturer) assumes no responsibility for any errors or omissions. The manufacturer makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. The manufacturer specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller Systeme GmbH

Heinrich-Hertz-Str. 5
D-40721 Hilden

Germany

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

Manual versions

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

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

Manual version	Date	By	Explanation
1.00	070823	TS	First version
2.00	080312	TS	Add cpu specifics for H8S 2600 cpu’s

Software versions

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

About this document

Assumptions

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

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

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

How to use this manual

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

Typographic conventions for syntax

This manual uses the following typographic conventions:

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

Table 1.1: Typographic conventions

SEGGER Microcontroller Systeme GmbH develops and distributes software development tools and ANSI C software components (middleware) for embedded systems in several industries such as telecom, medical technology, consumer electronics, automotive industry and industrial automation.

SEGGER's intention is to cut software development-time for embedded applications by offering compact flexible and easy to use middleware, allowing developers to concentrate on their application.

Our most popular products are emWin, a universal graphic software package for embedded applications, and embOS, a small yet efficent real-time kernel. emWin, written entirely in ANSI C, can easily be used on any CPU and most any display. It is complemented by the available PC tools: Bitmap Converter, Font Converter, Simulator and Viewer. embOS supports most 8/16/32-bit CPUs. Its small memory footprint makes it suitable for single-chip applications.

Apart from its main focus on software tools, SEGGER developes and produces programming tools for flash microcontrollers, as well as J-Link, a JTAG emulator to assist in development, debugging and production, which has rapidly become the industry standard for debug access to ARM cores.

Corporate Office:
http://www.segger.com

United States Office:
http://www.segger-us.com

EMBEDDED SOFTWARE (Middleware)

emWin
Graphics software and GUI

emWin is designed to provide an efficient, processor- and display controller-independent graphical user interface (GUI) for any application that operates with a graphical display. Starterkits, eval- and trial-versions are available.

embOS
Real Time Operating System

embOS is an RTOS designed to offer the benefits of a complete multitasking system for hard real time applications with minimal resources. The profiling PC tool embOSView is included.

emFile
File system

emFile is an embedded file system with FAT12, FAT16 and FAT32 support. emFile has been optimized for minimum memory consumption in RAM and ROM while maintaining high speed. Various Device drivers, e.g. for NAND and NOR flashes, SD/MMC and CompactFlash cards, are available.

USB-Stack
USB device stack

A USB stack designed to work on any embedded system with a USB client controller. Bulk communication and most standard device classes are supported.

SEGGER TOOLS

Flasher
Flash programmer

Flash Programming tool primarily for microcontrollers.

J-Link
JTAG emulator for ARM cores

USB driven JTAG interface for ARM cores.

J-Trace
JTAG emulator with trace

USB driven JTAG interface for ARM cores with Trace memory. supporting the ARM ETM (Embedded Trace Macrocell).

J-Link / J-Trace Related Software

Add-on software to be used with SEGGER's industry standard JTAG emulator, this includes flash programming software and flash breakpoints.

Table of Contents

Chapter 1

About this document

This guide describes how to use embOS for H8 Real Time Operating System for the RENESAS H8 series of microcontroller using Renesas HEW4 and the RENESAS h8 compiler.

How to use this manual

This manual describes all CPU and compiler specifics for embOS using H8 CPUs with Renesas HEW4 workbench and h8 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 Renesas compiler and HEW. If you have no experience using embOS, you should follow this introduction, even if you do not plan to use HEW 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 Renesas compiler.

Chapter 2

Using embOS with HEW work-bench

The following chapter describes how to install and work with embOS for H8 CPUs and HEW 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 Renesas HEW workbench to develop your application, no further installation steps are required. You will find a prepared sample workspace and a start project for different H8 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 HEW Embedded Workbench for your application development in order to become familiar with embOS.

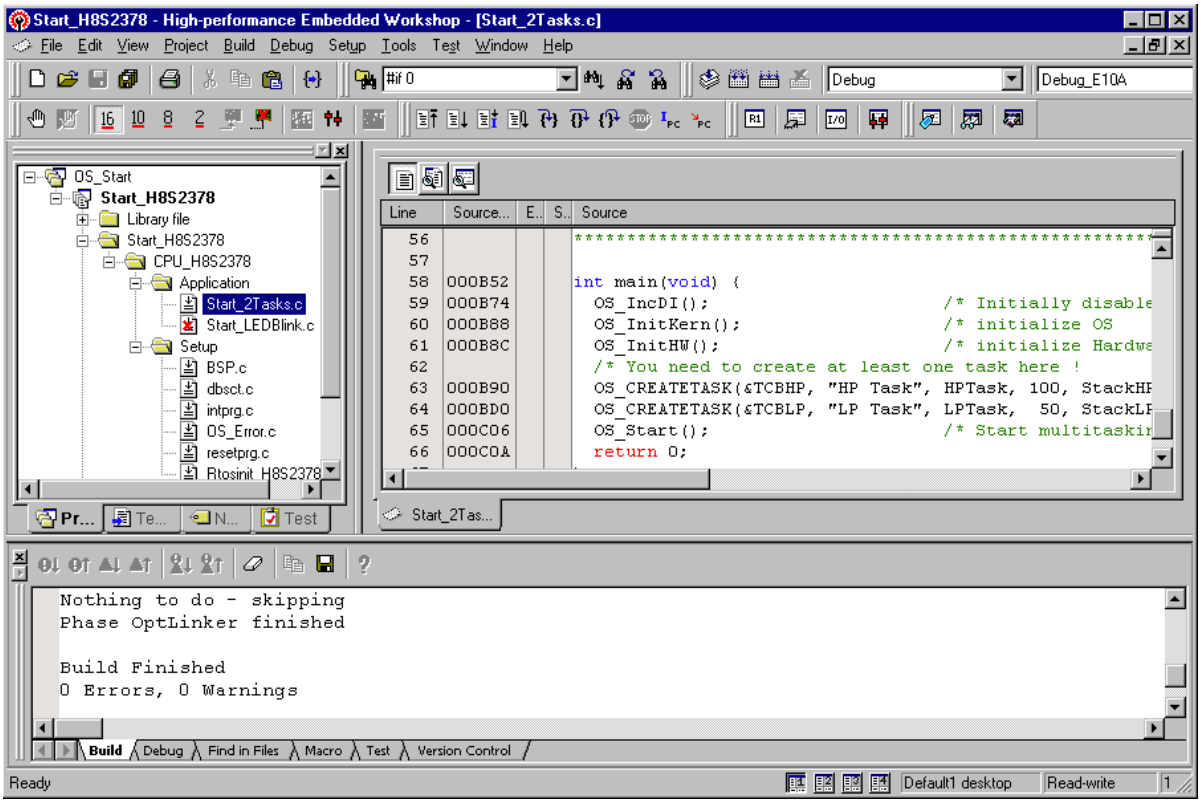
embOS does in no way rely on the HEW Embedded Workbench, it may be used without the workbench using batch files or a make utility without any problem.

2.2 First steps

After installation of embOS (See "Installation" in chapter 2.1) you are able to create your first multitasking application. You received several ready to go sample start workspace for Renesas H8 CPUs which might be used as a starting point for your applications.Your embOS distribution contains one folder 'Start' which contains the sample start workspaces and different subfolders containing the project and all CPU specific files required for the projects.Every additional files used to build your embOS application are located in the Start folder and its subfolders.

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 e.g. the start workspace 'Start_H8S2378.hws'. (e.g. by double clicking it)·You may select the Configuration "Debug" and session "Debug_E10A" which allows downloading and debugging of the the sample application into target FLASH using the E10A debugger.
- Build the start project. After building the start project, your screen should look like follows



2.3 The sample application Start_LEDBlink.c

The following is a printout of the sample application Start_LEDBlink.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 sus-pend for the specified time and continue execution.

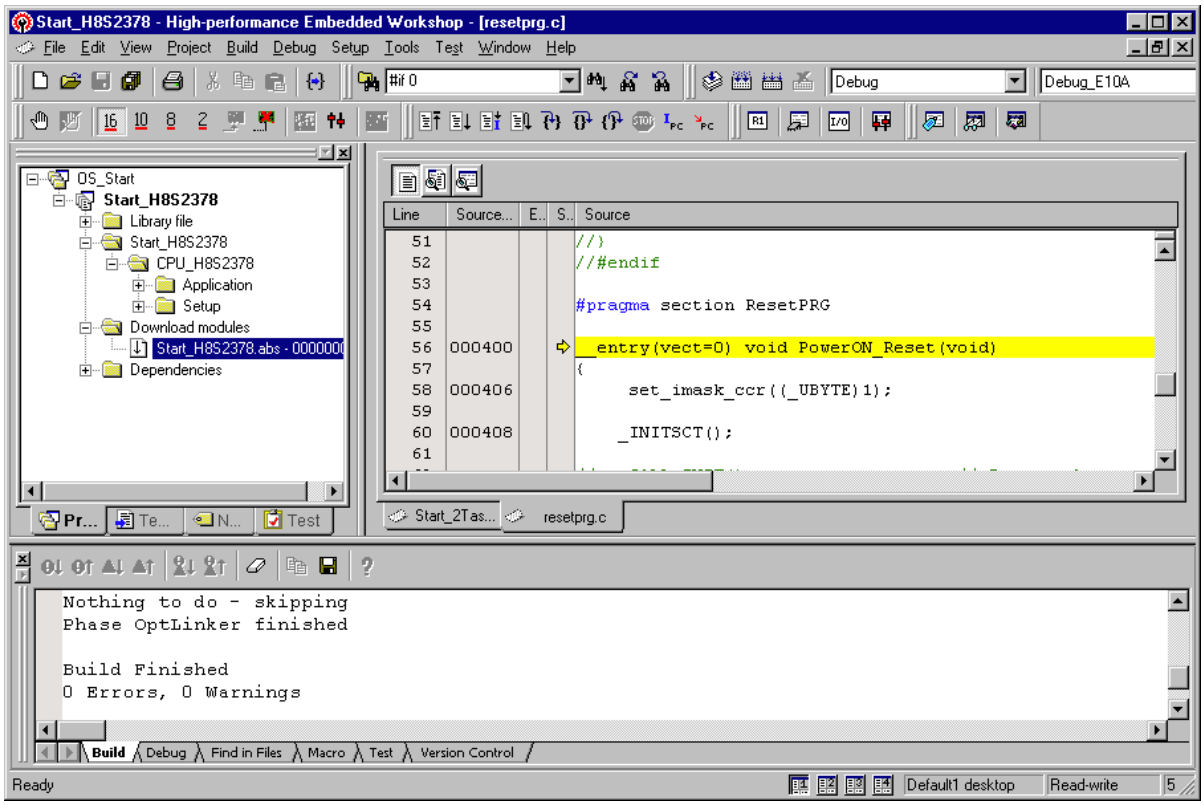
```
/*********************************************************************
*               SEGGER MICROCONTROLLER SYSTEME GmbH                  *
*       Solutions for real time microcontroller applications         *
**********************************************************************
*                                                                    *
*       (c) 1995 - 2007  SEGGER Microcontroller Systeme GmbH         *
*                                                                    *
*       www.segger.com     Support: support@segger.com               *
*                                                                    *
**********************************************************************

----------------------------------------------------------------------
File    : Start_LEDBlink.c
Purpose : Sample program for OS running on EVAL-boards with LEDs
--------- END-OF-HEADER ----------------------------------------------*/

#include "RTOS.h"
#include "BSP.h"

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

static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_Delay (50);
  }
}

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
  }
}

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

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

/****** End of file *************************************************/
```


2.4 Using debugging tools to debug the application

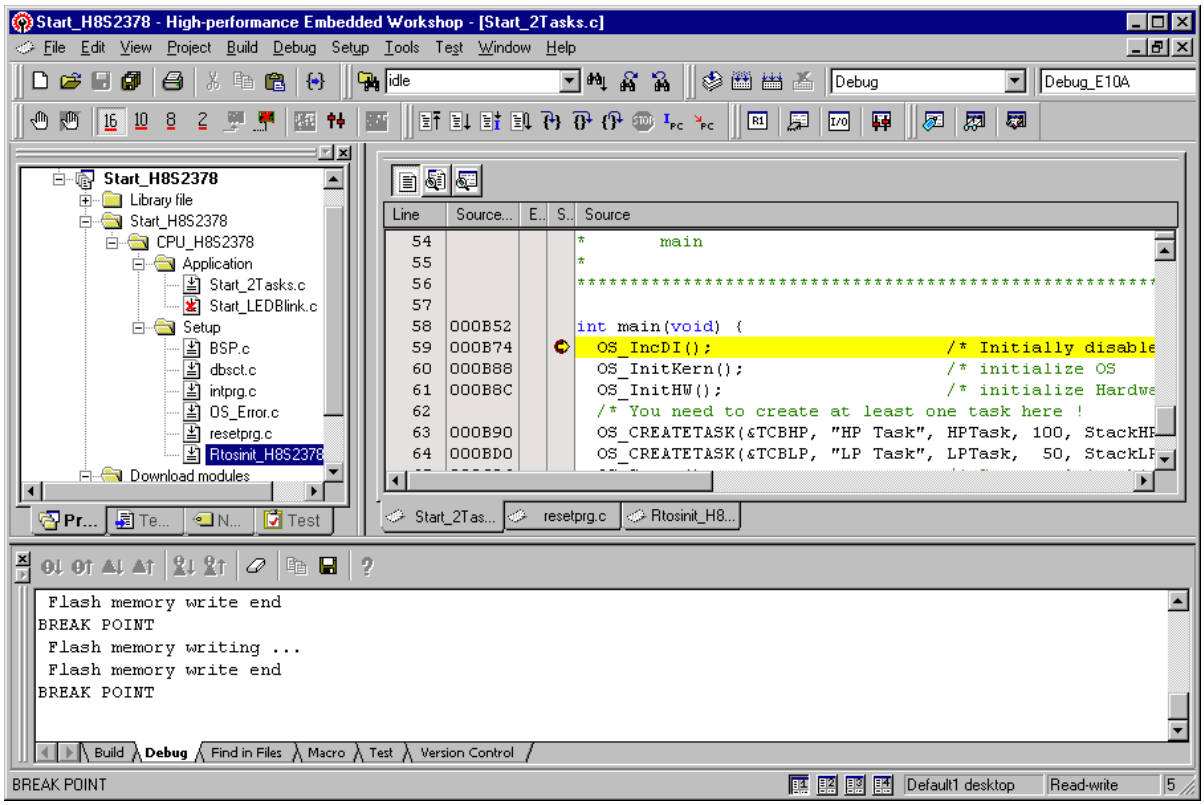
The embOS start project contains a configuration which may be used to download and debug the sample application into the target FLASH using the E10A emulator.
You should use this one to run the sample start application and become familiar with embOS. You may alternatively generate a session for the H8 simulator to run the sample application using the simulator.
The following description shows a sample session with the E10A debugger. A simulator session should look similar

2.4.1 Using Renesas E10A emulator and HEW workbench

After building the application, connect to the target, download the generated output file, and perform a reset command. The debug window will show the startup code:

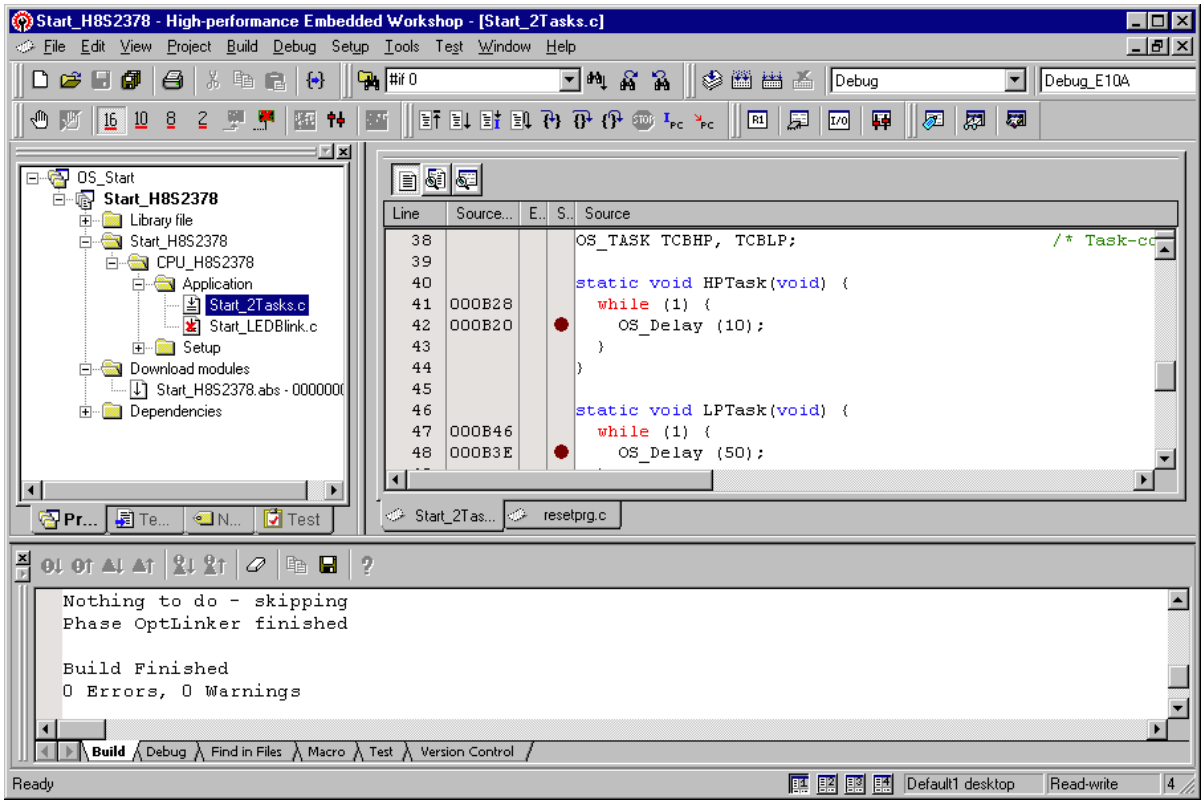


You may single-step through the startup code to reach main(), or you may open the “Start_LEDBlink.c” file and set a breakpoint at main:

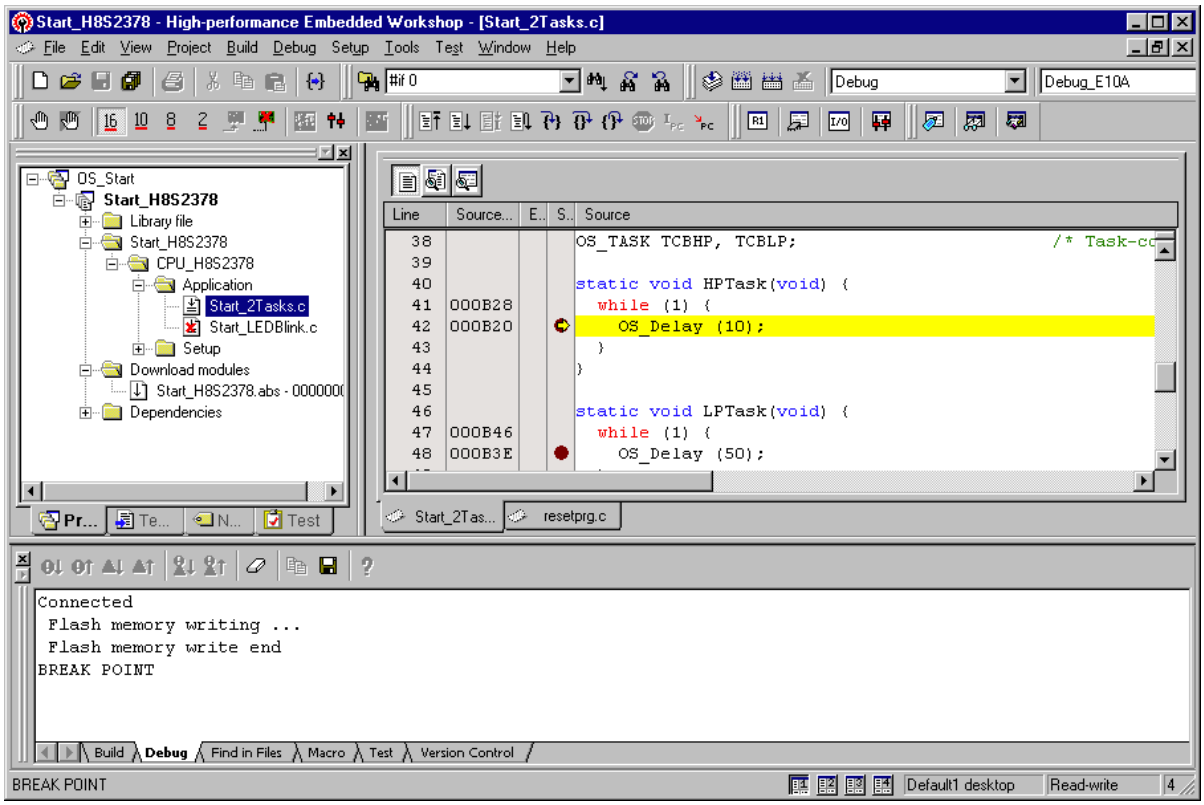


When you then issue a “Go” command, you will reach at main().
OS_IncDI() disables interrupts and tells embOS, that interrupts should not be enabled during OS_InitKern().
OS_InitKern()initializes embOS variables. If OS_incDI() was not called before, interrupts will be enabled. As this function is part 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_Start() is the last line executed in main, since it starts multitasking and does not return.

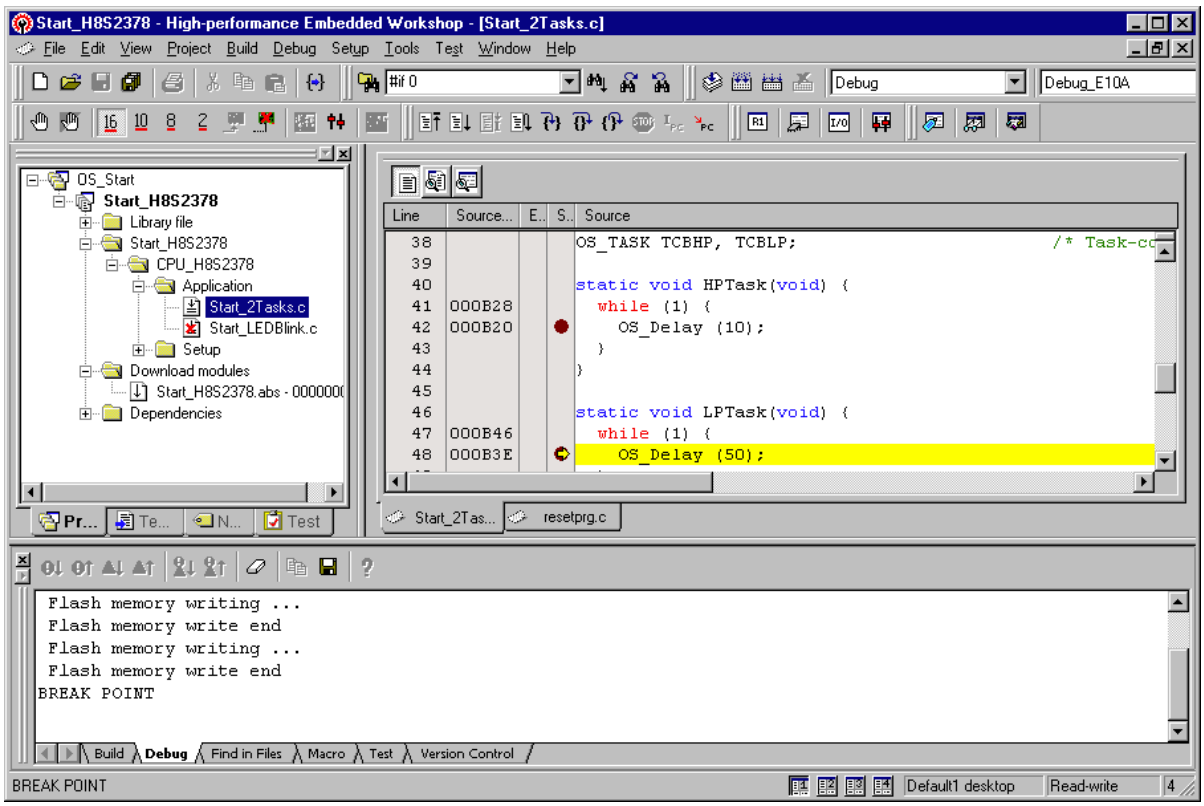
Before you step into OS_Start(), you should set two break points in the two tasks as shown below



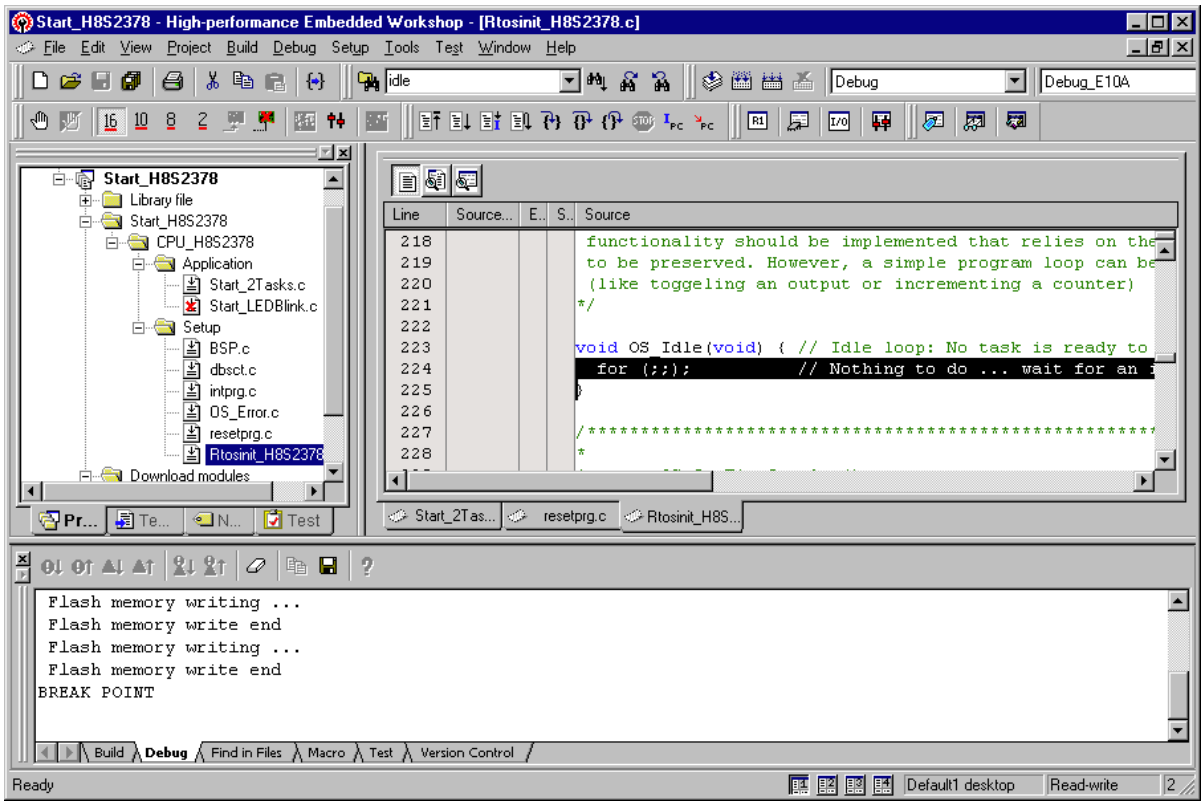
As OS_Start() is part of the embOS library, you can step through it in disassembly mode only. You may press GO, step over OS_Start(), or step into OS_Start() in disassembly mode until you reach the highest priority task.



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



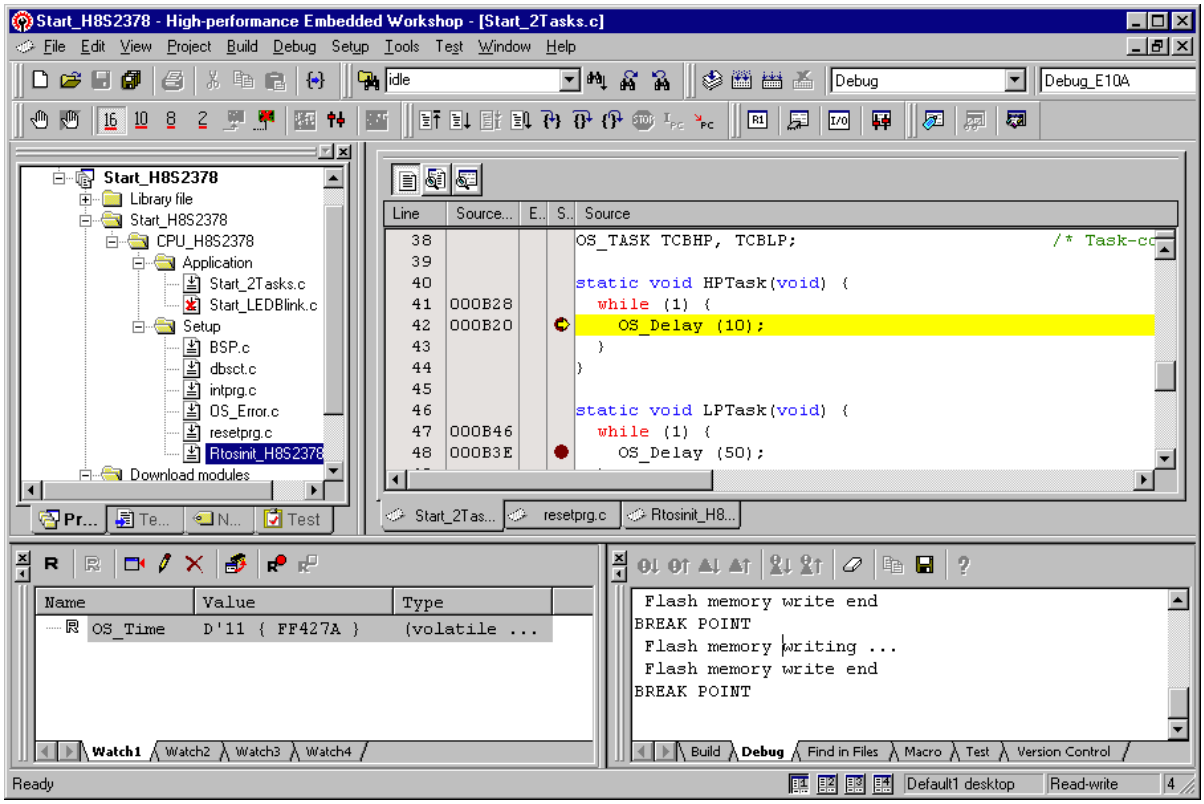
Continuing to step through the program, there is no other task ready for execu-tion. embOS will suspend LPTask and switch to the idle-loop, which is always executed if there is nothing else to do:



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 to 'Go' command:

i



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.

2.5 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 debugger 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 programcode 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.

2.6 Build your own application

To build your own application, you may 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.
You may also add all necessary files for embOS into your own project as described below.

2.7 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 Start\Inc\
  This header file declares all embOS API functions and data types and has to be included in any source file using embOS functions.
- OS_Config.h from the Start\Inc\ subfolder. This file may be used to define different options for different project configurations. Normally, this file is used to define the library types used for debug and release builds. You may add other options to this file.
- 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 Start\Lib\ subfolder. Please set the appropriate OS_LIBMODE define according to the chosen library. This is normally done in the file OS_Config.h
- OS_Error.c from subfolder Setup\ of the CPU specific subfolder, if any library other than Release build library is used in your project.

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

2.8 Add your own code

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

2.9 Change library mode

For your application you may wish to use a different embOS library. For debugging and program development you should use an embOS debug library. For your final application you may wish to use an embOS release library.
Therefore you may have to replace the embOS library in your project or target:

- Add the appropriate library from the Lib-subdirectory to your project.
- Remove the previous library from your project or exclude it from build.
- Set the appropriate OS_LIBMODE_* define as tool chain compiler option. Normally done in the OS_Config.h file.

Refer to chapter 5 about the library naming conventions to select the correct library.

Chapter 3

H8 specifics

3.1 CPU modes

embOS for H8 for HEW and Renesas H8 compiler is delivered with libraries for the default options and compiler settings.

3.2 Available libraries

embOS is shipped with 26 different libraries for H8 CPUs.

os<CPU><LibMode>.lib

Parameter	Meaning	Valzue
CPU	CPU Variant	H8: H8/300 CPUs H8S2000: H8S/2000 CPUs H8S2600: H8S/2600 CPUs
LIBMODE	Library mode	XR: Extreme Release R: Release S: Stack check SP: Stack check + profiling D: Debug DP: Debug + profiling DT: Debug + profiling + Trace

Table 3.1:

This results in 28 different libraries delivered with embOS.
For the different library versions, the following defines have to be set:

Library mode	Meaning	Define
XR	Extreme release	OS_LIBMODE_XR
R	Release	OS_LIBMODE_R
S	Stack check	OS_LIBMODE_S
SP	Stack check + profiling	OS_LIBMODE_SP
D	Debug	OS_LIBMODE_D
DP	Debug + profiling	OS_LIBMODE_DP
DT	Debug + profiling + Trace	OS_LIBMODE_DT

Table 3.2:

When using HEW workbench, please check the following points:·

- One embOS library is part of your project (included in one group of your target).
- The appropriate define according to embOS library mode is set as compiler pre-processor option for your project. May be defined in OS_Config.h.

3.3 Distributed project files

The distribution of embOS for H8 and HEW compiler contains severail start project for Renesas H8 CPUs.The start project contains an embOS debug and profiling library which should be used during program development.

3.4 H8 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.

3.5 Clock settings 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.

3.6 Clock settings for UART used for embOSView

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 ensure correct time base clock for baudrate generator used 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 tick timer.·
- 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.

3.7 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.

3.8 embOS hardware timer selection

embOS for H8 CPUs is prepared to use one Timer (TPU) channel as time base timer. The initialization code and interrupt handler are delivered in source code and are located in RTOSInit_*.c. If another timer has to be used, the interrupt vector table entries in “intprg.c” have to be modified accordingly.

3.9 UART for embOSView

Any UART of the H8 CPU may be used as communication channel for embOS-View which enables profiling analysis during runtime.The initialization code and interrupt handler are delivered in source code and are located in RTOSInit_*.c.

Chapter 4

Stacks

4.1 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 CPU. As H8 CPUs have a 32 bit stack pointer, the whole memory area can be used as task stack.
Please note, that the task stacks have to be aligned at EVEN addresses. To ensure proper alignment, implement the 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 CPU, the stack size for the CPU registers is 38 bytes.
As the H8 CPUs do not support a separate interrupt stack, all interrupts may run on the task stacks as well. Therefore we recommend at least a minimum of 256 bytes for task stacks.

4.2 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 softwaretimers also use the system-stack, the actual stack requirements depend on the application.
Because the H8 CPU does not support a separate interrupt stack, all interrupts may also run on the system stack.
The stack used as system stack is the one defined as STACK in the "S" section in linker command description. The stack size is defined in the "stackct.h" header file. We recommend at least a minimum of 256 bytes.

4.3 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.

4.4 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.

Chapter 5

Interrupts

5.1 Interrupts with H8 CPUs

The following chapter describes interrupt specifics of H8 CPUs and the interrupt modes used with embOS.

5.2 Interrupt processing with H8 CPUs

H8/2000 and H8SX CPUs support a priority controlled interrupt mode. This mode supports the following features:

- Interrupt priority registers to assign 7 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 processing is as follows:

- The CPU-core receives an interrupt request from the interrupt controller.
- If interrupts are enabled for the priority of the interrupting device, the inter-rupt is executed.
- The CPU stores PC, the CCR register and the EXR register onto the current stack.
- The interrupt mask level in the EXR register 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, CCR register and status register from the stack.

For more details, refer to the RENESAS manuals.

5.3 Fast interrupts with H8 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 **Fast interrupts**.
The default level limit for fast interrupts is set to 4, meaning, any interrupt with level 5 or above is never disabled and can be accepted anytime.

You must not execute any embOS function from within a fast interrupt function.

5.4 Interrupt priorities with embOS for H8 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 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 Fast 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 fast interrupts can be set at runtime by a call of OS_SetFastIntPriorityLimit().

5.5 Defining interrupt handlers for H8 CPUs in "C"

Routines preceded by the keywords "__interrupt" save & restore the temporary registers and all registers they modify onto the stack and return with RTE.
The interrupt function has to be declared in the interrupt vector table file "intprg.c". The interrupt handler itself may be implemented in any source file. Default dummy interrupt handler are delivered in the source file "intprg.c" . The interrupt handler used by embOS are implemented in the CPU specific RTOSInit_*.c file.

Example of an embOS interrupt handler

embOS interrupt handler have to be used for interrupt sources running at all priorities up to the user definable interrupt priority level limit for fast interrupts.

```
__interrupt void OS_ISR_Tick(void) {
  OS_CallNestableISR(_IsrTickHandler);
}
```

Any interrupt handler running at priorities from 1 to 4 has to be written accord-ing 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_CallISR(), when interrupts should not be nested. It has to call OS_CallNestableISR(), when nesting should be allowed.
- The interrupt handler must not perform any other operation, calculation or function call. This has to be done by the local function called from OS_CallISR() or OS_CallNestableISR().

Differences between OS_CallISR() and OS_CallNestableISR()

OS_CallISR() should be used as entry function in an embOS interrupt han-dler, when the corresponding interrupt should not be interrupted by another embOS interrupt. OS_CallISR() sets the interrupt priority of the CPU to the user definable "fast" interrupt priority level, thus locking any other embOS in-terrupt, Fast interrupts are not disabled.
OS_CallNestableISR() should be used as entry function in an embOS interrupt handler, when interruption by higher prioritized embOS interrupts should be allowed. OS_CallNestableISR() does not alter the interrupt priority of the CPU, thus keeping all interrupts with higher priority enabled.

Example of a Fast interrupt handler

Fast interrupt handler have to be used for interrupt sources running at priorities above the user definable interrupt priority limit.

```
__interrupt void FastUserInterrupt (void) {
  unsigned long Count;  // local variables are allowed
  Count = TPU_TCNT0;
  HandleCount(Count);   // Any function call except embOS functions is al-lowed
}
```

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.
- The priority of the interrupt has to be above the user definable priority limit for fast interrupts.

5.6 OS_SetFastIntPriorityLimit(): Setting the interrupt priority limit for fast interrupts

The interrupt priority limit for fast interrupts is set to 4 by default. This means, all interrupts with higher priority from 5 to 7 will never be disabled by embOS.

Description

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

Prototype

void OS_SetFastIntPriorityLimit(unsigned int Priority)

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

Table 5.1:

Return value

NONE.

Add. information

To disable fast 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. In the default projects, OS_SetFastIntPriorityLimit() is called from OS_IntHW() in RTOSInit_*.c.
All interrupts running at low priority from 1 to the user definable priority limit for fast interrupts have to call OS_CallISR() or OS_CallNestableISR() regardless any other embOS function is called in the interrupt handler.
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. This can only be guaranteed when OS_CallISR() or OS_CallNestableISR() is used.
Any interrupts running above the fast interrupt priority limit must not call any embOS function.

5.7 Interrupt vector table

The sample start project uses startup code and an interrupt vector table written in “C” source and header files.

For embOS, the embOS timer tick interrupt vector is defined in the vector table. The embOS timer interrupt handler itself is located in the in the source code file RTOSInit_*.c.

Chapter 6

Stop / Wait mode

6.1 Saving power

In case your controller does support some kind of power saving mode, it should bepossible 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 somespecial sequence in function OS_Idle(), which you can find in embOS module RTOSIinit_*.c.

Chapter 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. Using H8 cpu, the minimum ROM requirement for the kernel itself is about 2.500 bytes. In the table below, you find the minimum RAM size for embOS resources. The sizes depend on selected embOS library mode; the table below is for a release build.

embOS resource	RAM [bytes]
Task control block	28
Resource semaphore	14
Counting semaphore	6
Mailbox	16
Software timer	14

Chapter 8

Files shipped with embOS

8.1 Files included in embOS

Directory	File	Explanation
root	*.pdf	Generic API and target specific documentation
root	embOSView.exe	Utility for runtime analysis, described in generic documentation
root	Release.html	Version control document
Start\Start*.hws\	*.*	Sample workspaces and project files for Renesas HEW
Start\Start*\CPU**\Appli-cation\	*.*	Sample programs to serve as a start
Start\Start*\CPU*\Setup\	*.*	CPU specific hardware routines
Start\Inc\	BSP.h	Include file for BoardSupport packages, to be included in every "C"-file using BSP-functions
Start\Inc\	OS_Config.h	Include file for embOS library mode configuration, included by RTOS.h
Start\Inc\	RTOS.h	Include file for embOS, to be included in every "C"-file using embOS-functions
Start\Lib\	os*.lib	embOS libraries

Any additional files shipped serve as example.

