embOS

Real Time Operating System

CPU & Compiler specifics for RENESAS SH2 CPUs and RENESAS HEW4

Document Rev. 1

SEGGER

A product of Segger Microcontroller Systeme GmbH

www.segger.com

Contents

1. About this document

This guide describes how to use ***embOS*** for SH2 Real Time Operating System for the RENESAS SH2 series of microcontroller using Renesas HEW4 and the RENESAS shc compiler.

1.1. How to use this manual

This manual describes all CPU and compiler specifics for ***embOS*** using SH2 CPUs with Renesas HEW4 workbench and shc 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 SH2 CPUs and Renesas compiler.

2. Using *embOS* with HEW Workbench

The following chapter describes how to install and work with ***embOS*** for SH2 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 an SH7086 CPU, 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.

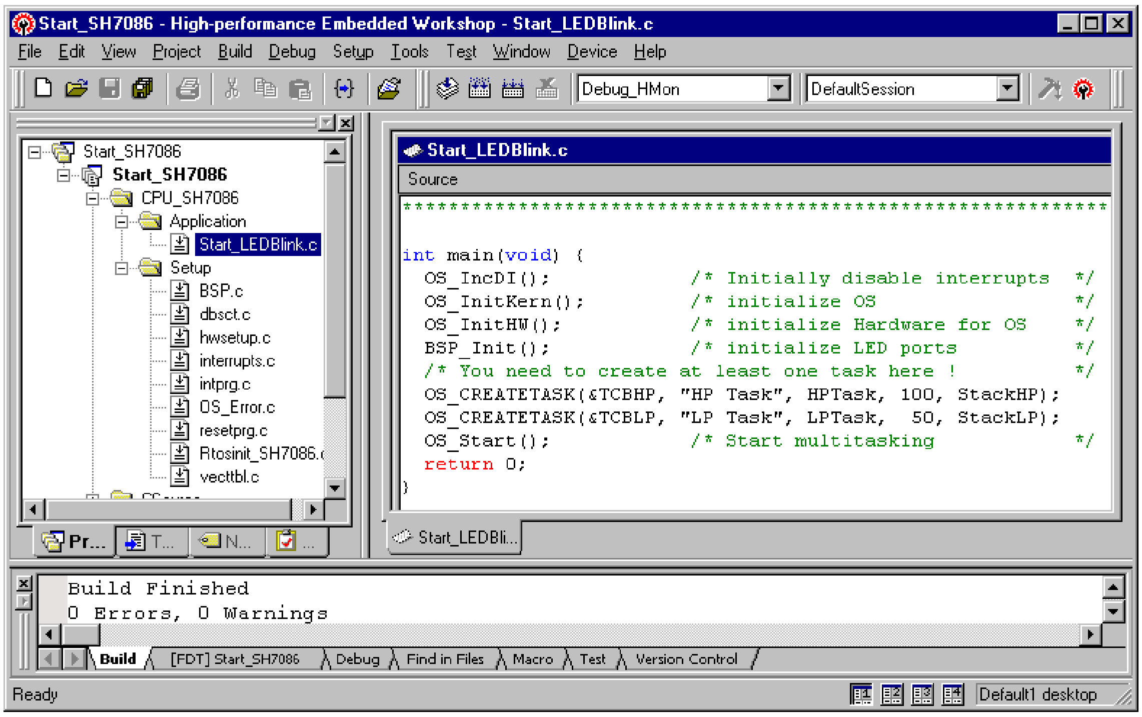
2.2. First steps

After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received a ready to go sample start workspace for an SH7086 CPU which might be used as a starting point for your applications.

Your ***embOS*** distribution contains one folder ‘Start’ which contains the sample start workspace and a subfolder Start_SH7086 containing the project and all CPU specific files required for this project.
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 the start workspace ‘Start_SH7086.hws’. (e.g. by double clicking it)
- You may select the Configuration “Debug_HMon” and session “Session_SH2_HMon” which allows downloading and debugging of the the sample application into target RAM using the E8 emulator.
- Build the start project

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    */
  BSP_Init();                  /* initialize LED ports          */
  /* 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;
}
```

```
Build Finished
0 Errors, 0 Warnings
```

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 suspend for the specified time and continue execution.

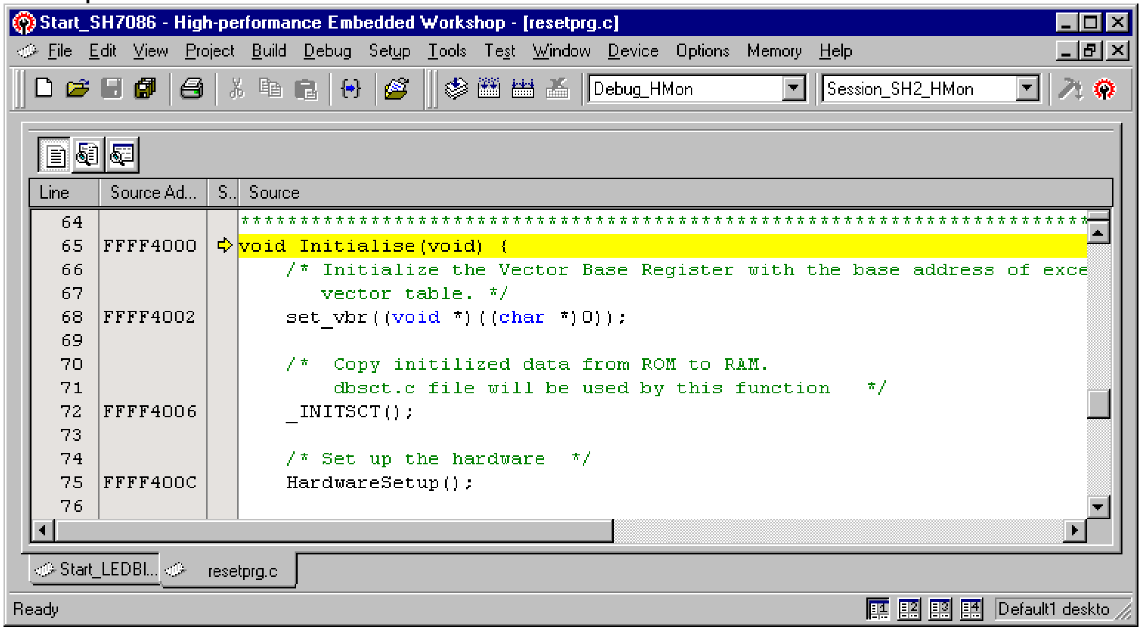
```
/*********************************************************************
*               SEGGER MICROCONTROLLER SYSTEME GmbH                  *
*       Solutions for real time microcontroller applications         *
**********************************************************************
*                                                                    *
*       (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 *************************************************/
```

3. 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 RAM using the E8 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 SH2 simulator to run the sample application using the simulator.
The following description shows a sample session with the E8 emulator. A simulator session should look similar.

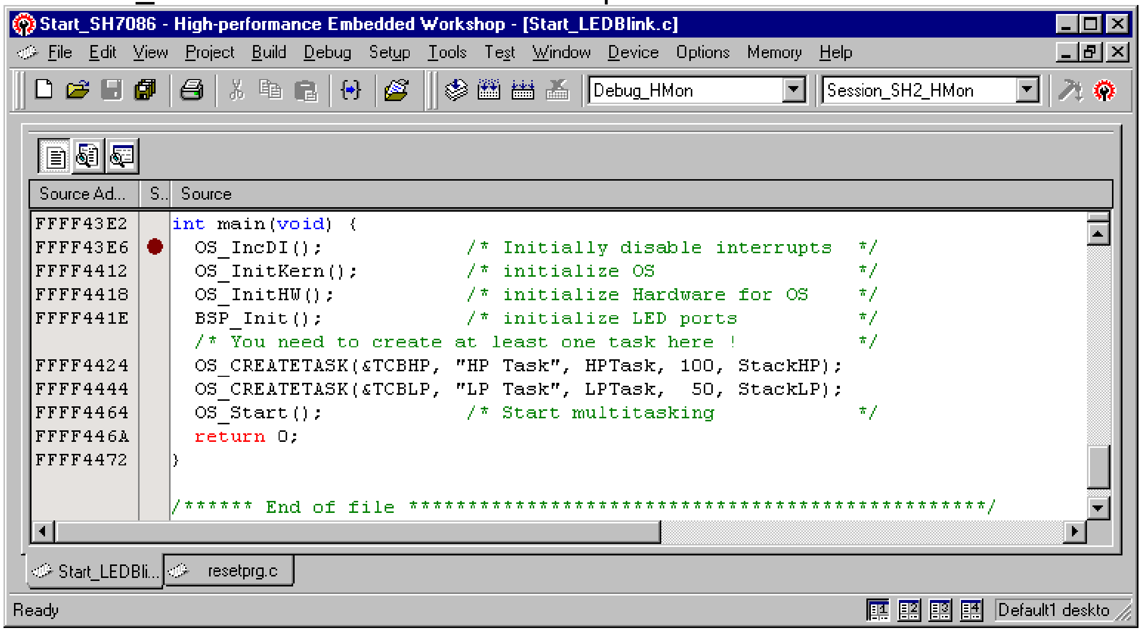
3.1. Using Renesas E8 emulator and HMon

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:


Start_SH7086 - High-performance Embedded Workshop - [resetprg.c]
File Edit View Project Build Debug Setup Tools Test Window Device Options Memory Help
Debug_HMon
Session_SH2_HMon
Line
Source Ad...
S..
Source
Start_LEDBl...
resetprg.c
Ready
Default1 deskto

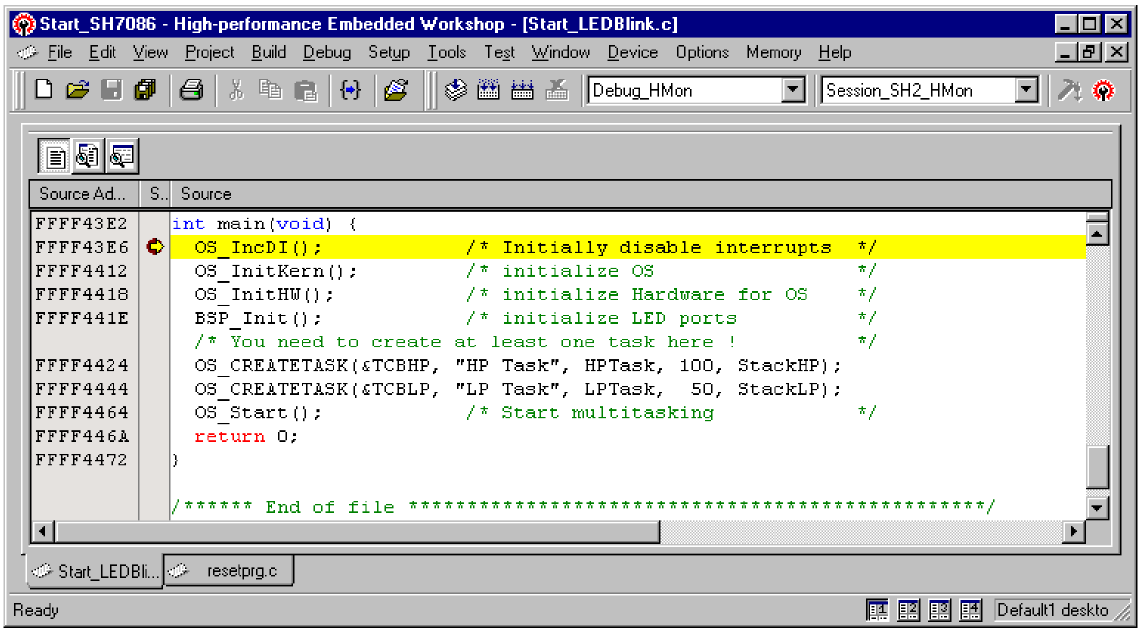
```
          ************************************************************************
FFFF4000  void Initialise(void) {
              /* Initialize the Vector Base Register with the base address of exce
                 vector table. */
FFFF4002      set_vbr((void *)((char *)0));

              /*  Copy initilized data from ROM to RAM.
                  dbsct.c file will be used by this function    */
FFFF4006      _INITSCT();

              /* Set up the hardware   */
FFFF400C      HardwareSetup();
```

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:


Start_SH7086 - High-performance Embedded Workshop - [Start_LEDBlink.c]
File Edit View Project Build Debug Setup Tools Test Window Device Options Memory Help
Debug_HMon
Session_SH2_HMon
Source Ad...
S..
Source
Start_LEDBli...
resetprg.c
Ready
Default1 deskto

```
FFFF43E2    int main(void) {
FFFF43E6  ●   OS_IncDI();              /* Initially disable interrupts   */
FFFF4412      OS_InitKern();           /* initialize OS                  */
FFFF4418      OS_InitHW();             /* initialize Hardware for OS     */
FFFF441E      BSP_Init();              /* initialize LED ports           */
              /* You need to create at least one task here !             */
FFFF4424      OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
FFFF4444      OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
FFFF4464      OS_Start();              /* Start multitasking             */
FFFF446A      return 0;
FFFF4472    }

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

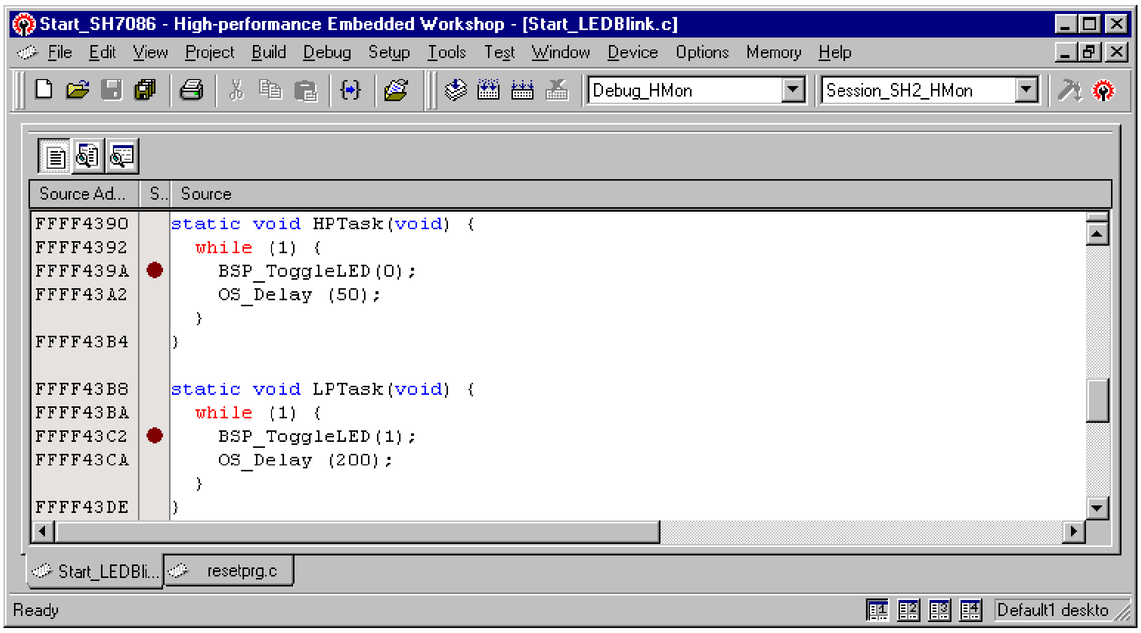
When you then issue a “Go” command, you will reach at main().


Start_SH7086 - High-performance Embedded Workshop - [Start_LEDBlink.c]
File Edit View Project Build Debug Setup Tools Test Window Device Options Memory Help
Debug_HMon
Session_SH2_HMon
Source Ad... S.. Source
Start_LEDBli...
resetprg.c
Ready
Default1 deskto

```
FFFF43E2   int main(void) {
FFFF43E6     OS_IncDI();              /* Initially disable interrupts  */
FFFF4412     OS_InitKern();           /* initialize OS                 */
FFFF4418     OS_InitHW();             /* initialize Hardware for OS    */
FFFF441E     BSP_Init();              /* initialize LED ports          */
             /* You need to create at least one task here !            */
FFFF4424     OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
FFFF4444     OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
FFFF4464     OS_Start();              /* Start multitasking            */
FFFF446A     return 0;
FFFF4472   }

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

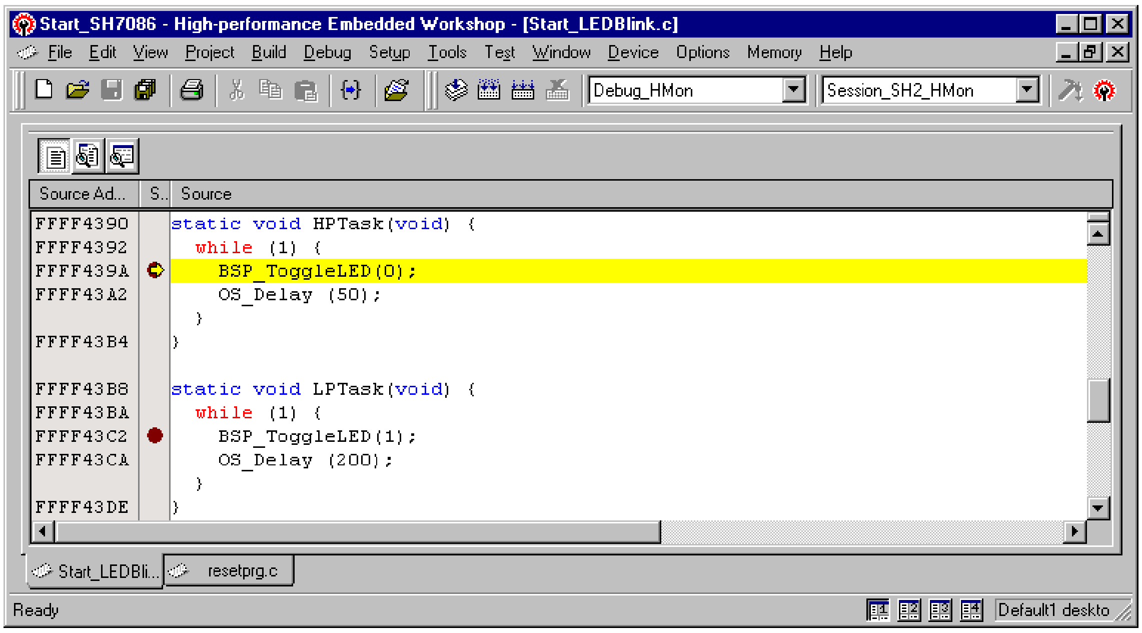
`OS_IncDI()` disables interrupts and tells ***embOS***, that interrupts should not be enabled during `OS_InitKern()`.
`OS_InitKern()`initializes ***embOS*** –Variables. If `OS_incDI()` was not called before, interrupts will be enabled. As this function is part of the ***embOS*** library, you may step 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


Start_SH7086 - High-performance Embedded Workshop - [Start_LEDBlink.c]
File Edit View Project Build Debug Setup Tools Test Window Device Options Memory Help
Debug_HMon
Session_SH2_HMon
Source Ad... S.. Source
Start_LEDBli...
resetprg.c
Ready
Default1 deskto

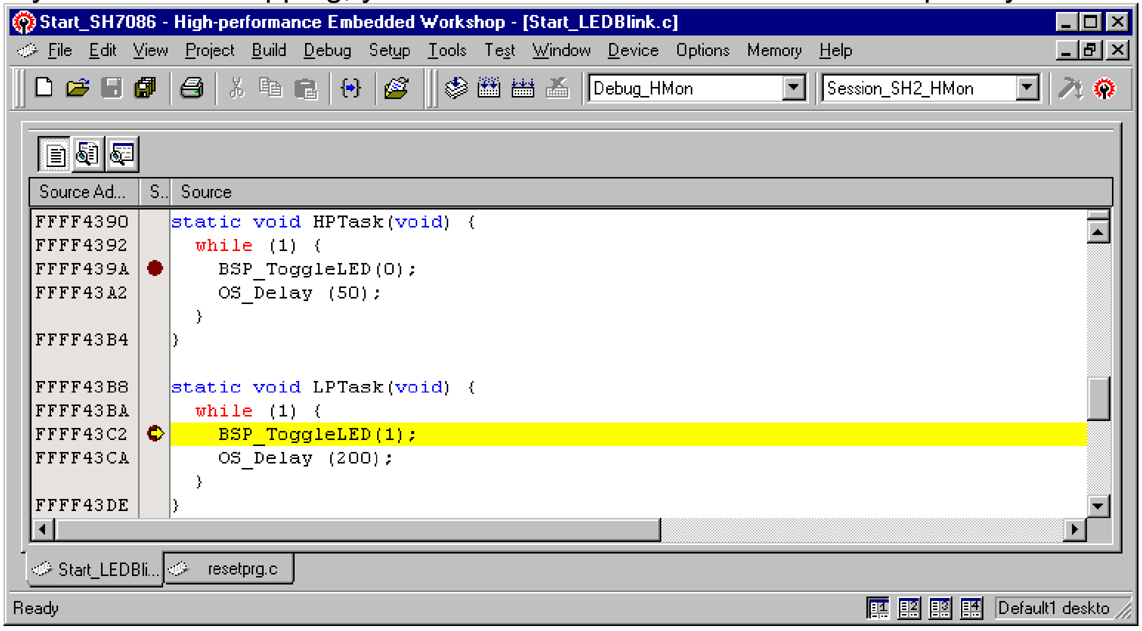
```
FFFF4390   static void HPTask(void) {
FFFF4392     while (1) {
FFFF439A  ●    BSP_ToggleLED(0);
FFFF43A2       OS_Delay (50);
             }
FFFF43B4   }

FFFF43B8   static void LPTask(void) {
FFFF43BA     while (1) {
FFFF43C2  ●    BSP_ToggleLED(1);
FFFF43CA       OS_Delay (200);
             }
FFFF43DE   }
```

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.


Start_SH7086 - High-performance Embedded Workshop - [Start_LEDBlink.c]
File Edit View Project Build Debug Setup Tools Test Window Device Options Memory Help
Debug_HMon
Session_SH2_HMon
Source Ad... S.. Source
Start_LEDBli... resetprg.c
Ready
Default1 deskto

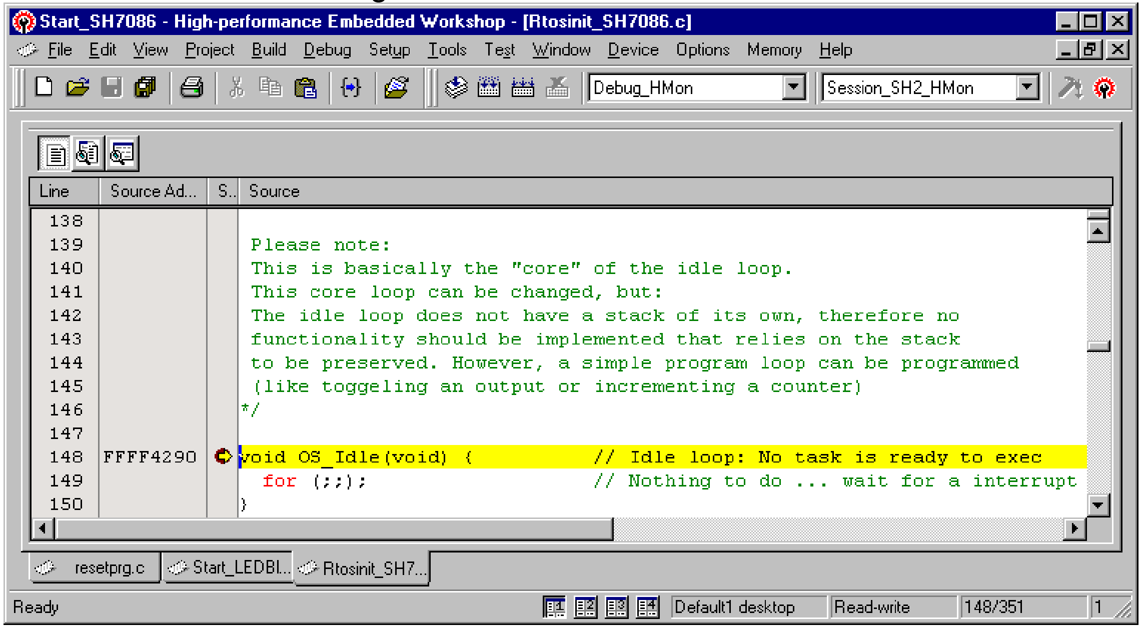
```
FFFF4390   static void HPTask(void) {
FFFF4392     while (1) {
FFFF439A       BSP_ToggleLED(0);
FFFF43A2       OS_Delay (50);
             }
FFFF43B4   }

FFFF43B8   static void LPTask(void) {
FFFF43BA     while (1) {
FFFF43C2       BSP_ToggleLED(1);
FFFF43CA       OS_Delay (200);
             }
FFFF43DE   }
```

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


Start_SH7086 - High-performance Embedded Workshop - [Start_LEDBlink.c]
File Edit View Project Build Debug Setup Tools Test Window Device Options Memory Help
Debug_HMon
Session_SH2_HMon
Source Ad... S.. Source
Start_LEDBli... resetprg.c
Ready
Default1 deskto

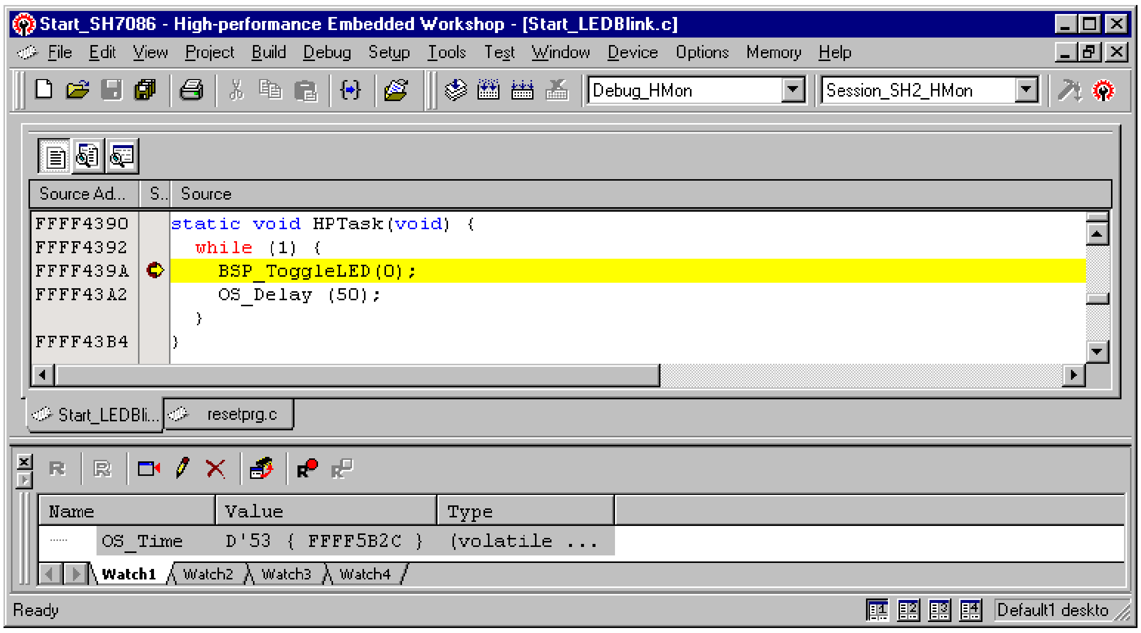
```
FFFF4390   static void HPTask(void) {
FFFF4392     while (1) {
FFFF439A       BSP_ToggleLED(0);
FFFF43A2       OS_Delay (50);
             }
FFFF43B4   }

FFFF43B8   static void LPTask(void) {
FFFF43BA     while (1) {
FFFF43C2       BSP_ToggleLED(1);
FFFF43CA       OS_Delay (200);
             }
FFFF43DE   }
```

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


Start_SH7086 - High-performance Embedded Workshop - [Rtosinit_SH7086.c]
File Edit View Project Build Debug Setup Tools Test Window Device Options Memory Help
Debug_HMon
Session_SH2_HMon
Line Source Ad... S.. Source
resetprg.c Start_LEDBl... Rtosinit_SH7...
Ready
Default1 desktop Read-write 148/351 1

```
             Please note:
             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)
           */
FFFF4290   void OS_Idle(void) {          // Idle loop: No task is ready to exec
             for (;;);                   // Nothing to do ... wait for a interrupt
           }
```

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:


Start_SH7086 - High-performance Embedded Workshop - [Start_LEDBlink.c]
File Edit View Project Build Debug Setup Tools Test Window Device Options Memory Help
Debug_HMon
Session_SH2_HMon
Source Ad... S.. Source
Start_LEDBli...
resetprg.c
Watch1 Watch2 Watch3 Watch4
Ready
Default1 deskto

```
FFFF4390    static void HPTask(void) {
FFFF4392      while (1) {
FFFF439A        BSP_ToggleLED(0);
FFFF43A2        OS_Delay (50);
              }
FFFF43B4    }
```

```
Name        Value              Type
OS_Time     D'53 { FFFF5B2C }  (volatile ...
```

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 50 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 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.

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 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.

4.2. 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 src\.

4.3. 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.

5. HEW compiler specifics

5.1. Memory models, compiler options

embOS for SH2 for HEW and shc compiler is delivered with libraries for the default options and compiler settings.

5.2. Available libraries

embOS is shipped with libraries for SH2 CPUs.

RTOS <CPU> <FPU> <Endianess>_<LibMode>.lib

Parameter	Meaning	Values
CPU	CPU variant	2 : SH2 CPU
FPU	Floating point unit	N: None
Endianess	Type of endianess	B: Big
LibMode	Library mode	XR: Release
		R: Release
		S: Stack check
		SP: Stack check + profiling
		D: Debug
		DP: Debug + profiling
		DT: Debug + profiling + Trace

This results in 7 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 + stack check	OS_LIBMODE_D
DP	Debug + stack check + Profiling	OS_LIBMODE_DP
DT	Debug + stack check + profiling + Trace	OS_LIBMODE_DT

When using HEW workbench, please check the following points:

- The endianess is set as general project option
- 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 preprocessor option for your project. May be defined in `OS_Config.h`.

5.3. Distributed project files

The distribution of ***embOS*** for SH2 and HEW compiler contains a start project for an SH7086 CPU.
The start project contains an ***embOS*** debug and profiling library which should be used during program development.

6. SH2 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 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 / 8`, 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.

The CPU clock generator and PLL itself is initialized during startup in the function `HardwareSetup()` which is implemented in the source file `hwsetup.c`.

6.2. 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.

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.

6.4. *embOS* hardware timer selection

embOS for SH2 CPUs is prepared to use one Compare Match Timer (CMT) 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 “vect.h” and “vecttbl.c” have to be modified accordingly.

6.5. UART for embOSView

Any SCI UART of the SH2 CPU may be used as communication channel for embOSView which enables profiling analysis during runtime.
The initialization code and interrupt handler are delivered in source code and are located in RTOSInit_*.c.
`OS_UART` i may be defined from 0 to 2 to select, initialize and enable one of the SCIs. When embOSView should not be used, define OS_UART to –1. This may be done in `OS_Config.h`.
The UART used for embOSView requires three interrupt handler which are defined in RTOSInit.c:

- **`OS_ISR_err()`** is the reception error interrupt handler.
- **`OS_ISR_rx()`** is the reception interrupt.
- **`OS_ISR_tx()`** is the transmission interrupt which is called on Tx end condition.

The interrupt vector entries in the interrupt vector definition files “vect.h” and “vecttbl.c” have to be set according the UART channel which is used for embOSViev.

7. Stacks

7.1. Task stack for SH2 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 SH2 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 SH2 CPU, the stack size for the CPU registers is 48 bytes.
As the SH2 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.

7.2. System stack for SH2 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.
Because the SH2 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 “stacksct.h” header file.
We recommend at least a minimum of 256 bytes.

7.3. Interrupt stack for SH2 CPUs

The SH2 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 SH2 CPUs

The following chapter describes interrupt specifics of SH2 CPUs and the interrupt modes used with ***embOS***.

8.1. Interrupt processing with SH2 CPUs

SH2 CPUs support a priority controlled interrupt mode. This mode supports the following features:

- Interrupt priority registers to assign 16 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 interrupt is executed.
- The CPU stores PC and the status register onto the current stack.
- The interrupt mask level in the status 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 and status register from the satck.
- For more details, refer to the RENESAS manuals.

8.2. Fast interrupts with SH2 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 8, meaning, any interrupt with level 9 or above is never disabled and can be accepted anytime.
You must not execute any *embOS* function from within a *fast interrupt* function.

8.3. Interrupt priorities with *embOS* for SH2 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()`.

8.4. Defining interrupt handlers for SH2 CPUs in "C"

Routines preceded by the keywords `#pragma 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 `“vect.h”` and the interrupt vector has to be inserted in the vector table in `“vecttbl.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.

```
#pragma interrupt void OS_ISR_Tick(void) {
  OS_CallNestableISR(_IsrTickHandler);
}
```

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_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 handler, 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*** interrupt, 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.

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

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

The interrupt priority limit for fast interrupts is set to 8 by default. This means, all interrupts with higher priority from 9 to 15 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)
```

Parameter	Meaning
`Priority`	The highest value useable as priority for ***embOS*** interrupts. All interrupts with higher priority are never disabled by ***embOS***. Valid range: 1 <= `Priority` <= 15

Return value

NONE.

Add. information

To disable fast interrupts at all, the priority limit may be set to 15 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.

8.6. 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.

9. Non generic, port specific functions

The following chapter describes additional non generic ***embOS*** functions which are available for ***embOS*** SH2 and which are not described in the generic ***embOS*** manual.

9.1. OS_ExtendTaskContext(): Make global variables or processor registers task specific.

Description

OS_ExtendTaskContext() can be used to add global variables or special registers, like floating point registers, to the task context and make them task specific.

Prototype

```
void OS_ExtendTaskContext(
    OS_TASK * pTask,
    void (*pfSave)(void * pStack),
    void (*pfRestore)(const void * pStack)
);
```

Parameter	Meaning
`pTask`	Pointer to the tasks who's task context should be extended. A NULL pointer may be used to address the current running task.
`pfSave`	function pointer, addresses the function used to save the extended task context. This function is called, when the task is suspended for any reason.
`pfRestore`	function pointer, addresses the function used to restore the extended task context. This function is called when the task is resumed.

Return value

NONE.

The function may be used to store global variables like errno or others into the task context, if these variables have to be task specific, which is the case, if they are used by different tasks.

The following listing shows, how to use OS_ExtendTaskContext():

```
--------------------------------------------------------------------
File    : ExtendTaskContext.c
Purpose : Sample program for embOS demonstrating how to dynamically
          extend the task context.
          This example adds a global variable to the task context of
          certain tasks.
--------  END-OF-HEADER  ---------------------------------------------
*/
#include "RTOS.h"

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

/*********************************************************************
*
*       _Restore / _Save
*
*  Function description
*    This function pair saves and restores an extended task context.
*    In this case, the extended task context consists of just a single
*    member, which is a global variable.
*/
typedef struct {
  int GlobalVar;
} CONTEXT_EXTENSION;

void OS_Save(void * pStack) {
  CONTEXT_EXTENSION * p;
  p = ((CONTEXT_EXTENSION*)pStack) - 1;     // Create pointer to our structure
  //
  // Save all members of the structure
  //
  p->GlobalVar = GlobalVar;
}

void OS_Restore(void * pStack) {
  CONTEXT_EXTENSION * p;
  p = ((CONTEXT_EXTENSION*)pStack) - 1;     // Create pointer to our structure
  //
  // Restore all members of the structure
  //
  GlobalVar = p->GlobalVar;
}

/*********************************************************************
*
*       HPTask
*
*  Function description
*    During the execution of this function, the thread-specific
*    global variable has always the same value of 1.
*/
static void HPTask(void) {
  OS_ExtendTaskContext(NULL, OS_Save, OS_Restore);
  GlobalVar = 1;
  while (1) {
    OS_Delay (10);
  }
}

/*********************************************************************
*
*       LPTask
*
*  Function description
*    During the execution of this function, the thread-specific
*    global variable has always the same value of 2.
*/
static void LPTask(void) {
  OS_ExtendTaskContext(NULL, OS_Save, OS_Restore);
  GlobalVar = 2;
  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;
}
```

This sample application can be found in the “Samples” folder of the ***embOS*** distribution.

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 release build library.

Short description	ROM [byte]	RAM [byte]
Kernel	approx.2000	49
Add. Task	---	40
Add. Semaphore	---	16
Add. Mailbox	---	24
Add. Timer	---	20
Power-management	---	---

12. Files shipped with *embOS*

embOS for SH2 and Renesas 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*.hws`	Start workspace for HEW Embedded Workbench.
Start_SH*\	`*.hwp`	CPU specific project file for ***embOS***
Start\Inc\	`RTOS.h`	***embOS*** API header file. To be included in any file using ***embOS*** functions
Start\Lib\	`*.lib`	***embOS*** libraries
Start_SH*\CPU_SH*\	`*.*`	CPU specific sample application and setup files
CPU\	`*.*`	***embOS*** start project sources and files to build libraries and start projects (Source version only)
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