embOS

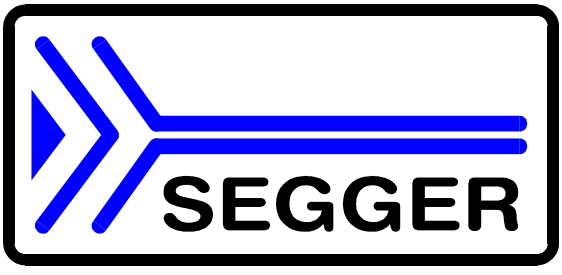
Real Time Operating System

CPU & Compiler specifics for

Renesas V850E/ES/E2 CPUs

and Green Hills compiler

Document Rev. 5



A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Contents

1. About this document

This guide describes how to use ***embOS*** V850E Real Time Operating System for the Renesas V850E, V850ES and V850E2 series of microcontroller using Green Hills compiler for V850 and MULTI Embedded Workbench

1.1. How to use this manual

This manual describes all CPU and compiler specifics for ***embOS*** V850E for GHS 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 GHS MULTI workbench. If you have no experience using ***embOS***, you should follow this introduction, even if you do not plan to use the MULTI 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 V850E using GHS compiler.

2. Using *embOS* with MULTI

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 MULTI to develop your application, no further installation steps are required. You will find prepared sample start projects for different V850E 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 MULTI for your application development in order to become familiar with ***embOS.***

If for some reason you will not work with MULTI, you should:
Copy either all or only the library-file that you need to your work-directory. Also copy the entire CPU specific subdirectory and the ***embOS*** header file RTOS.h. This has the advantage that when you switch to an updated version of ***embOS*** later in a project, you do not affect older projects that use ***embOS*** also.
embOS does in no way rely on MULTI, 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 ready to go sample start projects for different V850E and V850E2 CPUs and it is a good idea to use one of those as a starting point for all of your applications.

Your ***embOS*** distribution contains one folder 'Start' which contains the projects and libraries for different V850E and V850E2 CPUs in CPU specific subfolders in "Start\BoardSupport".

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\BoardSupport' in your work directory.
- Select and open one start project in a CPU specific subfolder.
- Build the start project

You may then step through the project using your simulator or emulator.

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.

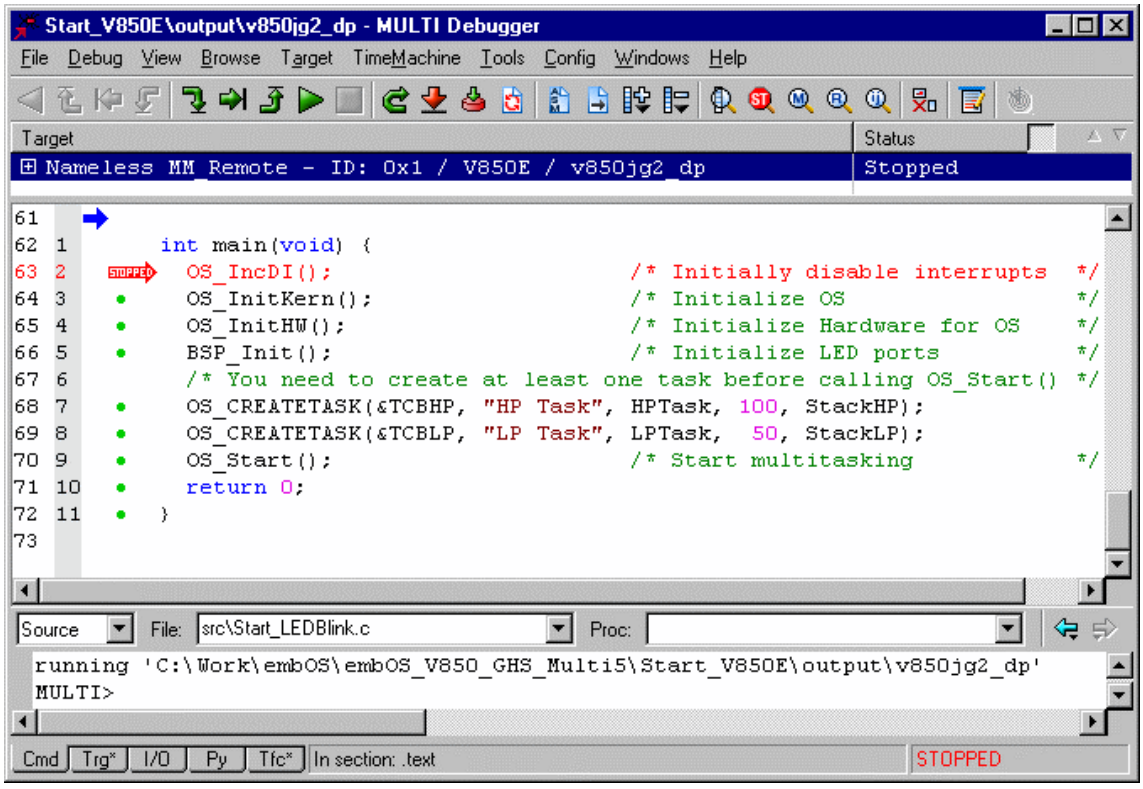
What happens is easy to see:
After initialization of ***embOS***, two tasks are created and started.
The two tasks are activated and execute until they run into the delay, then suspend for the specified time and continue execution.

```
/*********************************************************************
*                SEGGER MICROCONTROLLER GmbH & Co.KG                 *
*       Solutions for real time microcontroller applications         *
**********************************************************************

----------------------------------------------------------------------
File    : Start_2Tasks.c
Purpose : Skeleton program for OS
--------  END-OF-HEADER  ---------------------------------------------
*/

#include "RTOS.h"

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

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

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

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

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

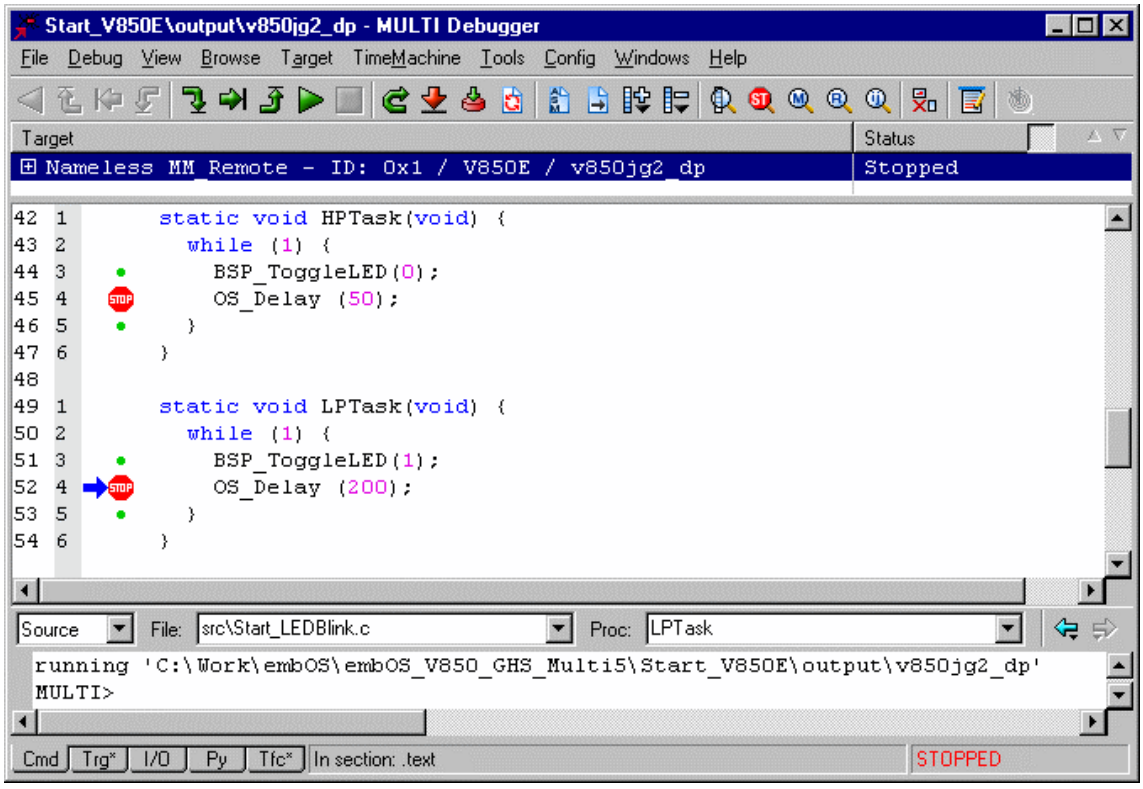
2.4. Stepping through the sample application using an emulator

When starting the simulator or emulator, 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.

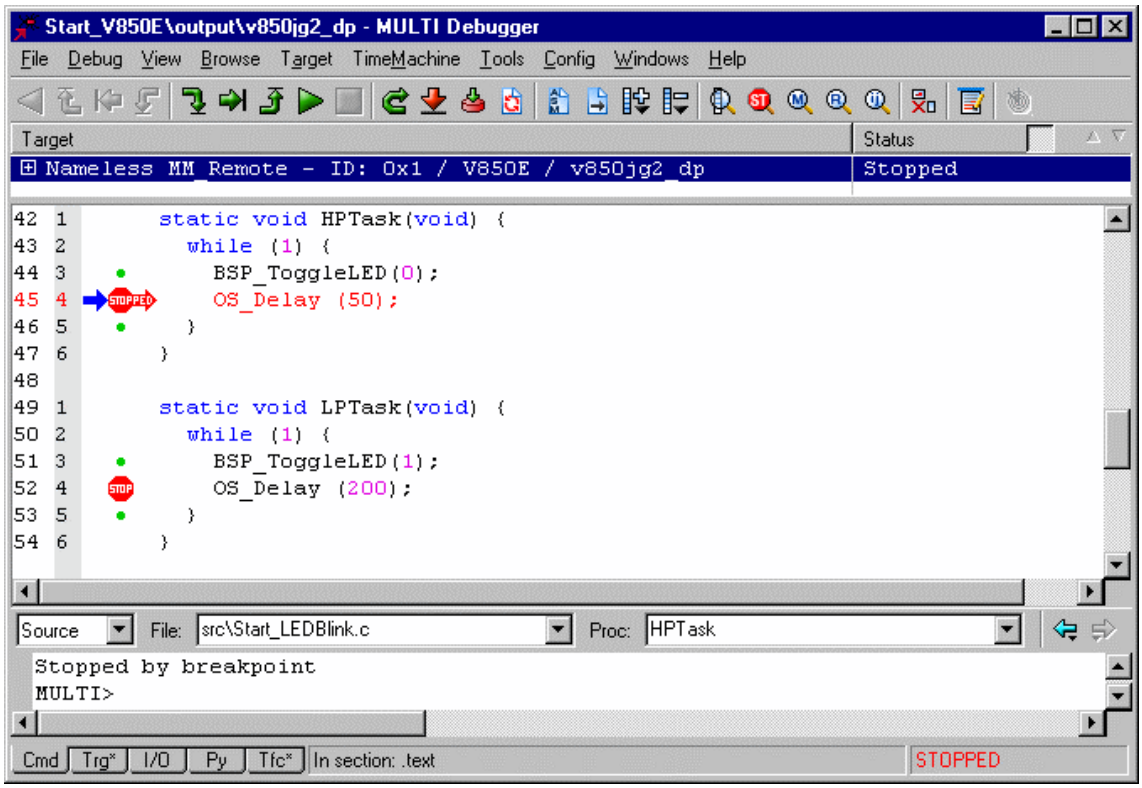
`OS_InitHW()` is part of RTOSINIT.c and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the timer-tick-interrupt for ***embOS.*** Step through it to see what is done.
`OS_COM_Init()` in `OS_InitHW()` is optional. It is required if embOSView shall be used. As simulators usually can not simulate UART operations, OS_UART should be defined as (-1) to disable UART initialization and communication when using a simulation target.
`OS_Start()` should be the last line in main, since it starts multitasking and does not return.



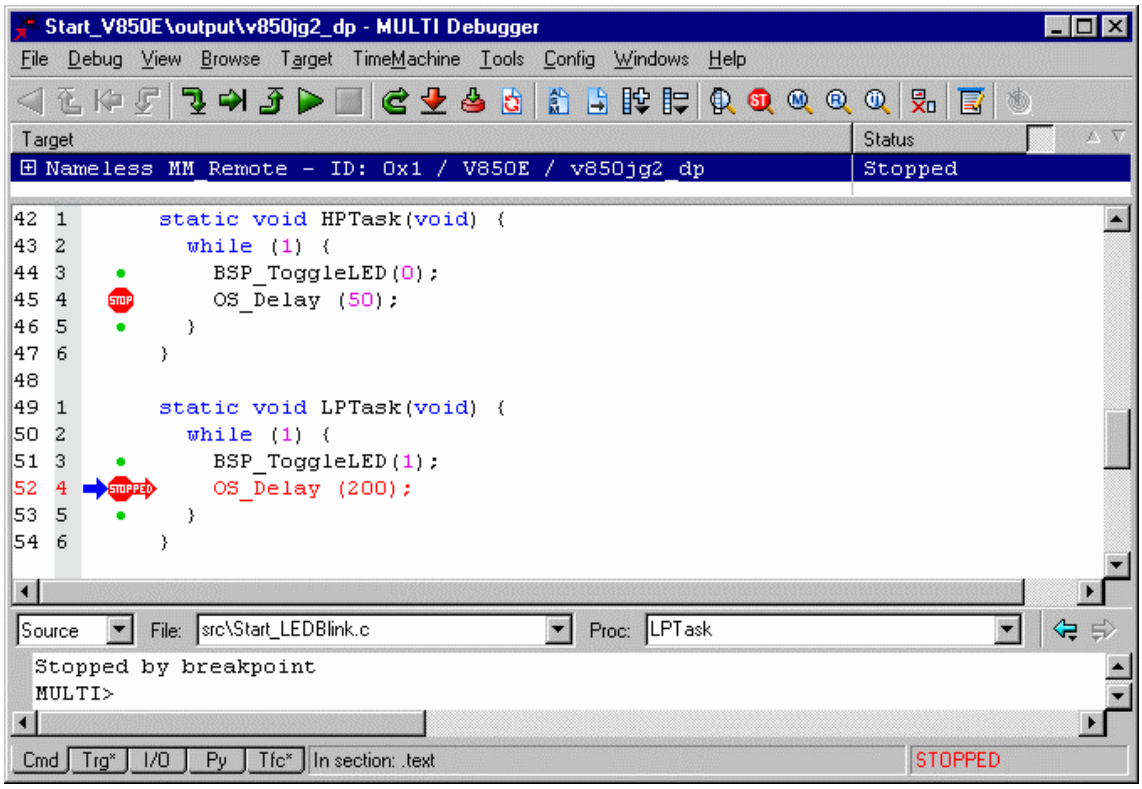
Before you step into `OS_Start()`, you should set breakpoints into the two tasks.



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

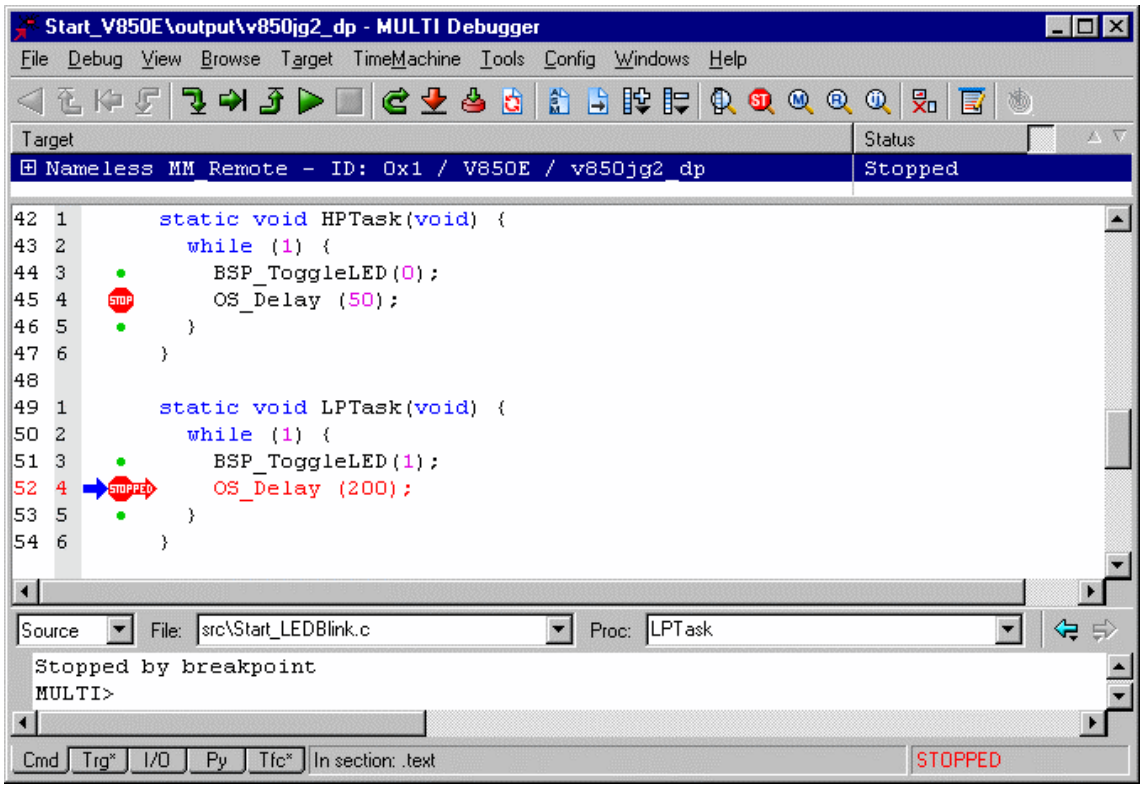


If you continue stepping, you will arrive in the task with the lower priority:



Continuing to step through the program, there is no other task ready for execution. ***embOS*** will suspend LPTask and switch to the idle-loop, which is an endless loop which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

`OS_Idle()` is found in RTOSInit*.c:



If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay.
Coming from `OS_Idle()`, you should execute the 'Go' command to arrive at the highest priority task after its delay is expired.
This will not work in the simulator, because timer interrupts are not simulated.

3. 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.

3.1. 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 specific Setup subfolder.
  It contains the hardware dependent initialization code for the ***embOS*** timer and optional code for a UART for communication with embOSiew.
- **OS_Error.c** from the subfolder Setup
  The `OS_Error()` function is called when any error during runtime is detected by the stack check or the debug library. When using an emulator, it may be helpful to set a breakpoint at `OS_Error()`. Therefore it is delivered as source code.
- One ***embOS*** **library** from the Lib\ subfolder
- Additional CPU specific files in the CPU specific Setup subfolder may be required depending on the V850 CPU variant.

When you decide to write your own startup code, please ensure that non initialized variables are initialized with zero, according to “C” standard, as this is required by ***embOS***.
Your `main()` function has to initialize ***embOS*** by calling `OS_InitKern()` and `OS_InitHW()` prior any other ***embOS*** functions except `OS_IncDI()` are called.

3.2. Select a start project

embOS comes with different start projects for different Renesas V850E or V850E2 CPU derivates.
For your own application, select a start project that (mostly) fits to your CPU.

3.3. Add your own code

For your own code, you may add a new group to the project.
You should then modify or replace the sample main source file in the subfolder “Application”.

3.4. Change memory model or library mode

If you have to select another memory model or want to use another type of ***embOS*** library which is not used in the selected start project, you have to replace the ***embOS*** library in your project:

- Add the appropriate library from the Lib-subdirectory to your project.
- Remove the previous library from your project (or comment it out).

Finally check the project options about target CPU, memory model settings and compiler settings according the library mode used. Refer to chapter 4 about the library naming conventions to select the correct library.

3.5. Modifications for a CPU which is not supported

If your CPU is not supported by the current version of ***embOS***, you have to check and modify the hardware dependent functions found in RTOSInit_*.c.

Check all RTOSInit_*.c files found in CPU specific subfolders CPU_*\Setup to find out which one is closest to your unsupported CPU.
You should not modify the files delivered with ***embOS***.
Make a copy of the CPU specific folder which contains the selected RTOSInit_*.c and rename it according to your CPU derivate.
Then check and modify the following entries in your new RTOSInit_*.c

- Modify the special function register #include according to your CPU.

Normally, the sfr definition file is delivered by GHS. If there is no special file for your CPU available, check whether you may use any file available. Check the addresses of sfrs used in RTOSInit_*.c.

- Check and modify the timer init function `OS_InitHW()`
- Check and modify the time measurement function `OS_GetTime_Cycles()`
- Check the interrupt vector related to `OS_ISR_Tick()`

```
__interrupt void OS_ISR_Tick(void);
#pragma intvect OS_ISR_Tick 0x280

__interrupt void OS_ISR_Tick(void) {
  OS_EnterNestableInterrupt();
  OS_EnterIntStack();
  OS_TICK_Handle();
  OS_LeaveIntStack();
  OS_LeaveNestableInterrupt();
}
```

When embOSView should be used, a UART has to be initialized and handled in RTOSInit_*.c.

- Check and modify the UART init function `OS_COM_Init()`
- Check and modify the transmit function `OS_COM_Send1()`
- Check and modify the transmit interrupt handler function `OS_ISR_tx()` and its related interrupt vector.
- Check and modify the receive interrupt handler function `OS_ISR_rx()` and its related interrupt vector.

4. V850E and V850E2 specifics

4.1. Memory models

embOS supports the default memory and code model combination that Green Hills C-Compiler supports.

4.2. Available libraries

embOS for V850 for GHS compiler is shipped with 7 libraries for V850E cores and 7 libraries for V850E2 cores, one for each *embOS* library type.
The libraries are named as follows:

rtos <CPU> <LIBMODE>.a

Parameter	Meaning	Values
CPU	Specifies the CPU variant	E: V850E E2: V850E2
LIBMODE	Library mode	XR: Extreme Release
		R: Release
		S: Stack check
		SP: Stack check + profiling
		D: Debug + stack check
		DP: Debug + stack check + profiling
		DT: Debug + stack check + profiling + Trace

Example:

rtosESP.a is the library for a project using a V850**E** core, **S**tack check and **P**rofiling library type.
Depending on the library type, you have to set the appropriate compiler setting (define) for your project:

CPU core	Library type	Library	define
V850E	Extreme release	rtosEXR.a	OS_LIBMODE_XR
V850E	Release	rtosER.a	OS_LIBMODE_R
V850E	Stack-check	rtosES.a	OS_LIBMODE_S
V850E	Stack-check + Profiling	rtosESP.a	OS_LIBMODE_SP
V850E	Debug	rtosED.a	OS_LIBMODE_D
V850E	Debug + Profiling	rtosEDP.a	OS_LIBMODE_DP
V850E	Debug + Profiling + Trace	rtosEDT.a	OS_LIBMODE_DT
V850E2	Extreme release	rtosE2XR.a	OS_LIBMODE_XR
V850E2	Release	rtosE2R.a	OS_LIBMODE_R
V850E2	Stack-check	rtosE2S.a	OS_LIBMODE_S
V850E2	Stack-check + Profiling	rtosE2SP.a	OS_LIBMODE_SP
V850E2	Debug	rtosE2D.a	OS_LIBMODE_D
V850E2	Debug + Profiling	rtosE2DP.a	OS_LIBMODE_DP
V850E2	Debug + Profiling + Trace	rtosE2DT.a	OS_LIBMODE_DT

When using MULTI, please check the following points:

- Setup the CPU variant, memory and code model as general project options
- One *embOS* library is part of your project (included)
- The library type definition (OS_LIBMODE_*) is set as compiler option

4.3. V850E core specifics

4.3.1. Location of embOS variables

The V850E cores allow efficient addressing of RAM using short addresses in the zero data area, which is RAM or ROM around address 0.
All *embOS* variables are located in the zero data area. Therefore, the zero data area has to be defined in the linker script file.

4.4. V850E2 core specifics

4.4.1. Location of embOS variables

The V850E2 CPUs may have a different memory layout and do not allow short addressing around address zero, because no RAM is available there.
Therefore, the *embOS* variables are not located in a special memory region, the zero data area is not supported.
The *embOS* start projects for E2 cores are already setup not to use the zero data area.
We recommend to use an already prepared embOS start project as a starting point for your application, as all required settings are already done.
When you build your own project from scratch, ensure to disable the zero data area by option –zda=none, set the project options
Target -> Memory Models -> Special Data Area, Zero Data Area = none.

4.4.2. Interrupt vector table and handler functions

As described in the “Interrupts” section later on in this manual, the Green Hills toolchain allows definition of interrupt vector table entries in the C-source by a special `#pragma` which defines the interrupt vector number for a function and automatically adds the entry in the interrupt vector table.
The start projects for the V850E2 core are based on startup and header files which were generated by a RENESAS tool.
The startup files already include a complete CPU specific interrupt vector table.
The interrupt vectors are enabled by a definition in a separate CPU specific *_irq.h header file.
Therefore, the interrupt vectors must not be defined as `#pragma` in the C-source code.
To enable a peripheral interrupt, uncomment the
`define xxx_ENABLE 0x0000nnnn`
in the *_irq.h file, according the peripheral interrupt handler that should be inserted in the vector table.
The interrupt handler function itself may be implemented in any C-source file.
The function has to be declared as `__interrupt`. The function name should be the same as the predefined referenced name in the startup code.

5. Stacks

5.1. Task stack for V850

Every *embOS* task has to have its own stack. Task stacks are normally declared as arrays of integers and can be located in any RAM memory location.
The stack-size required is the sum of the stack-size of all routines called during the task execution plus a basic stack size used for the task context.
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, plus the stack size required for one interrupt service routine entry.
For the V850E CPUs, this minimum stack size is about 52 bytes to store the permanent CPU registers. A practical minimum value is about 256 bytes, because function calls and at least one interrupt will also run on the task stack.

5.2. System stack for V850

The system stack size required by *embOS* is about 40 bytes. 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 and interrupts also use the system-stack, the actual stack requirements depend on the application.
Interrupt stack switching of *embOS* also uses the system stack for interrupts.
The size of the system stack is given in the link-file as size of the .stack section.

5.3. Interrupt stack for V850

V850 CPUs do not support a separate hardware interrupt stack. Therefore every interrupt runs on the stack which is used when the interrupt occurs. This may be the system stack, or a task stack. To reduce the amount of task-stack needed for interrupts, *embOS* for V850 delivers functions for stack switching in an interrupt service routine.
The functions `OS_EnterIntStack()` and `OS_LeaveIntStack()` may be used in an interrupt service routine to reduce the task stack load.

5.4. Stack specifics of the Renesas V850 family

The Renesas V850 family of microcontroller can address the whole memory space as stack. Therefore, stacks can be located anywhere in RAM. For performance reasons you should try to locate stacks in fast RAM.

6. Interrupts

6.1. What happens when an interrupt occurs?

- The CPU-core receives an interrupt request
- As soon as interrupts are enabled and the processors interrupt priority level is below the interrupt priority level of the pending interrupt, the interrupt is executed
- the CPU saves the PC into the EIPC register
- the CPU saves the current processor status into the EIPSW register
- An exception is written into ECR
- Further interrupts are disabled, EP bit is cleared
- 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 RETI command, restoring saved processor status word and saved PC thus continuing the interrupted task.

6.2. Defining interrupt handlers in "C"

Routines defined with the keyword `__interrupt` automatically save & restore the temporary CPU registers and all additional registers they modify.
The interrupt function returns with the RETI command.
The corresponding interrupt vector number may be defined by a `#pragma` directive prior the interrupt service routine. The toolchain will automatically insert the interrupt vector into the vector table.
When the interrupt vector table is included in the project as separate source file, the interrupt function has to be inserted in the vector table manually, or if already declared in the vector table, the interrupt function has to implemented and named in any source file according the name in the interrupt vector table.

For a detailed description on how to define an interrupt routine in "C", refer to the IAR C-Compiler's user's guide.

"Simple" interrupt-routine:

```
__interrupt void ISR_Timer(void);
#pragma intvect ISR_Timer 0x280

__interrupt void ISR_Timer(void) {
  HandleTimer();
}
```

6.3. Interrupt handler using *embOS* functions

Every interrupt service routine which uses ***embOS*** functions has to inform ***embOS*** that interrupt code is running. Therefore the first command in an interrupt service routine should be `OS_EnterInterrupt()`, the last command has to be `OS_LeaveInterrupt()`.
If interrupts should be re-enabled in an interrupt service routine, thus allowing nested interrupts, use `OS_EnterNestableInterrupt()` and `OS_LeaveNestableInterrupt()`

Interrupt-routine using ***embOS*** functions:

```
#pragma intvect OS_ISR_Tick 0x280
__interrupt void OS_ISR_Tick(void) {
  OS_EnterNestableInterrupt();  // Enable nested interrupts
  OS_TICK_Handle();
  OS_LeaveNestableInterrupt();
}
```

6.4. Interrupt stack switching with *embOS*

Since the V850 CPUs do not have a separate stack pointer for interrupts, every interrupt runs on the current stack. To reduce stack load of tasks, ***embOS*** offers its own interrupt stack which is located in the system stack.
To use the ***embOS*** interrupt stack, call `OS_EnterIntStack()` at the beginning of an interrupt handler just after the call of `OS_EnterInterrupt()` or `OS_EnternestableInterrupt()` and `OS_LeaveIntStack()` at the end just before `OS_LeaveInterrupt()` or `OS_LeaveNestableInterrupt()`.

Please note, that an interrupt handler using interrupt stack switching MUST NOT use local variables and should not perform any calculations. An interrupt handler using interrupt stack switching should call a function which handles the interrupt.

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

```
#pragma vector = 0x360
__interrupt void OS_ISR_rx(void) {
  OS_EnterInterrupt();     /* Inform embOS that ISR is running         */
  OS_EnterIntStack();      /* We will use interrupt stack              */
  _ISR_RxHandler();        /* Call to handler is required !            */
  OS_LeaveIntStack();      /* Interrupt stack switching does           */
  OS_LeaveInterrupt();     /* not allow local variables in ISR         */
}
```

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

```
static void _ISR_RxHandler(void) {
  if (ASIS1 & 0x07) {           /* Check any reception error        */
    Dummy = RXBL1;              /* Reset error, discard Byte        */
  } else {
    OS_OnRx(RXBL1);             /* Process data                     */
  }
}

#pragma vector = 0x360
__interrupt void OS_ISR_rx(void) {
  OS_EnterNestableInterrupt();  /* We will enable interrupts        */
  OS_EnterIntStack();           /* We will use interrupt stack      */
  _ISR_RxHandler();             /* Call to handler is required !    */
  OS_LeaveIntStack();           /* Interrupt stack switching does   */
  OS_LeaveNestableInterrupt();  /* not allow local variables in ISR */
}
```

7. HALT / IDLE / STOP Mode

Usage of the HALT mode 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.
As the internal peripheral clock is not stopped in this mode, ***embOS*** keeps functioning. Any interrupt will wake up the CPU and will therefore continue suspended tasks if required.

IDLE and STOP mode stop internal peripheral clock and can only be resumed by NMI or RESET and should therefore not be used to reduce power consumption during idle times in `OS_Idle()`

8. Technical data

8.1. 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 tiny memory model, short address mode and release build library.

Short description	ROM [byte]	RAM [byte]
Kernel	approx.1870	40
Add. Task	---	32
Add. Counting Semaphore	---	8
Add. Mailbox	---	24
Add. Timer	---	20
Power-management	---	---

9. Files shipped with *embOS* for GHS V850 compiler

Directory	File	Explanation
root	`*.pdf`	Generic API- and target specific documentation
root	`Release*.html`	Release notes of ***embOS*** V850
root	`embOSView.exe`	Utility for runtime analysis, described in generic documentation
Start\Inc	`RTOS.h`	The API header file, to be included into any source file which uses embOS functions.
Start\Lib	`rtos*.a`	The embOS libraries
Start\BoardSupport\CPU_*\	`*.gpj`	CPU specific start projects

10. Index