embOS

Real Time Operating System

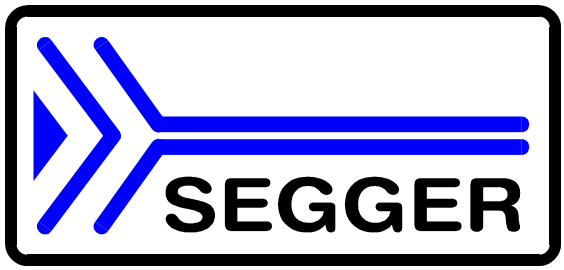
CPU & Compiler specifics for

Renesas

V850 / V850E/ES/E2/E2M CPUs

and IAR compiler for V850

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*** V850 Real Time Operating System for the Renesas V850 series of microcontroller using IAR compiler for V850 and IAR's Embedded Workbench 4.x

1.1. How to use this manual

This manual describes all CPU and compiler specifics for ***embOS*** V850 for IAR compiler. Before actually using ***embOS***, you should read or at least glance through this manual in order to become familiar with the software.
Chapter 2 gives you a step-by-step introduction, how to install and use ***embOS*** using IAR workbench. If you have no experience using ***embOS***, you should follow this introduction, even if you do not plan to use C-SPY or IAR's Embedded Workbench, because it is the easiest way to learn how to use ***embOS*** in your application.
Most of the other chapters in this document are intended to provide you with detailed information about the functionality and fine-tuning of ***embOS*** for V850 using the IAR compiler and IARs Embedded Workbench.

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

2.1. Installation

embOS is shipped on CD-ROM or as a zip-file in electronic form.

In order to install it, proceed as follows:

If you received a CD, copy the entire contents to your hard-drive into any folder of your choice. When copying, please keep all files in their respective sub directories. Make sure the files are not read only after copying.
If you received a zip-file, please extract it to any folder of your choice, preserving the directory structure of the zip-file.

Assuming that you are using IAR's Embedded Workbench to develop your application, no further installation steps are required. You will several sample workspaces and prepared sample start projects for different V850 CPUs, which you should use and modify to write your application. So follow the instructions of the next chapter 'First steps'.

You should do this even if you do not intend to use IAR's Embedded Workbench for your application development in order to become familiar with ***embOS.***

If for some reason you will not work with IAR's Embedded Workbench, 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 IAR™s Embedded Workbench, it may be used without the workbench using batch files or a make utility without any problem.

2.2. First steps

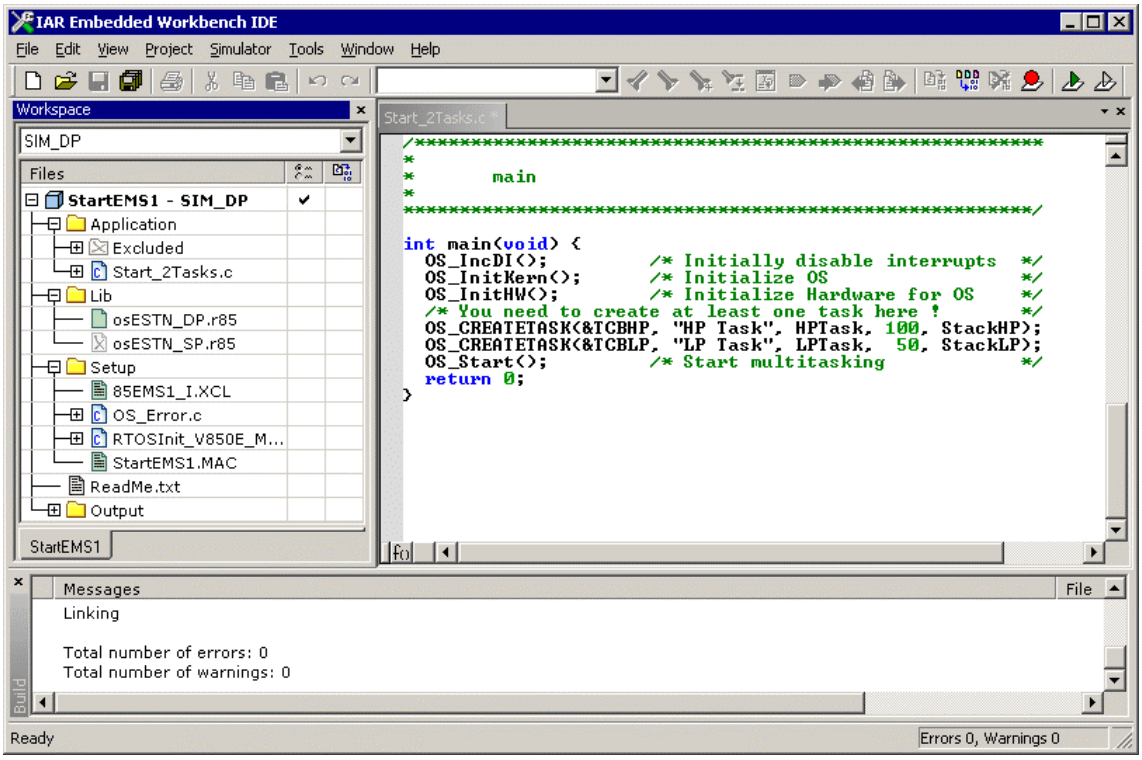
After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received several ready to go sample start projects for different V850 CPUs and it is a good idea to use one of those as a starting point of all of your applications.

Your ***embOS*** distribution contains the folder ‘Start\Boardsupport’ which contains several CPU specific subfolders. Each CPU specific subfolder contains a sample start project readily setup for the specific CPU. Sample applications are located in the ‘Application’ subfolders and every additional files needed are located in the ‘setup’ folder.
To start, select one CPU specific start project.
Every project also contains a configuration setup for the IAR CSpy simulator, so there is no real hardware required to strt.

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 one project workspace from the BoardSupport\CPU_*\ subfolder”.
- Select a configuration, for example SIM_DP, which is uilt for the CSpy simulator.
- Build the start project

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



2.3. The sample application Start2Tasks.c

The following is a printout of the sample application Start2Tasks.c. It is a good starting-point for your application.

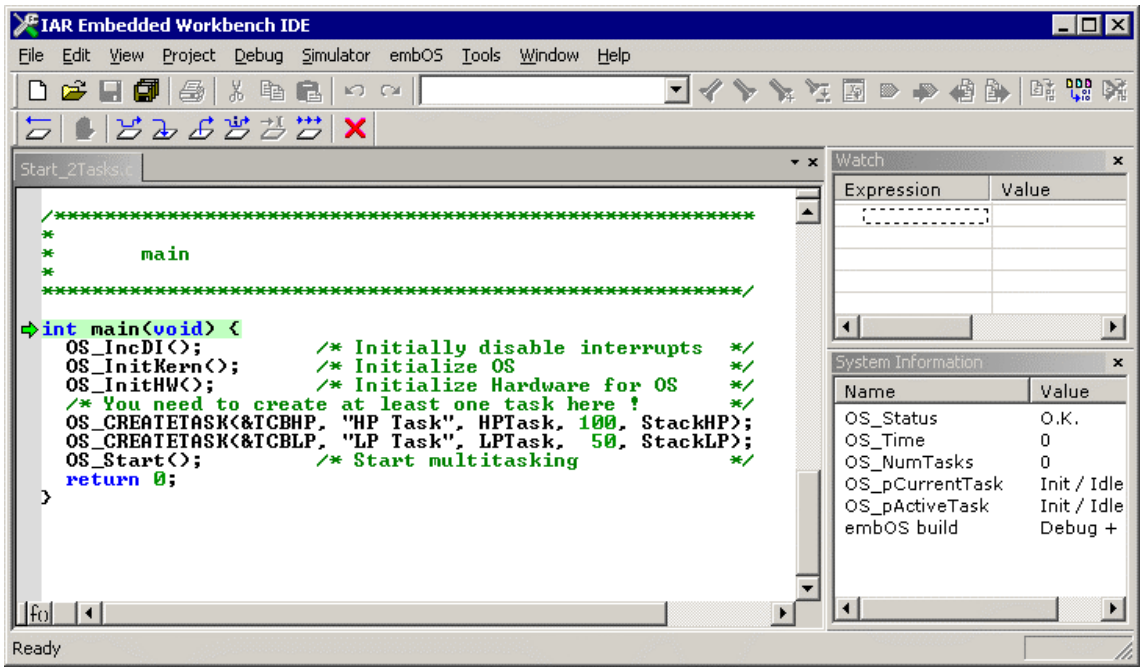
What happens is easy to see:
After initialization of ***embOS,*** two tasks are created and started
The two tasks are activated and execute until they run into the delay, then suspend for the specified time and continue execution.

```
/*********************************************************************
*                 SEGGER MICROCONTROLLER GmbH & Co KG
*        Solutions for real time microcontroller applications
**********************************************************************
----------------------------------------------------------------------
File    : Start_2Tasks.c
Purpose : Skeleton program for embOS
--------  END-OF-HEADER  ---------------------------------------------
*/

#include "RTOS.h"

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

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

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

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

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

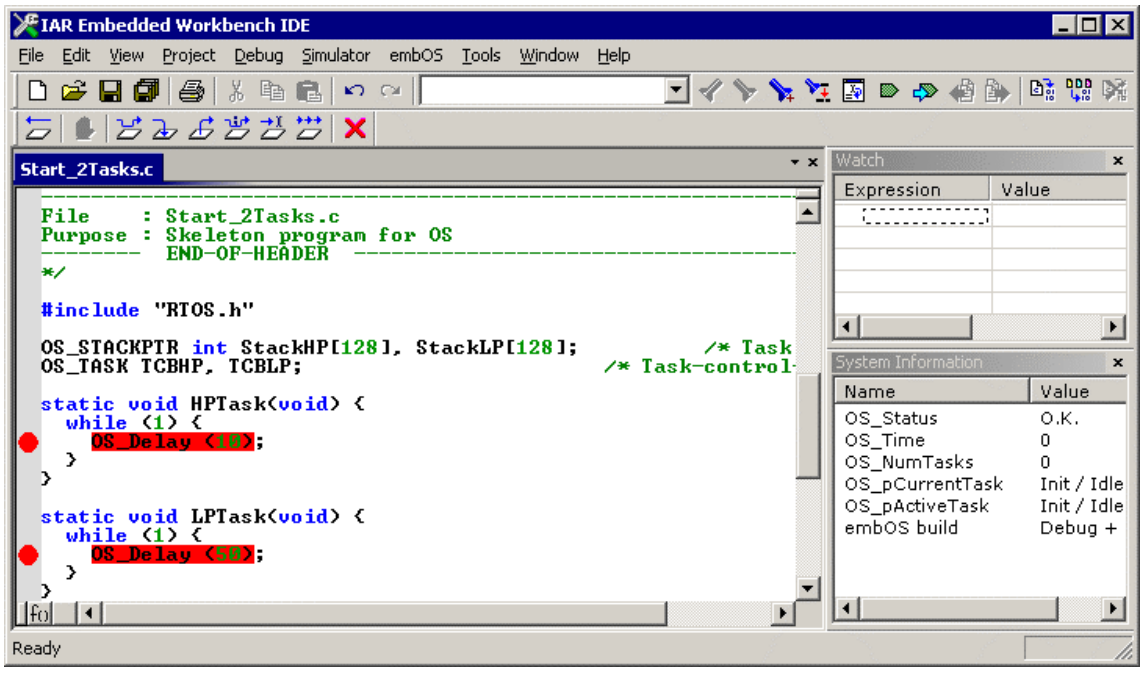
2.4. Stepping through the sample application Main.c using CSpy

When starting the CSpy simulator or emulator after building the project, 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.

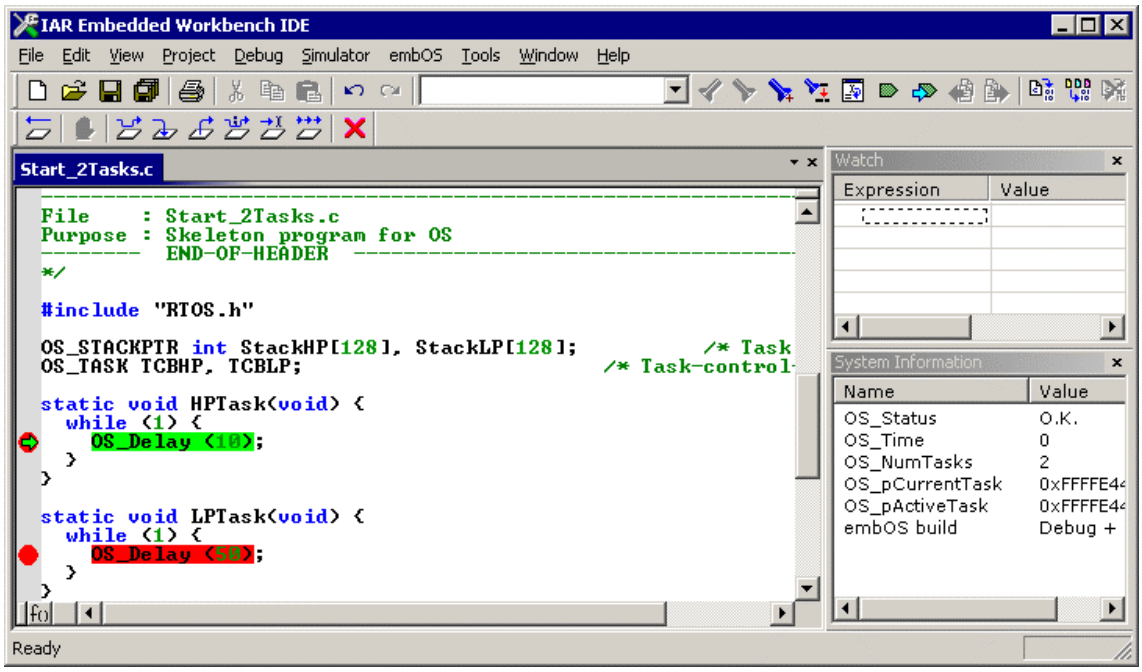
`OS_InitHW()` is part of RTOSINIT.c and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the timer-tick-interrupt for ***embOS.*** Step through it to see what is done.
`OS_COM_Init()` in `OS_InitHW()` is optional. It is required if embOSView should be used. As simulators usually can not simulate UART operations, OS_UART may 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.



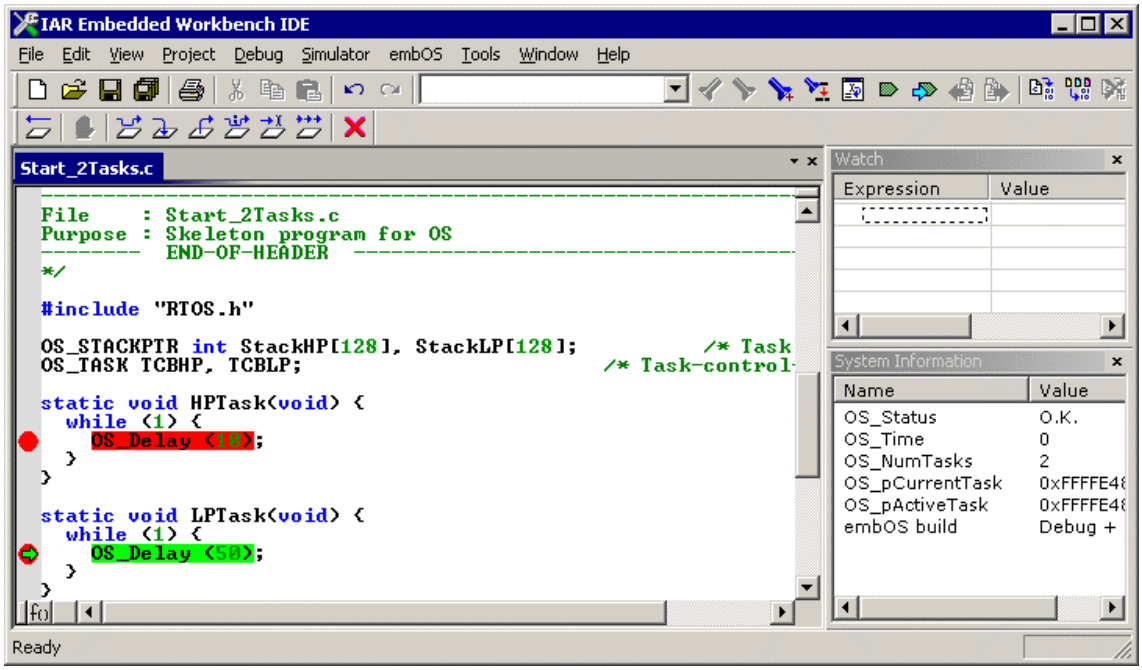
Before you step into `OS_Start()`, you should set breakpoints in the two tasks:



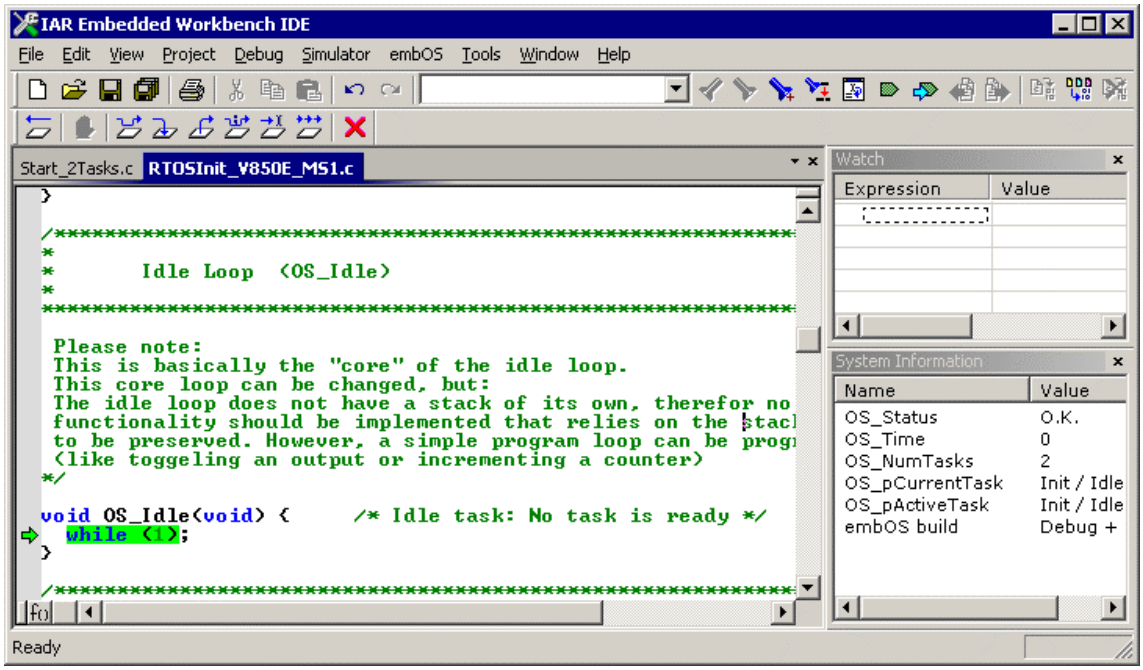
When you step over `OS_Start()`, the next line executed is already in the highest priority task created. (you may also step into `OS_Start()`, then stepping through the task switching process in disassembly mode). In our small start program, `HPTask()` is the highest priority task and is therefore active.



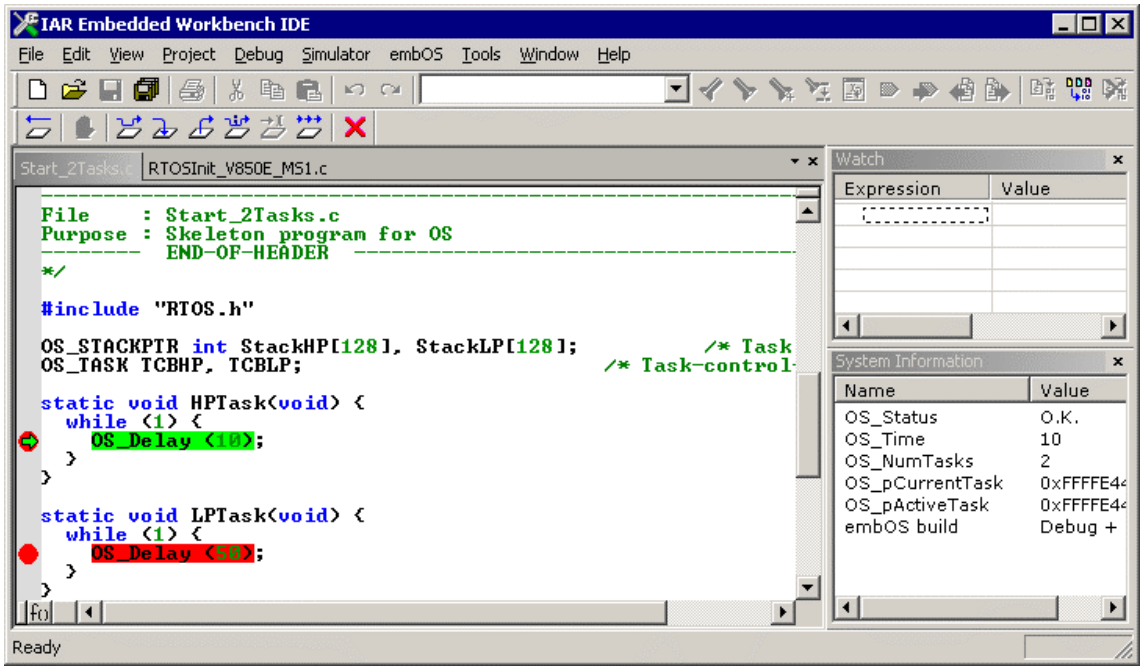
If you continue stepping, you will arrive in the task with the lower priority:



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.



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 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.
- **RTOSInit_*.c**, the CPU specific initialization, from one CPU specific subfolder Start\Boardsupport\CPU_*\Setup.
  It contains the hardware dependent initialization code for the ***embOS*** timer and optional functions for a UART to communicate with embOSiew.
- **OS_Error.c** from a CPU specific 'Start\BoardSupport\CPU_*\Setup\' folder.
  The `OS_Error()` function is called when any error during runtime is detected by the stack check or 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 'Start\Lib\' subfolder
- Additional CPU specific files in the 'Start\BoardSupport\CPU_*\Setup\' folder 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 call of `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 V850 CPU derivates.
For your own application, select a start project that (mostly) fits your CPU.

3.3. Add your own code

For your own code, you may add a new group to the project.
You may also modify or replace the sample application in the Application folder.

3.4. Change memory model or library mode

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

- Add the appropriate library from the Lib-subdirectory to the Lib group.
- Disable or remove all other libraries.

Finally check the project options about the target CPU memory and code model settings and compiler settings according the library mode used. Refer to chapter 4 about the library naming conventions to select the correct library and set the appropriate define in the preprocessor settings for your project.

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 one of the CPU specific 'Setup' subfolders 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 project, 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.

```
#include <io_V850_df3017.h>     /* SFR file delivered from IAR      */
```

Normally the sfr definition file is delivered by IAR. If there is no special file for your CPU available, please 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()`

```
#ifdef __ghs                    /* This declaration for Green Hills */
__interrupt void OS_ISR_Tick(void)
  #pragma intvect OS_ISR_Tick 0x280
#else                           /* This for IAR compiler            */
  #pragma vector = 0x280
__interrupt void OS_ISR_Tick(void)
#endif
{
  OS_TickHandler();
}
```

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. V850 / V850E specifics

4.1. Memory models

embOS supports all memory and code model combinations that IAR' s C-Compiler supports.

4.2. Available libraries

embOS for V850 for IAR compiler is shipped with 196 different libraries, one for each CPU / addressing mode / memory model / code model and library type combination. The libraries are named as follows:

OS w x y z_LM.r85

Parameter	Meaning	Values	
W	Specifies the CPU variant	V:	V850
		E:	V850E/V850ES/V850E2M
X	Short address mode	S:	short address
		N:	NO short address
Y	Memory model	T:	Tiny
		S:	Small
		L:	Large
		M:	Medium (V850E2M only)
Z	Code model	N:	Normal
		L:	Large
LM	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:

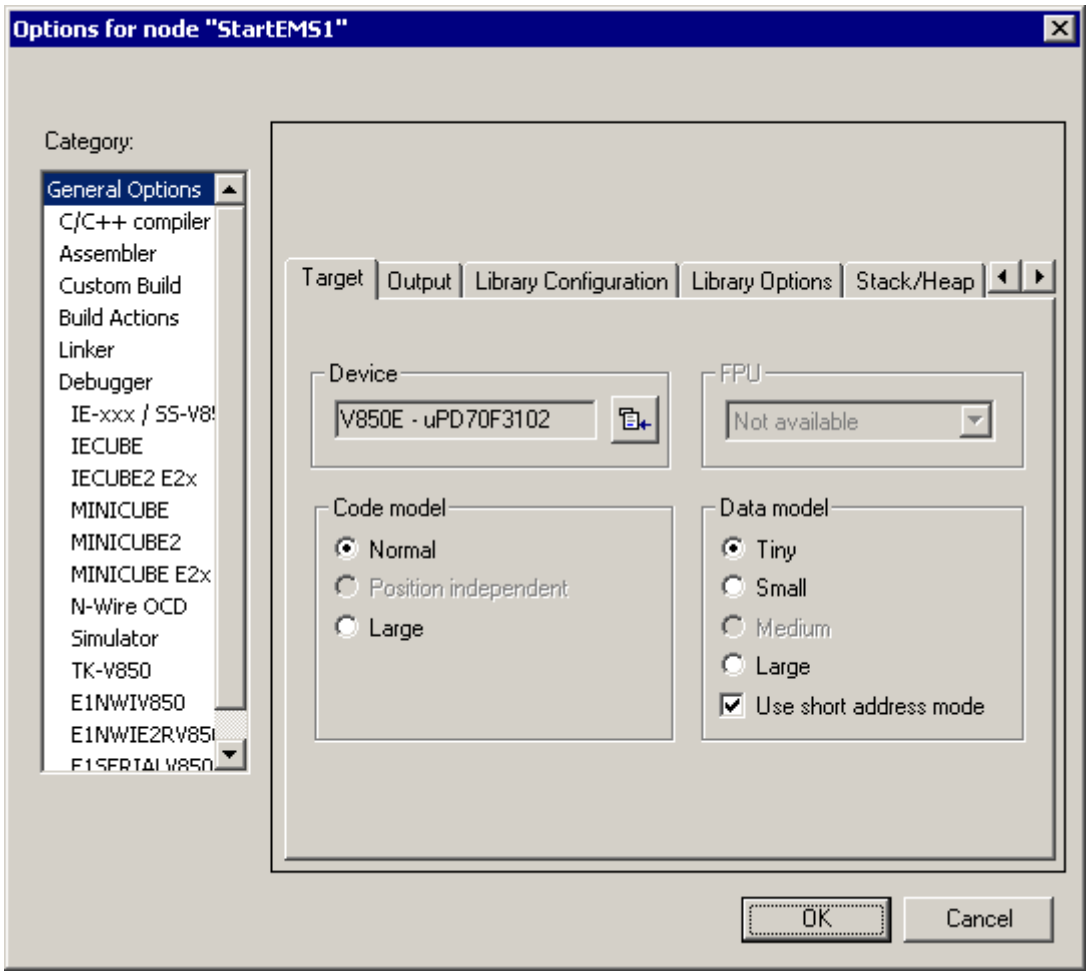
osESTN_R.r85 is the library for a V850E core, short addressing mode, tiny memory model, normal code model and release build library type. Depending on the library type, you have to set the appropriate compiler setting (define) for your project.

embOS library modes and library mode definition:

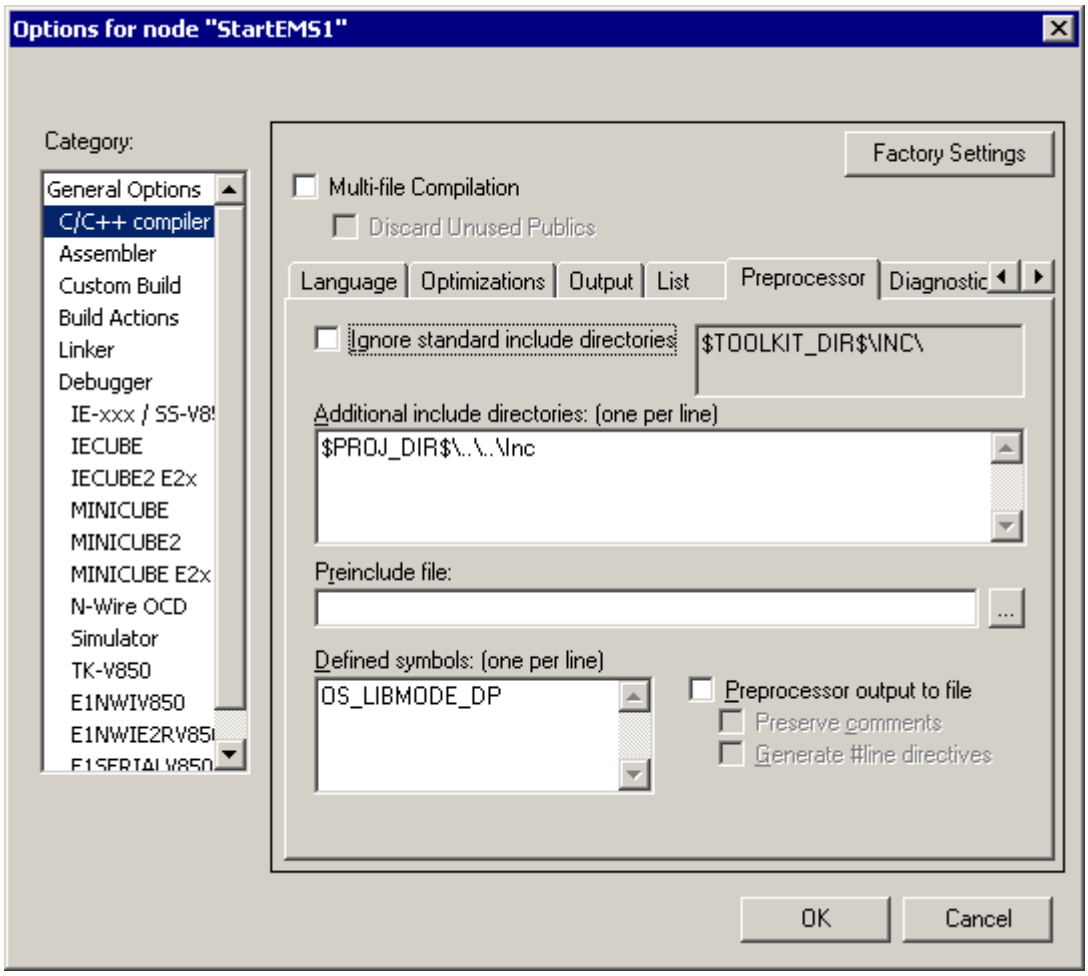
Processor	Library type	Library	define
V850	eXtreme Release	OSV x y z_R	OS_LIBMODE_XR
V850	Release	OSV x y z_R	OS_LIBMODE_R
V850	Stack-check	OSV x y z_S	OS_LIBMODE_S
V850	Stack-check + Profiling	OSV x y z_SP	OS_LIBMODE_SP
V850	Debug	OSV x y z_D	OS_LIBMODE_D
V850	Debug + Profiling	OSV x y z_DP	OS_LIBMODE_DP
V850	Debug + Profiling + Trace	OSV x y z_DT	OS_LIBMODE_DT
V850E	eXtreme Release	OSV x y z_R	OS_LIBMODE_R
V850E	Release	OSV x y z_R	OS_LIBMODE_R
V850E	Stack-check	OSV x y z_S	OS_LIBMODE_S
V850E	Stack-check + Profiling	OSV x y z_SP	OS_LIBMODE_SP
V850E	Debug	OSV x y z_D	OS_LIBMODE_D
V850E	Debug + Profiling	OSV x y z_DP	OS_LIBMODE_DP
V850E	Debug + Profiling + Trace	OSV x y z_DT	OS_LIBMODE_DT
V850E2M	eXtreme Release	OSE x M z_R	OS_LIBMODE_XR
V850E2M	Release	OSE x M z_R	OS_LIBMODE_R
V850E2M	Stack-check	OSE x M z_S	OS_LIBMODE_S
V850E2M	Stack-check + Profiling	OSE x M z_SP	OS_LIBMODE_SP
V850E2M	Debug	OSE x M z_D	OS_LIBMODE_D
V850E2M	Debug + Profiling	OSE x M z_DP	OS_LIBMODE_DP
V850E2M	Debug + Profiling + Trace	OSE x M z_DT	OS_LIBMODE_DT

When using the IAR workbench, please check the following points:

- Setup the CPU variant, memory and code model as general project options



- One ***embOS*** library is included and enabled in the project.
- The library type definition (OS_LIBMODE_*) is set as compiler option



5. Stacks

5.1. Task stack for V850

Every ***embOS*** task has to have its own stack. Task stacks can be located in any RAM memory location.
The stack-size required is the sum of the stack-size of all routines plus a 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 V850 CPUs, this minimum stack size is about 136 bytes to store the CPU registers. A practical minimum value is about 180 bytes

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 of `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 `CSTACK`.

5.3. Interrupt stack for V850

V850 CPUs do not support a separate hardware interrupt stack. Therefore every interrupt runs on the task stack, as long as interrupt functions do not use interrupt stack switching functions.
To reduce task stack load by interrupts, ***embOS*** uses the system stack as interrupt stack. Interrupt handler should use `OS_EnterIntStack()` and `OS_LeaveIntstack()` to switch to the interrupt stack. Please refer to chapter “Interrupts”.

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 current interrupt priority level of the interrupting source, the interrupt is accepted and executed.
- The CPU saves the current PC in the EIPC register.
- The CPU saves the current processor status in the EIPSW register.
- An exception is written into ECR
- Further interrupts are disabled, the EP bit is cleared
- The CPU jumps to the address specified in the vector table for the interrupt service routine (ISR) of the interrupting source.
- ISR : Save registers.
- ISR : User-defined functionality
- ISR : Restore registers
- ISR: Execute the RETI command, restoring the saved processor status word and the saved PC thus continuing the interrupted program.

6.2. Defining interrupt handlers in "C"

Routines defined with the keyword `__interrupt` automatically save & restore the registers they modify and return with RETI.
The corresponding interrupt vector number may be defined by a #pragma directive prior the interrupt service routine.
For a detailed description on how to define an interrupt routine in "C", refer to the IAR Compiler Reference guide.

"Simple" interrupt-routine:

```
#pragma vector = 0x1c0
__interrupt void OS_ISR_tx(void) {
  SendNextChar();
}
```

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

```
#pragma vector = 0x1c0
__interrupt void OS_ISR_tx(void) {
  OS_EnterInterrupt();
  OS_OnTx();
  OS_LeaveInterrupt();
}
```

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()`

6.3. Interrupt stack switching

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 ***embOS*** interrupt stack, call `OS_EnterIntStack()` at the beginning of an interrupt handler just after the call of the ***embOS*** ISR entry function `OS_EnterInterrupt()` or `OS_EnterNestableInterrupt()` and `OS_LeaveIntStack()` at the end just before calling `OS_LeaveNestableInterrupt()` or `OS_LeaveInterrupt()`.

An interrupt handler using interrupt stack switching must not use local variables.
An interrupt handler using interrupt stack switching shall call a function that does the work and handles the interrupt.

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

```
static void OS_ISR_Rx_Handler(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      */
  OS_ISR_Rx_Handler();          /* Call to handler is required !    */
  OS_LeaveIntStack();           /* Interrupt stack switching does   */
  OS_LeaveNestableInterrupt();  /* not allow local variables in ISR */
}
```

Interrupt stack switching is efficient when using multiple nestable interrupts with different priorities, because only the first interruptible interrupt will store some registers onto the current stack, before switching to the embOS interrupt stack.
All additional interrupts with higher priority run on the interrupt stack as long as the interrupt stack is active.

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 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	38
Add. Task	---	32
Add. Semaphore	---	8
Add. Mailbox	---	20
Add. Timer	---	20
Power-management	---	---

9. Files shipped with *embOS* for IAR 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`	To be included in any file using ***embOS*** functions
Start\Lib\	`os*.r85`	***embOS*** libraries
Start\CPU_*\	`Start*.eww`	CPU specific sample workspace
Start\CPU_*\	`Start*.ewp`	CPU specific sample project
Start\CPU_*\	`Start*.eww`	Debugger configuration file for sample project
Start\CPU_*\ Application\	`*.c`	Sample application programs.
Start\CPU_*\ Setup\	`OS_Error.c`	***embOS*** error handler, used in stack check or debug builds
Start\CPU_*\ Setup\	`RTOSInit_*.c`	Target CPU specific init functions. May be modified according to your hardware.
Start\CPU_*\ Setup\	`*.mac`	Target CPU specific simulation macro files for C-SPY simulator.
Start\CPU_*\ Setup\	`*.*`	Target CPU specific linker files and others required for the specific CPU variant

10. Index