embOS

Real-Time Operating System

CPU & Compiler specifics
for Renesas RH850 and IAR

Document: UM01066
Software Version: 5.04
Revision: 0
Date: September 25, 2018

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

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

Copyright notice

You may not extract portions of this manual or modify the PDF file in any way without the prior written permission of SEGGER. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

© 2010-2018 SEGGER Microcontroller GmbH, Monheim am Rhein / Germany

Trademarks

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

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

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel. +49 2173-99312-0
Fax. +49 2173-99312-28
E-mail: support@segger.com
Internet: *www.segger.com*

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please inform us and we will try to assist you as soon as possible. Contact us for further information on topics or functions that are not yet documented.

Print date: September 25, 2018

Software	Revision	Date	By	Description
5.04	0	180925	TS	New software version.
4.24	0	160805	MC	Initial version.

 © 2010-2018 SEGGER Microcontroller GmbH

About this document

Assumptions

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

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

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

How to use this manual

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

Typographic conventions for syntax

This manual uses the following typographic conventions:

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

© 2010-2018 SEGGER Microcontroller GmbH

 © 2010-2018 SEGGER Microcontroller GmbH

Table of contents

1 Using embOS 8

1.1 Installation 9
1.2 First Steps 10
1.3 The example application OS_StartLEDBlink.c 11
1.4 Stepping through the sample application 12

2 Build your own application 16

2.1 Introduction 17
2.2 Required files for an embOS 17
2.3 Change library mode 17
2.4 Select another CPU 17

3 Libraries 18

3.1 Naming conventions for prebuilt libraries 19

4 CPU and compiler specifics 20

4.1 CPU modes 21
4.2 Standard system libraries 22
4.3 Thread-safe system libraries 22
4.4 Thread-Local Storage TLS 23

5 Stacks 25

5.1 Task stack 26
5.2 System stack 26
5.3 Interrupt stack 26

6 Interrupts 27

6.1 What happens when an interrupt occurs? 28
6.2 Defining interrupt handlers in C 28
6.3 Interrupt vector table 28
6.4 Interrupt-stack switching 29
6.5 Interrupt nesting 29

7 Technical data 32

7.1 Memory requirements 33

 © 2010-2018 SEGGER Microcontroller GmbH

Chapter 1

Using embOS

This chapter describes how to start with and use embOS. You should follow these steps to become familiar with embOS.

© 2010-2018 SEGGER Microcontroller GmbH

1.1 Installation

embOS is shipped as a zip-file in electronic form.

To install it, proceed as follows:

Extract the zip-file to any folder of your choice, preserving the directory structure of this file. Keep all files in their respective sub directories. Make sure the files are not read only after copying.

Assuming that you are using an IDE to develop your application, no further installation steps are required. You will find a lot of prepared sample start projects, which you should use and modify to write your application. So follow the instructions of section *First Steps* on page 10.

You should do this even if you do not intend to use the IDE for your application development to become familiar with embOS.

If you do not or do not want to work with the IDE, you should: Copy either all or only the library-file that you need to your work-directory. The advantage is that when switching to an updated version of embOS later in a project, you do not affect older projects that use embOS, too. embOS does in no way rely on an IDE, it may be used without the IDE using batch files or a make utility without any problem.

 © 2010-2018 SEGGER Microcontroller GmbH

1.2 First Steps

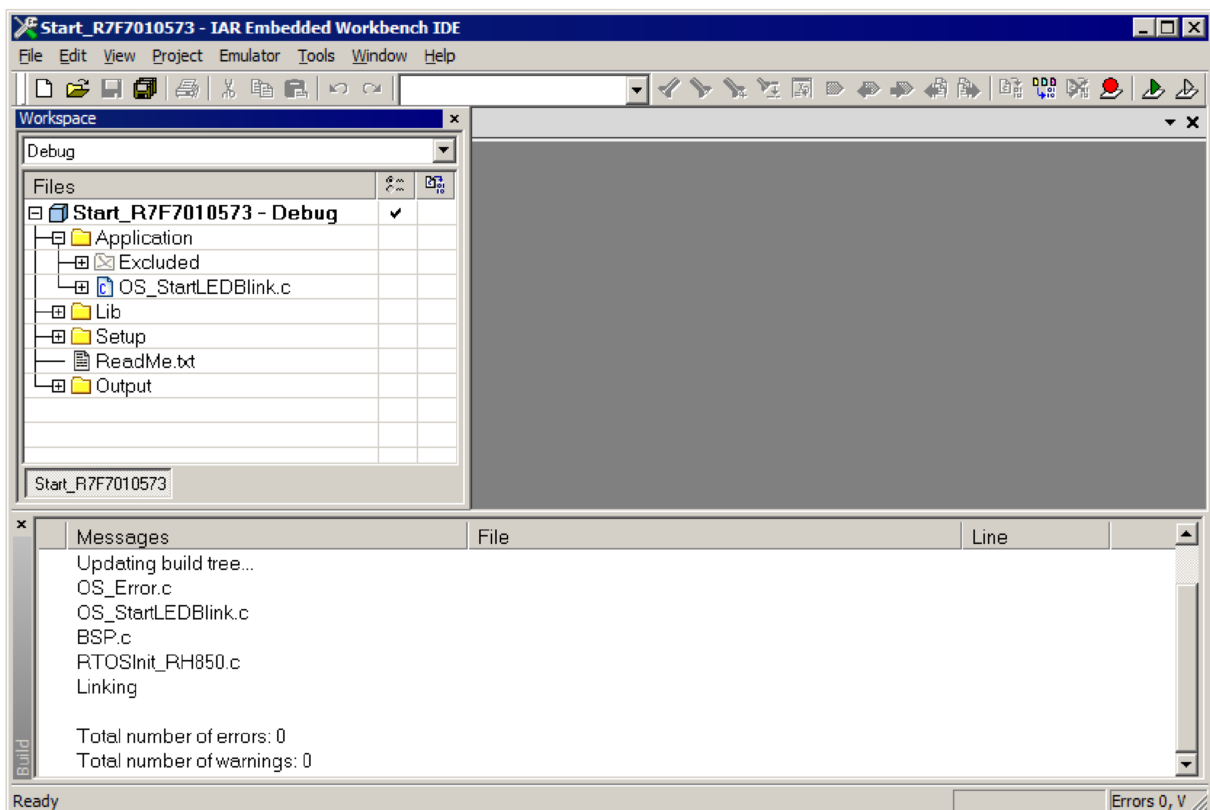
After installation of embOS you can create your first multitasking application. You have received several ready to go sample start workspaces and projects and every other files needed in the subfolder Start. It is a good idea to use one of them as a starting point for all of your applications. The subfolder BoardSupport contains the workspaces and projects which are located in manufacturer- and CPU-specific subfolders.

To start with, you may use any project from BoardSupport subfolder.

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

- Create a work directory for your application, for example c:\work.
- Copy the whole folder Start which is part of your embOS distribution into your work directory.
- Clear the read-only attribute of all files in the new Start folder.
- Open one sample workspace/project in Start\BoardSupport\<DeviceManufacturer>\<CPU> with your IDE (for example, by double clicking it).
- Build the project. It should be built without any error or warning messages.

After generating the project of your choice, the screen should look like this:



For additional information you should open the ReadMe.txt file which is part of every specific project. The ReadMe file describes the different configurations of the project and gives additional information about specific hardware settings of the supported eval boards, if required.

 © 2010-2018 SEGGER Microcontroller GmbH

1.3 The example application OS_StartLEDBlink.c

The following is a printout of the example application `OS_StartLEDBlink.c`. It is a good starting point for your application. (Note that the file actually shipped with your port of embOS may look slightly different from this one.)

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                    *
*                        The Embedded Experts                        *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------
File    : OS_StartLEDBlink.c
Purpose : embOS sample program running two simple tasks, each toggling
          a LED of the target hardware (as configured in BSP.c).
*/

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

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

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

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

/*********************************************************************
*
*       main()
*/
int main(void) {
  OS_Init();    // Initialize embOS
  OS_InitHW();  // Initialize required hardware
  BSP_Init();   // Initialize LED ports
  OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
  OS_TASK_CREATE(&TCBLP, "LP Task",  50, LPTask, StackLP);
  OS_Start();   // Start embOS
  return 0;
}

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

 © 2010-2018 SEGGER Microcontroller GmbH

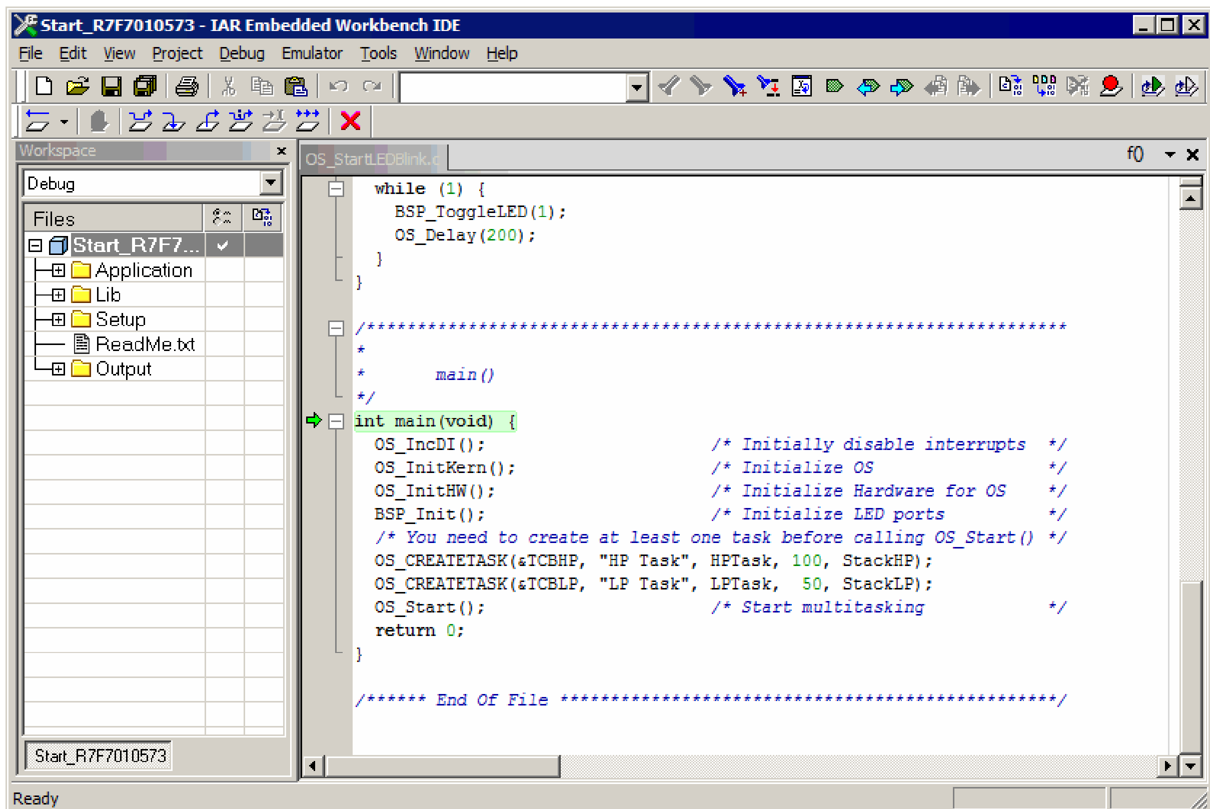
1.4 Stepping through the sample application

When starting the debugger, you will see the `main()` function (see example screen shot below). The `main()` function appears as long as project option `Run to main` is selected, which it is enabled by default. Now you can step through the program.

`OS_Init()` is part of the embOS library and written in assembler; you can there fore only step into it in disassembly mode. It initializes the relevant OS variables.

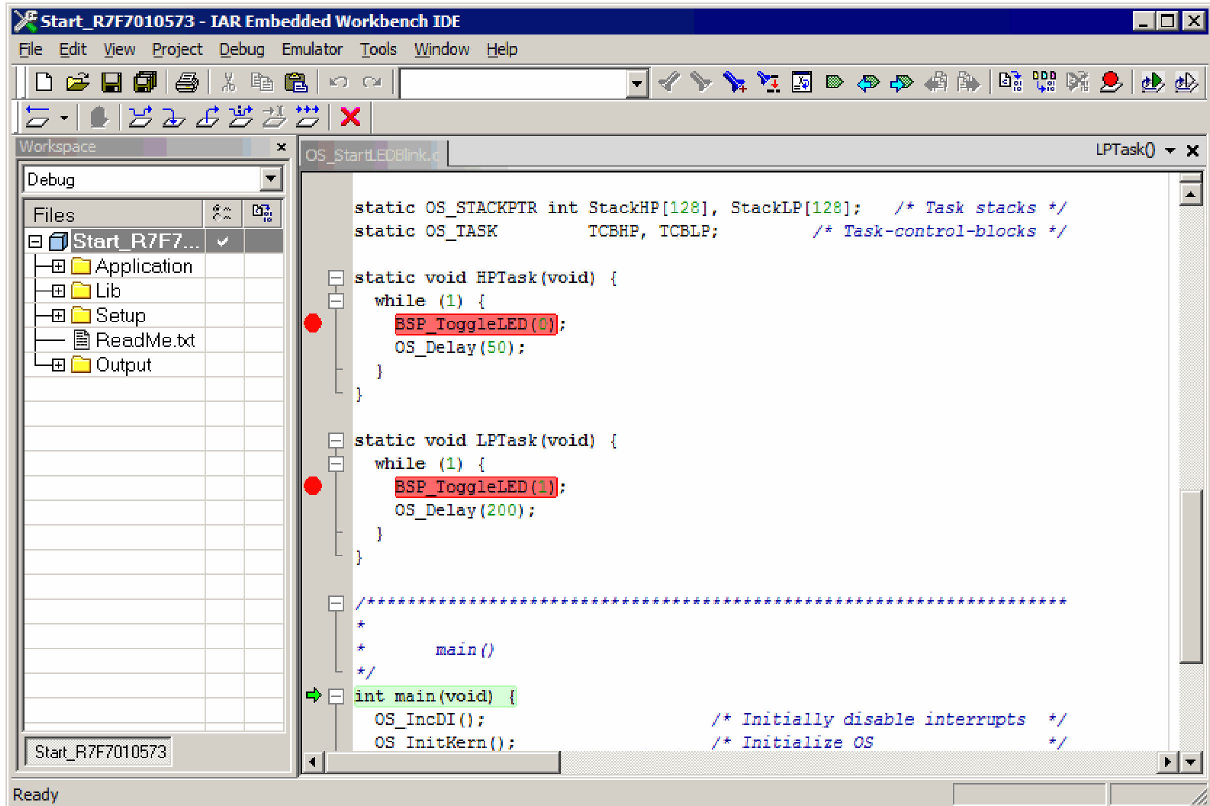
`OS_InitHW()` is part of `RTOSInit.c` and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the system tick interrupt for embOS. Step through it to see what is done.

`OS_Start()` should be the last line in `main()`, because it starts multitasking and does not return.



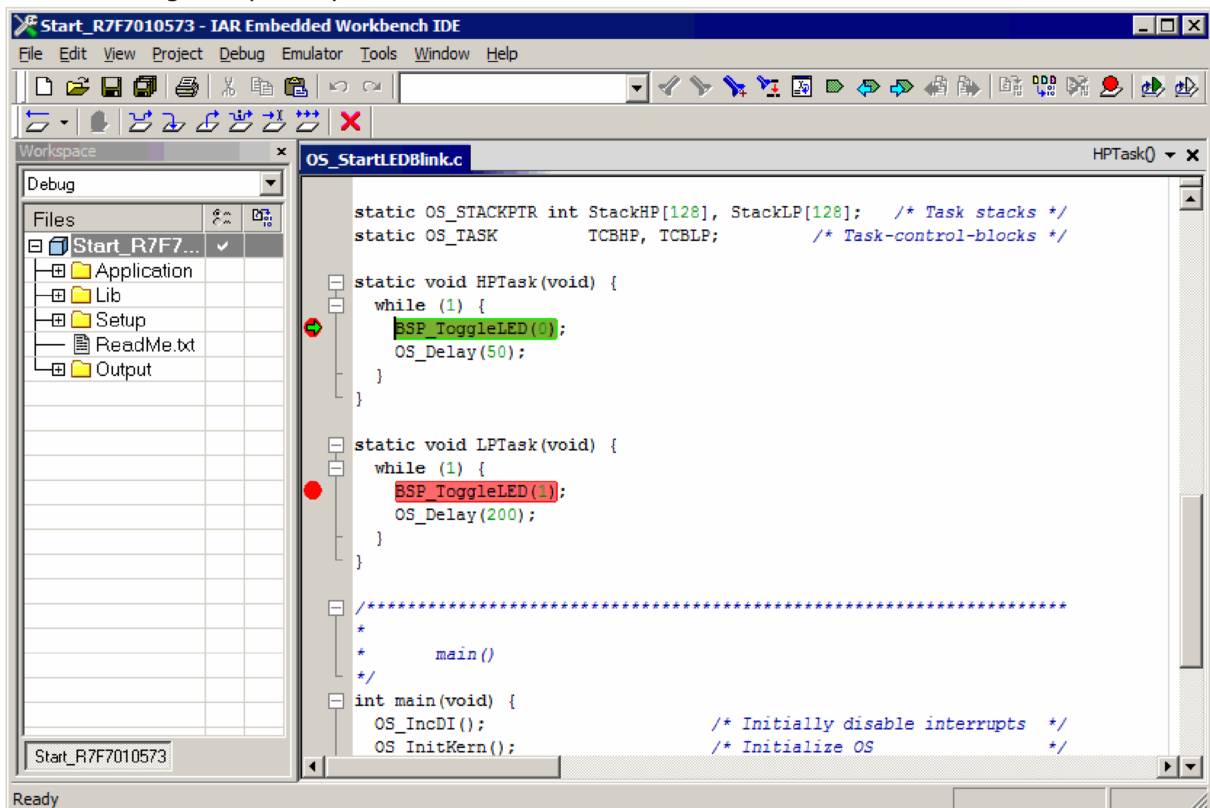
 © 2010-2018 SEGGER Microcontroller GmbH

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



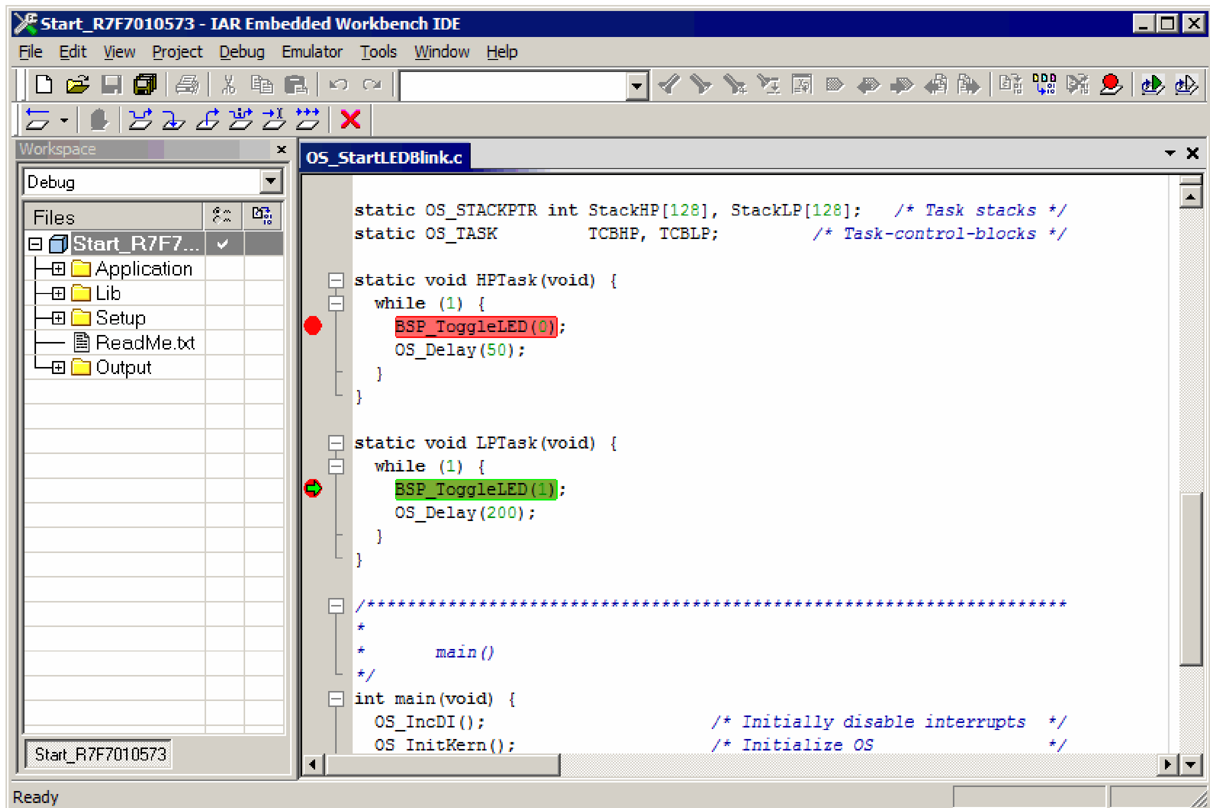
As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only.

Click `GO`, step over `OS_Start()`, or step into `OS_Start()` in disassembly mode until you reach the highest priority task.



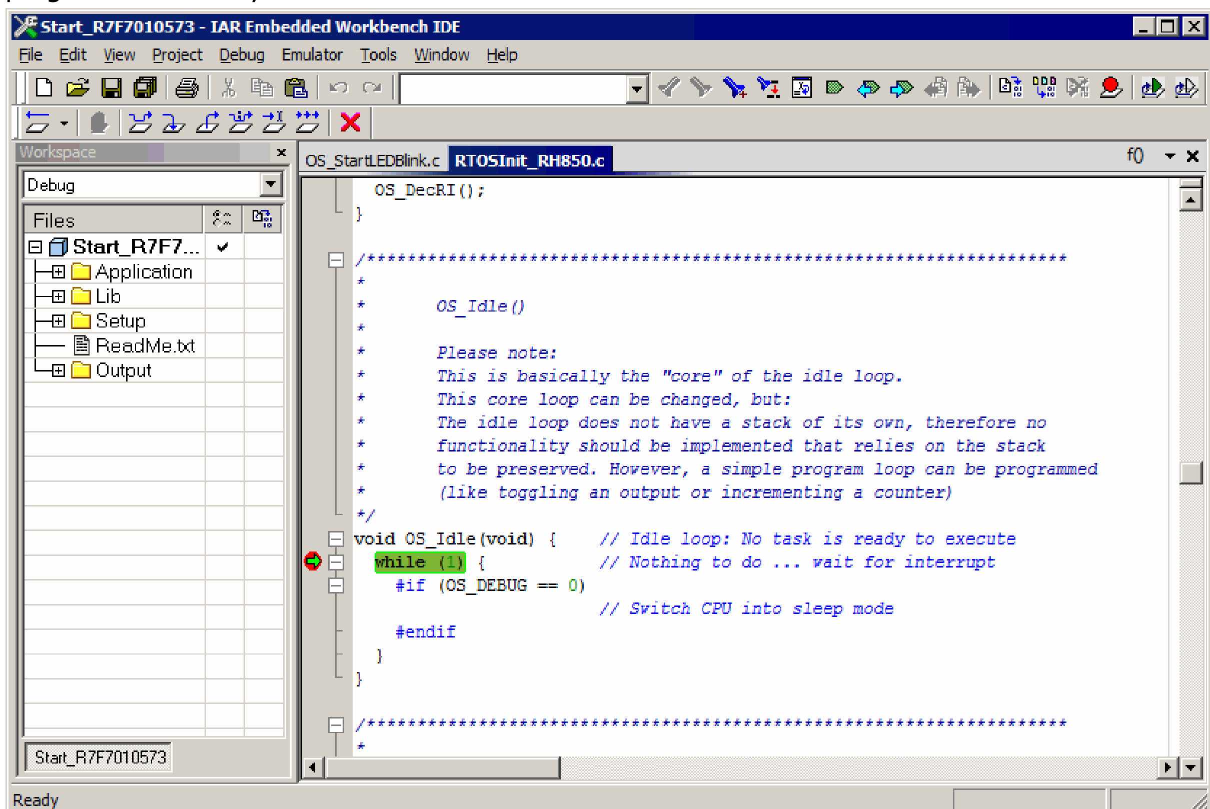
 © 2010-2018 SEGGER Microcontroller GmbH

If you continue stepping, you will arrive at the task that has lower priority:



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

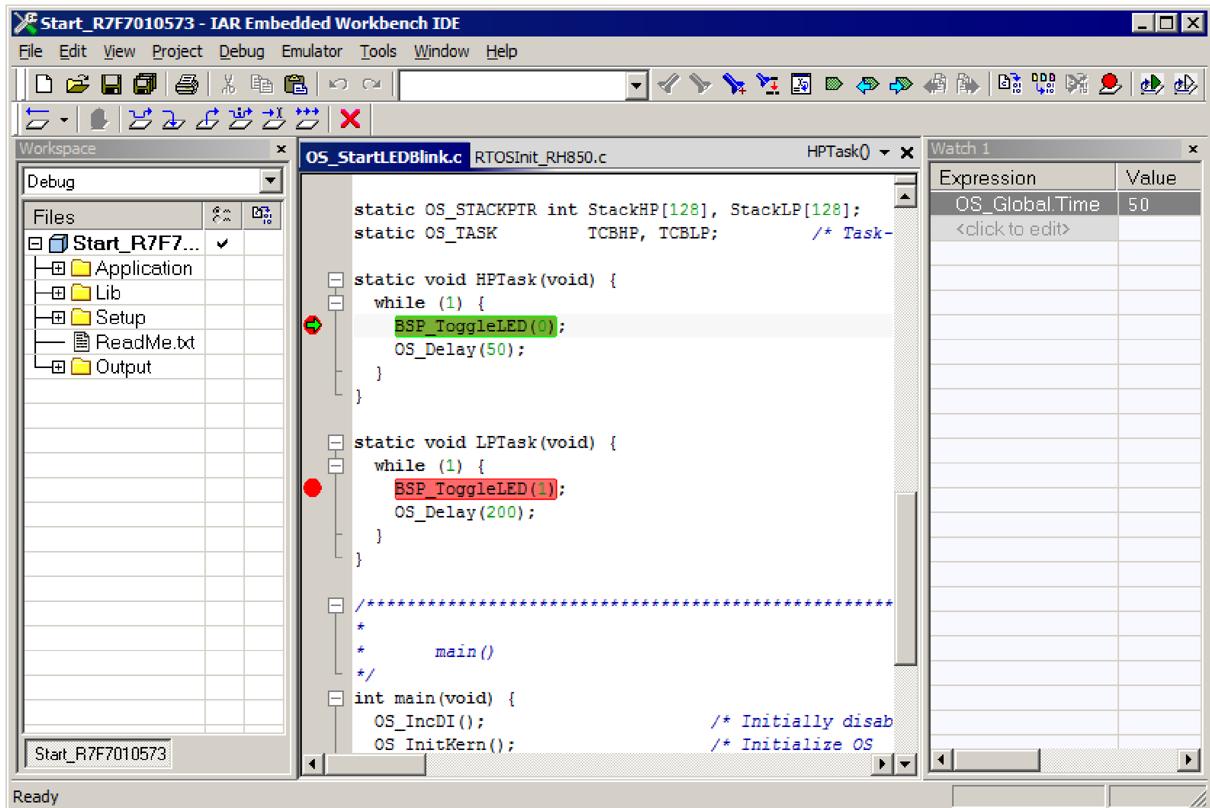
You will arrive there when you step into the `OS_TASK_Delay()` function in disassembly mode. `OS_Idle()` is part of `RTOSInit.c`. You may also set a breakpoint there before stepping over the delay in `LPTask()`.



 © 2010-2018 SEGGER Microcontroller GmbH

If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay.

As can be seen by the value of embOS timer variable `OS_Global.Time`, shown in the Watch window, `HPTask()` continues operation after expiration of the 50 system tick delay.



 © 2010-2018 SEGGER Microcontroller GmbH

Chapter 2

Build your own application

This chapter provides all information to set up your own embOS project.

© 2010-2018 SEGGER Microcontroller GmbH

2.1 Introduction

To build your own application, you should always start with one of the supplied sample workspaces and projects. Therefore, select an embOS workspace as described in chapter *First Steps* on page 10 and modify the project to fit your needs. Using an embOS start project as starting point has the advantage that all necessary files are included and all settings for the project are already done.

2.2 Required files for an embOS

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 subfolder `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 target specific `BoardSupport\<Manufacturer>\<MCU>` subfolder. It contains hardware-dependent initialization code for embOS. It initializes the system timer interrupt and optional communication for embOSView via UART or JTAG.
- `OS_Error.c` from one target specific subfolder `BoardSupport\<Manufacturer>\<MCU>`. The error handler is used if any debug library is used in your project.
- One embOS library from the subfolder `Lib\`.
- Additional CPU and compiler specific files may be required according to CPU.

When you decide to write your own startup code or use a low level `init()` function, 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 calling `OS_Init()` and `OS_InitHW()` prior to any other embOS functions that are called.

2.3 Change library mode

For your application you might want to choose another 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 or a stack check library.

Therefore you have to select or replace the embOS library in your project or target:

- If your selected library is already available in your project, just select the appropriate configuration.
- To add a library, you may add the library to the existing Lib group. Exclude all other libraries from your build, delete unused libraries or remove them from the configuration.
- Check and set the appropriate `OS_LIBMODE_*` define as preprocessor option and/ or modify the `OS_Config.h` file accordingly.

2.4 Select another CPU

embOS contains CPU-specific code for various CPUs. Manufacturer- and CPU-specific sample start workspaces and projects are located in the subfolders of the `BoardSupport\` folder. To select a CPU which is already supported, just select the appropriate workspace from a CPU-specific folder.

If your CPU is currently not supported, examine all `RTOSInit.c` files in the CPU-specific subfolders and select one which almost fits your CPU. You may have to modify `OS_InitHW()`, `OS_COM_Init()`, the interrupt service routines for embOS system timer tick and communication to embOSView and the low level initialization.

Chapter 3

Libraries

This chapter includes CPU-specific information such as CPU-modes and available libraries.

 © 2010-2018 SEGGER Microcontroller GmbH

3.1 Naming conventions for prebuilt libraries

embOS is shipped with different pre-built libraries with different combinations of the following features:

- Alignment of `double` and `long long` data types - `Alignment`
- Size of `double` type - `Size_of_double`
- Data model - `Data_Model`
- Short adressing - `Short_Addressing`

The libraries are named as follows:

```
os<CPU><Code_Model><Alignment><Lib_Config><Core><Size_of_double><FPU><Da-
ta_Model><Short_adressing>_<Libmode>.a
```

Parameter	Meaning	Values
`CPU`	Specifies the CPU variant.	85: RH850 microcontroller
`Code_Model`	Specifies the code model. Code models are not supported by the compiler, this part of the library name is for forward compatibility only.	n: none
`Alignment`	Specifies the alignment of `double` and `long long` data types.	4: 4-byte alignment 8: 8-byte alignment
`Lib_Config`	Specifies the library configuration.	n: Normal
`Core`	Specifies the core variant.	5: RH850 core
`Size_of_double`	Specifies the size of `double` data type.	f: 32 bits d: 64 bits
`FPU`	Specifies the FPU support.	: no FPU
`Data_Model`	Specifies the data model.	s: small m: medium l: large
`Short_adressing`	Specifies the usage of short addressing.	s: short addressing n: no short addressing
`Libmode`	Specifies the library mode.	XR: Extreme Release R: Release S: Stack check SP: Stack check + profiling D: Debug DP: Debug + profiling DT: Debug + profiling + trace

Example

`os85n8n5dsn_DP.a` is the library for a project using 8-byte alignment, 64-bit doubles, and small data model without short addressing with debug and profiling support.

`os85n4n5fss_DP.a` is the library for a project using 4-byte alignment, 32-bit doubles, and small data model with short addressing with debug and profiling support.

 © 2010-2018 SEGGER Microcontroller GmbH

Chapter 4

CPU and compiler specifics

© 2010-2018 SEGGER Microcontroller GmbH

4.1 CPU modes

embOS for RH850 and IAR supports small, medium and large data models with and without short addressing. Tiny data model is currently not supported.

The IAR compiler provides a set of extended keywords which can be used as data memory attributes. These keywords let you override the default memory type for individual data objects:

Keyword	Default in data model	Address range
`__near`	`Tiny`	Â± 32 KBytes of memory around `0x0`.
`__brel`	`Small`	64 KBytes anywhere in RAM and 64 KBytes anywhere in ROM.
`__brel23`	`Medium`	8 MBytes in RAM and 8 MBytes in ROM.
`__huge`	`Large`	Full memory.
`__saddr`	`---`	`EP` to `EP` + 255 Bytes (`EP` = Element pointer, an alias for register `R30`). Objects that can be accessed using byte access can only occupy 128 of these bytes. If the variable data contains an `unsigned char` or `unsigned short`, only the first 32 bytes can be accessed. Due to the above limitations, short addressing is never used for embOS data objects, but may still be used by the application.

© 2010-2018 SEGGER Microcontroller GmbH

4.2 Standard system libraries

embOS for RH850 and IAR may be used with IAR standard libraries.

If non thread-safe functions are used from different tasks, embOS functions may be used to encapsulate these functions and guarantee mutual exclusion.

The system libraries from the IAR Embedded Workbench come with built-in hook functions, which enable thread safe calls of all system functions automatically when supported by the operating system.

embOS compiled for IAR Embedded Workbench is prepared to use these hook functions. Adding source code modules which are delivered with embOS activates the automatic thread locking functionality of the new IAR DLib.

4.3 Thread-safe system libraries

Using embOS with C++ projects and file operations or just normal call of heap management functions may require thread-safe system libraries if these functions are called from different tasks. Thread-safe system libraries require some locking mechanism which is RTOS specific.

To activate thread safe system library functionality, special source modules delivered with embOS have to be included in the project.

To enable the automatic thread safe locking functions, the source module xmtx.c which is included in every CPU specific Setup folder of the embOS shipment has to be included in the project and the function `OS_INIT_SYS_LOCKS()` must be called. Additionally the option “Enable thread support in library” must be set in “Project>Options>General Options>Library Configuration”.

To support thread safe file i/o functionality, the source module `xmtx2.c` has to be added. To support C++ dynamic lock functionality, the source module `xmtx3.c` has to be added.

The embOS libraries come with all code required to automatically handle the thread safe system libraries when the source module `xmtx.c`, `xmtx2.c` and `xmtx3.c` from the embOS shipment are included in the project.

Note that thread safe system library, file i/o and C++ dynamic lock support is required only, when non thread safe functions are called from multiple tasks, or thread local storage is used in multiple tasks.

 © 2010-2018 SEGGER Microcontroller GmbH

4.4 Thread-Local Storage TLS

The dlib of EWRH850 supports usage of thread-local storage. Several library objects and functions need local variables which have to be unique to a thread. Thread-local storage will be required when these functions are called from multiple threads.

embOS for RH850 is prepared to support the tread-local storage, but does not use it per default. This has the advantage of no additional overhead as long as thread-local storage is not needed by the application. The embOS implementation of thread-local storage allows activation of TLS sepa- rately for every task.

Only tasks that call functions using TLS need to activate the TLS by calling an initial- ization function when the task is started.

The IAR runtime environment allocates the TLS memory on the heap. Using TLS with multiple tasks shall therefore use thread safe system library functionality which is automatically enabled when the `xmtx.c` module from the embOS distribution is added to the project.

Library objects that need thread-local storage when used in multiple tasks are:

- error functions -- errno, strerror.
- locale functions -- localeconv, setlocale.
- time functions -- asctime, localtime, gmtime, mktime.
- multibyte functions -- mbrlen, mbrtowc, mbsrtowc, mbtowc, wcrtomb, wcsrtomb, wctomb.
- rand functions -- rand, srand.
- etc functions -- atexit, strtok.
- C++ exception engine.

4.4.1 OS_TASK_SetContextExtensionTLS()

Description

`OS_TASK_SetContextExtensionTLS()` may be called from a task to initialize and use Thread-local storage.

Prototype

```
void OS_TASK_SetContextExtensionTLS(void);
```

Additional information

`OS_TASK_SetContextExtensionTLS()` shall be the first function called from a task when TLS should be used in the specific task. The function must not be called multiple times from one task. The thread-local storage is allocated on the heap. To ensure thread safe heap management, the thread safe system library functionality shall also be enabled when using TLS.

Thread safe system library calls are automatically enabled when the source module `xmtx.c` which is delivered with embOS in the BSP Setup folders is included in the project.

Example

The following printout demonstrates the usage of task specific TLS in an application.

```
#include "RTOS.h"

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

static void HPTask(void) {
  OS_TASK_SetContextExtensionTLS();
  while (1) {
    errno = 42;  // errno specific to HPTask
    OS_TASK_Delay(50);
  }
```

 © 2010-2018 SEGGER Microcontroller GmbH

```
}

static void LPTask(void) {
  OS_TASK_SetContextExtensionTLS();
  while (1) {
    errno = 1;  // errno specific to LPTask
    OS_TASK_Delay(200);
  }
}

int main(void) {
  errno = 0;      // errno not specific to any task
  OS_Init();      // Initialize embOS
  OS_InitHW();    // Initialize required hardware
  OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
  OS_TASK_CREATE(&TCBLP, "LP Task",  50, LPTask, StackLP);
  OS_Start();     // Start embOS
  return 0;
}
```

Chapter 5

Stacks

This chapter describes how embOS uses the different stacks of the RH850 CPU.

 © 2010-2018 SEGGER Microcontroller GmbH

5.1 Task stack

Each task uses its individual stack. The stack pointer is initialized and set every time a task is activated by the scheduler. The stack-size required for a task 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 calling embOS-routines.

The minimum basic task stack size is about 56 bytes. Because any function call uses some amount of stack and every exception may push additional bytes onto the current stack, the task stack size has to be large enough to handle one exception too. We recommend at least 256 bytes stack as a start.

5.2 System stack

The minimum system stack size required by embOS is about 136 bytes (stack check & profiling build). 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 C-level interrupt handlers may also use the system-stack, the actual stack requirements depend on the application.

The size of the system stack can be changed by modifying the project settings. We recommend a minimum stack size of 256 bytes for the `CSTACK`.

5.3 Interrupt stack

RH850 CPUs do not support a separate hardware interrupt stack. All interrupts primarily run on the current stack. To reduce task stack load by interrupts, embOS may use the system stack as interrupt stack. Interrupt handlers should use `OS_INT_EnterIntStack()` and `OS_INT_LeaveIntStack()` to switch to the interrupt stack. Only the first level interrupt will use some amount of task stack in this case. Please also refer to chapter *Interrupts* on page 27.

 © 2010-2018 SEGGER Microcontroller GmbH

Chapter 6

Interrupts

© 2010-2018 SEGGER Microcontroller GmbH

6.1 What happens when an interrupt occurs?

- The CPU-core receives an interrupt request.
- As soon as the 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 calculates the exception handler address according to the current PSW value.
- The CPU saves the current PC in the EIPC register.
- The CPU saves the current PSW in the EIPSW register.
- An exception is written into EEIC and PSW is updated.
- The calculated exception handler address is stored in the PC register.
- Further interrupts are disabled, the PSW.EP bit is cleared.
- The exception handler is executed.
- ISR: Save registers.
- ISR: User-defined functionality.
- ISR: Restore registers.
- ISR: Execute the EIRET command, restoring the saved PSW and 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 `EIRET`.

The corresponding interrupt vector number or according interrupt vector name 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 C/C++ Development Guide for the Renesas RH850 family.

Example

Simple interrupt routine:

```
#pragma vector=INTRLIN30UR0_vector
__interrupt void RLIN30_TX_Handler(void) {
  SendNextChar();
}
```

Interrupt routine using embOS functions:

```
#pragma vector=INTOSTM0_vector
__interrupt void _Systick(void) {
  OS_INT_EnterNestable(); // Inform embOS about ISR and enable interrupts.
  OS_INT_EnterIntStack();          // Use interrupt stack.
  OS_HandleTick();
  OS_INT_LeaveIntStack();          // Leave interrupt stack.
  OS_INT_LeaveNestable(); // Inform embOS about return from ISR.
}
```

6.3 Interrupt vector table

The IAR toolchain automatically generates the interrupt vector table, which by default is populated with a default interrupt handler that calls the abort function. For each interrupt source that has no explicit interrupt service routine, the default interrupt handler will be called. If you write your own service routine for a specific vector, that routine will override the default interrupt handler. The interrupt vector number has to be assigned to the interrupt handler function by a `#pragma vector` declaration right in front of the interrupt handler function in the source code. The header file `iodevice.h`, where `device` corresponds to the selected device, contains predefined names for the existing interrupt vectors. The embOS

 © 2010-2018 SEGGER Microcontroller GmbH

timer interrupt handler is located in the in the source code file RTOSInit_*.c. With the RH850 microcontroller, the reset vector always starts at address 0x0, which is the base for the exception vectors pointed to by the RBASE system register. The interrupt vector base is pointed to by the system register INTBP. The exception vector can be moved and is then pointed to by the system register EBASE.

6.4 Interrupt-stack switching

Since the RH850 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_INT_EnterIntStack() at the beginning of an interrupt handler just after the call of the embOS ISR entry function OS_INT_Enter() or OS_INT_EnterNestable(), and OS_INT_LeaveIntStack() at the end just before calling OS_INT_LeaveNestable() or OS_INT_Leave().

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 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.

6.5 Interrupt nesting

The RH850 CPU uses a priority controlled interrupt scheduling which allows nesting of interrupts per default. Any interrupt or exception with a higher preemption priority may interrupt an interrupt handler running on a lower preemption priority. An interrupt handler calling embOS functions has to start with an embOS prolog function: it informs embOS that an interrupt handler is running. For any interrupt handler, the user may decide individually whether this interrupt handler may be preempted or not by choosing the prolog function.

 © 2010-2018 SEGGER Microcontroller GmbH

6.5.1 OS_INT_Enter()

Description

`OS_INT_Enter()` disables nesting.

Prototype

```
void OS_INT_Enter(void);
```

Additional information

`OS_INT_Enter()` has to be used as prolog function, when the interrupt handler should not be preempted by any other interrupt handler. An interrupt handler that starts with `OS_INT_Enter()` has to end with the epilog function `OS_INT_Leave()`.

Example

Interrupt-routine that can not be preempted by other interrupts.

```
#pragma vector=INTOSTM0_vector
__interrupt void _Systick(void) {
  OS_INT_Enter();  // Inform embOS that interrupt code is running
  OS_INT_EnterIntStack();   // Use interrupt stack.
  OS_HandleTick();      // Can not be interrupted by higher priority interrupts
  OS_INT_LeaveIntStack();   // Leave interrupt stack.
  OS_INT_Leave();  // Inform embOS that interrupt handler is left
}
```

 © 2010-2018 SEGGER Microcontroller GmbH

6.5.2 OS_INT_EnterNestable()

Description

`OS_INT_EnterNestable()` enables nesting.

Prototype

```
void OS_INT_EnterNestable(void);
```

Additional information

`OS_INT_EnterNestable()`, allow nesting. `OS_INT_EnterNestable()` may be used as prolog function, when the interrupt handler may be preempted by any other interrupt handler that runs on a higher interrupt priority. An interrupt handler that starts with `OS_INT_EnterNestable()` has to end with the epilog function `OS_INT_LeaveNestable()`.

Example

Interrupt-routine that can be preempted by other interrupts.

```
#pragma vector=INTOSTM0_vector
__interrupt void _Systick(void) {
  OS_INT_EnterNestable();  // Inform embOS that interrupt code is running
  OS_INT_EnterIntStack();           // Use interrupt stack.
  OS_HandleTick();
  // Can be interrupted by higher priority interrupts
  OS_INT_LeaveIntStack();           // Leave interrupt stack.
  OS_INT_LeaveNestable();  // Inform embOS that interrupt handler is left
}
```

Chapter 7

Technical data

This chapter lists technical data of embOS used with RH850 CPUs.

© 2010-2018 SEGGER Microcontroller GmbH

7.1 Memory requirements

These values are neither precise nor guaranteed, but they give you a good idea of the memory requirements. They vary depending on the current version of embOS. The minimum ROM requirement for the kernel itself is about 1.700 bytes.

In the table below, which is for X-Release build, you can find minimum RAM size requirements for embOS resources. Note that the sizes depend on selected embOS library mode.

embOS resource	RAM [bytes]
Task control block	36
Software timer	20
Mutex	16
Semaphore	8
Mailbox	24
Queue	32
Task event	0
Event object	12

© 2010-2018 SEGGER Microcontroller GmbH