embOS

Real-Time Operating System

CPU & Compiler specifics for Renesas RX using IAR compiler for RX

Document: UM01020
Software Version: 5.18.3.0
Revision: 0
Date: March 27, 2024

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

The information written in this document is assumed to be accurate without guarantee. The information in this manual is subject to change for functional or performance improvements without notice. SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions in this document. SEGGER disclaims any warranties or conditions, express, implied or statutory for the fitness of the product for a particular purpose. It is your sole responsibility to evaluate the fitness of the product for any specific use.

Copyright notice

Trademarks

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

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

Contact address

SEGGER Microcontroller 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

*By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at https://www.segger.com/legal/privacy-policy/.

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: March 27, 2024

Software	Revision	Date	By	Description
5.18.3.0	0	240327	TS	New software version.
5.18.1.0	0	230428	TS	New software version.
5.18.0.0	0	230328	TS	New software version.
5.16.1.0	0	220217	TS	New software version.
5.14.0.0	0	210802	MM	New software version.
5.12.0.0	0	201014	MC	New software version.
5.10.2.0	0	200715	MM	New software version.
5.06	1	190625	TS	Chapter “Libraries” and “Interrupts” regarding RXv3 updated.
5.06	0	190325	MC	New software version.
5.02	0	180710	TS	New software version.
4.24	0	160704	RH	Chapter “RTT and SystemView” added.
4.14	0	151123	TS	New software version.
4.10b	0	150601	TS	New software version.
4.10a	0	150519	TS	New software version.
3.90a	0	140410	TS	New generic embOS sources V3.90a. RXv2 library description added. DSP accumulator handling description added.
3.88g	0	131211	AW	New generic embOS sources V3.88g. Version 3.88g1 for EWRX V2.50.
3.88	0	130319	AW	New generic embOS sources V3.88. BSP for RX 100 CPU added.
3.86i	0	120926	TS	New generic embOS sources V3.86i.
3.86g	0	120806	AW	New generic embOS sources V3.86g.
3.86e	0	120614	TS	First FrameMaker version of the manual.

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.

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.

Table of contents

Chapter 1

Using embOS

1.1 Installation

This chapter describes how to get started with embOS. You should follow these steps to become familiar with embOS.

embOS is shipped as a zip-file in electronic form. To install it, you should extract the zip-file to any folder of your choice while preserving its directory structure (i.e. keep all files in their respective sub directories). Ensure the files are not read-only after extraction. Assuming that you are using an IDE to develop your application, no further installation steps are required.

> **Note**
>
> The projects at `/Start/BoardSupport/<DeviceManufacturer>/<Board>` assume a relative location for the `/Start/Lib` and `/Start/Inc` folders. If you copy a BSP folder to another location, you will need to adjust the include paths of the project accordingly.

At `/Start/BoardSupport/<DeviceManufacturer>/<Board>` you should find several example start projects, which you may adapt to write your application. To do so, follow the instructions of section *First Steps* on page 11.

In order to become familiar with embOS, consider using the example projects (even if you will not use the IDE for application development).

If you do not or do not want to work with an IDE, you may copy either all library files or only the library that is used with your project into your work directory. embOS does in not rely on an IDE, but may be used without an IDE just as well, e.g. using batch files or a make utility.

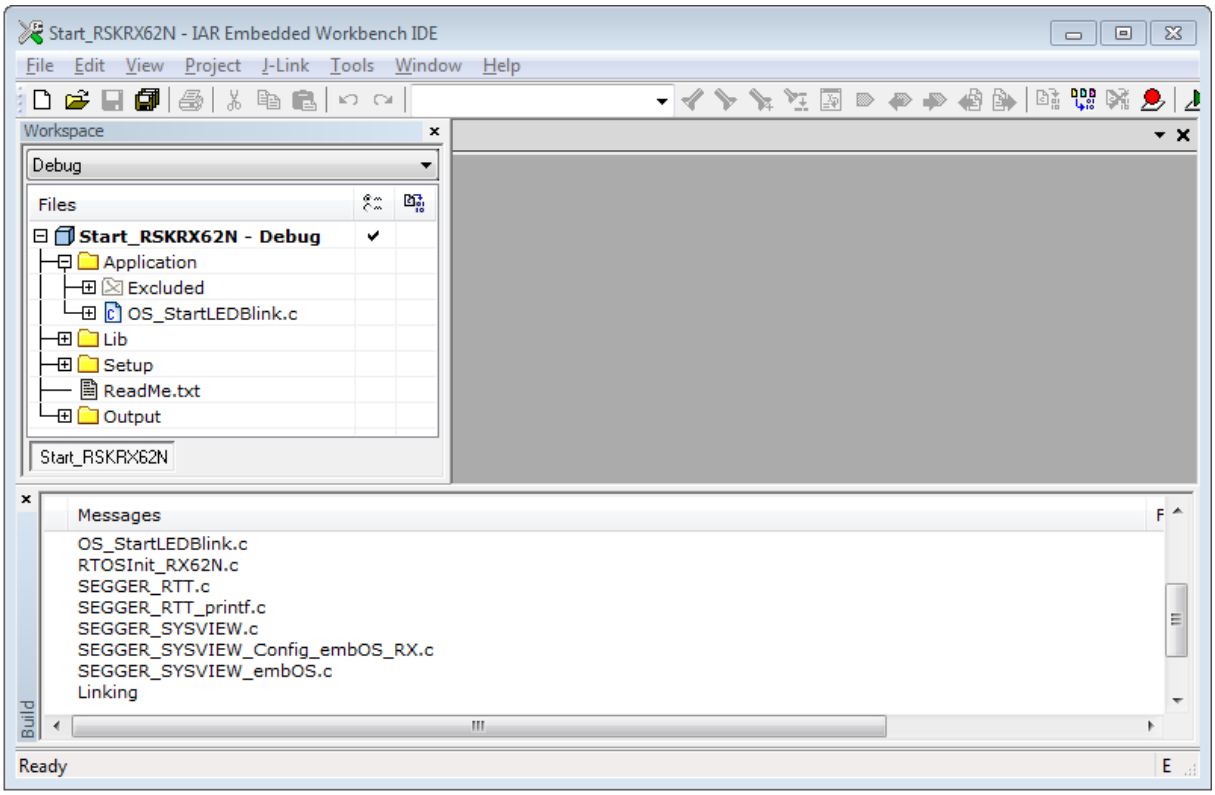
1.2 First Steps

After installation of embOS, you can create your first multitasking application. You received several ready-to-go sample workspaces and projects as well as all required embOS files inside the subfolder `Start`. The subfolder `Start/BoardSupport` contains the workspaces and projects, sorted into manufacturer- and board-specific subfolders. It is a good idea to use one of the projects as a starting point for any application development.

To get your new application running, you should:

- Create a directory for your development.
- Copy the whole `Start` folder from your embOS shipment into the directory.
- Clear the read-only attribute of all files in the copied `Start` folder.
- Open one sample workspace/project in `Start/BoardSupport/<DeviceManufacturer>/<Board>` with your IDE (for example, by double clicking it).
- Build the project. It should be built without any error or warning messages.

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



For additional information, you should open the ReadMe.txt file that is part of every BSP. It describes the different configurations of the project and, if required, gives additional information about specific hardware settings of the supported evaluation board(s).

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 (the actual file shipped with your port of embOS may differ slightly).

What happens is easy to see:

After initialization of embOS, two tasks are created and started. The two tasks get activated and execute until they run into a delay, thereby suspending themselves for the specified time, and eventually 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
          an 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 ******************************/
```

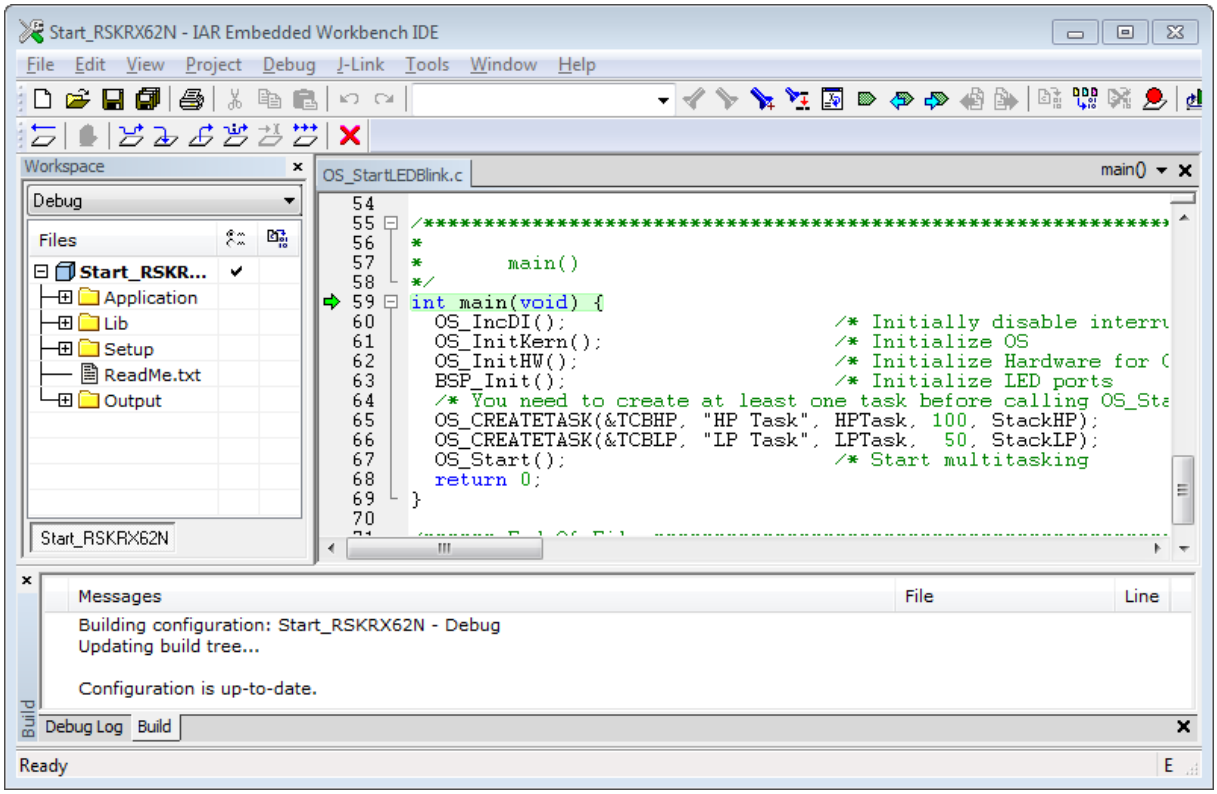

1.4 Stepping through the sample application

When starting the debugger, you will see the `main()` function (see example screenshot 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.

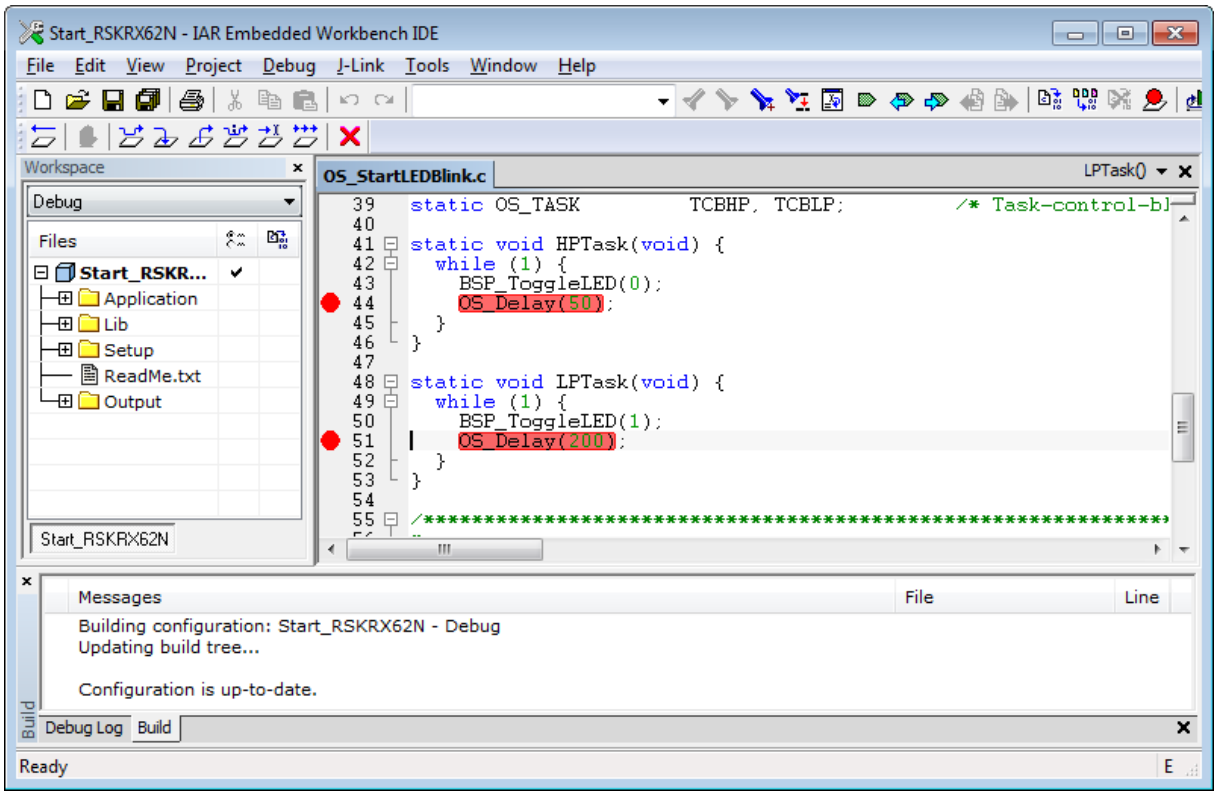
`OS_Init()` is part of the embOS library and written in assembler; you can therefore only step into it in disassembly mode. It initializes the relevant OS variables.

`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.

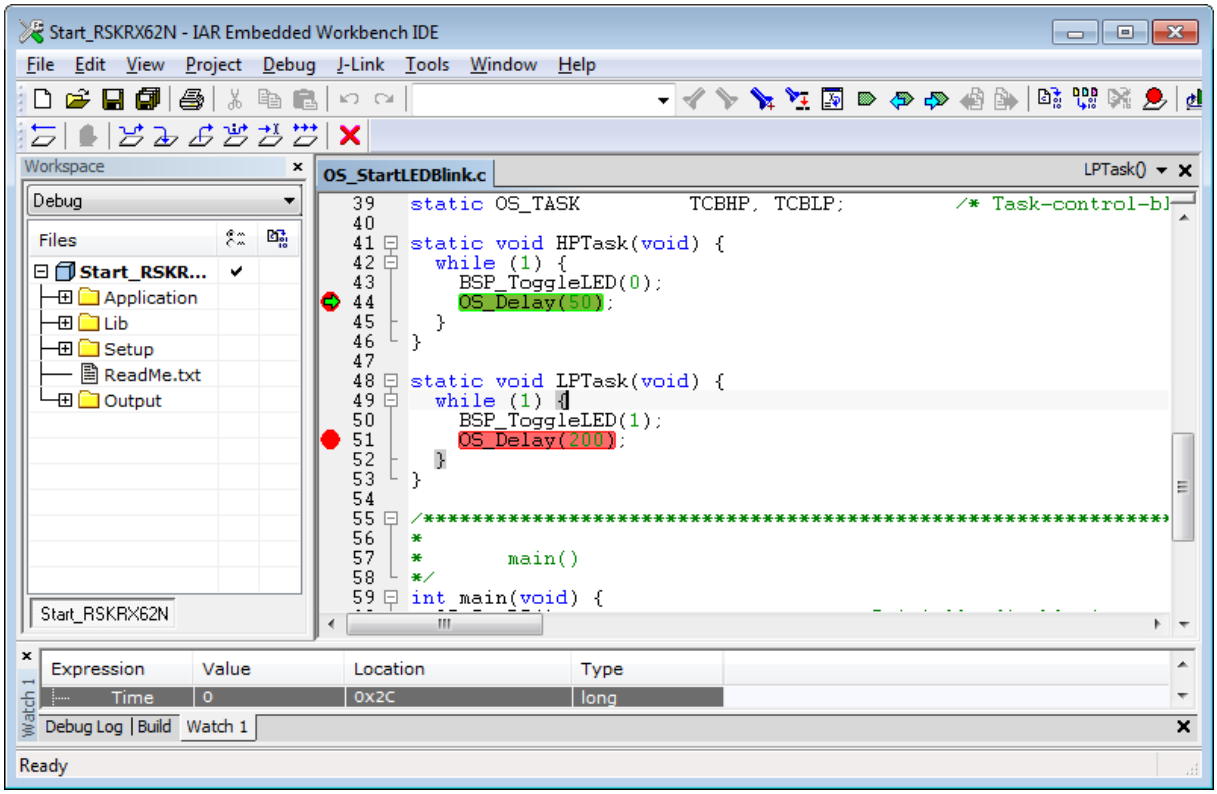


Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.

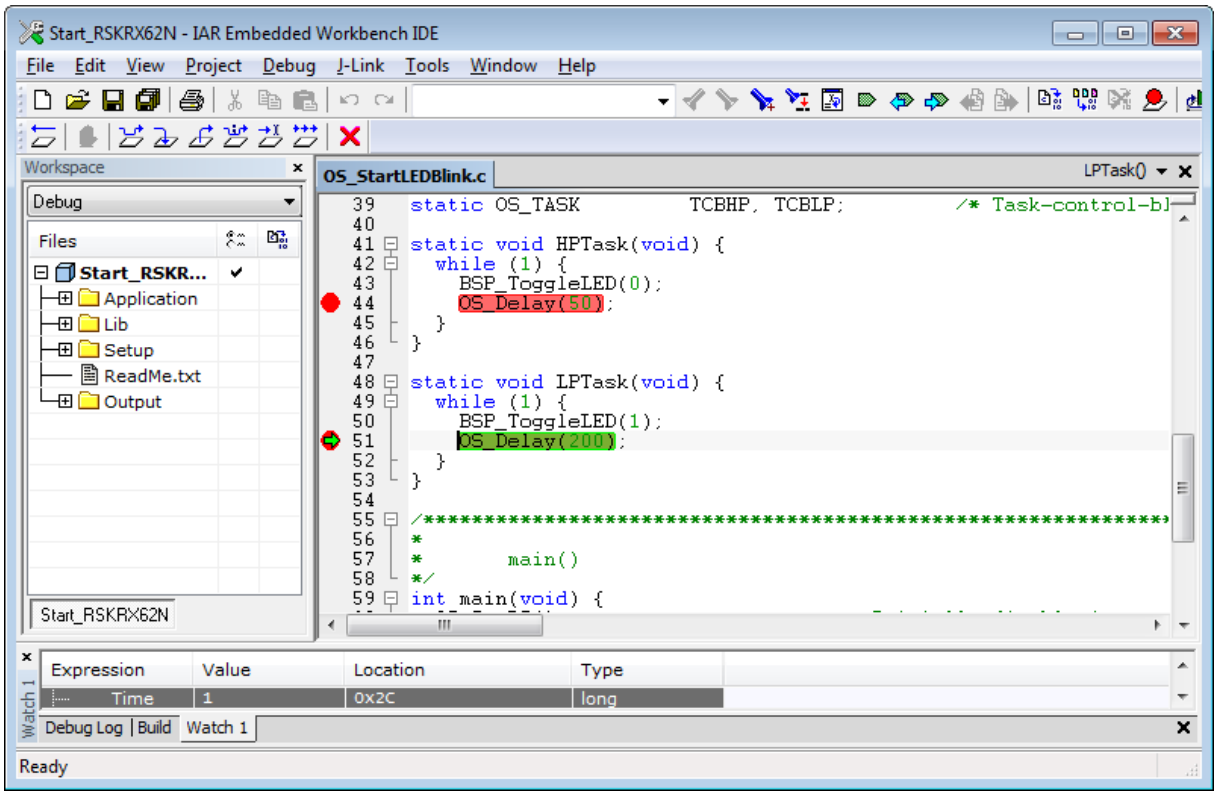


As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only.

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

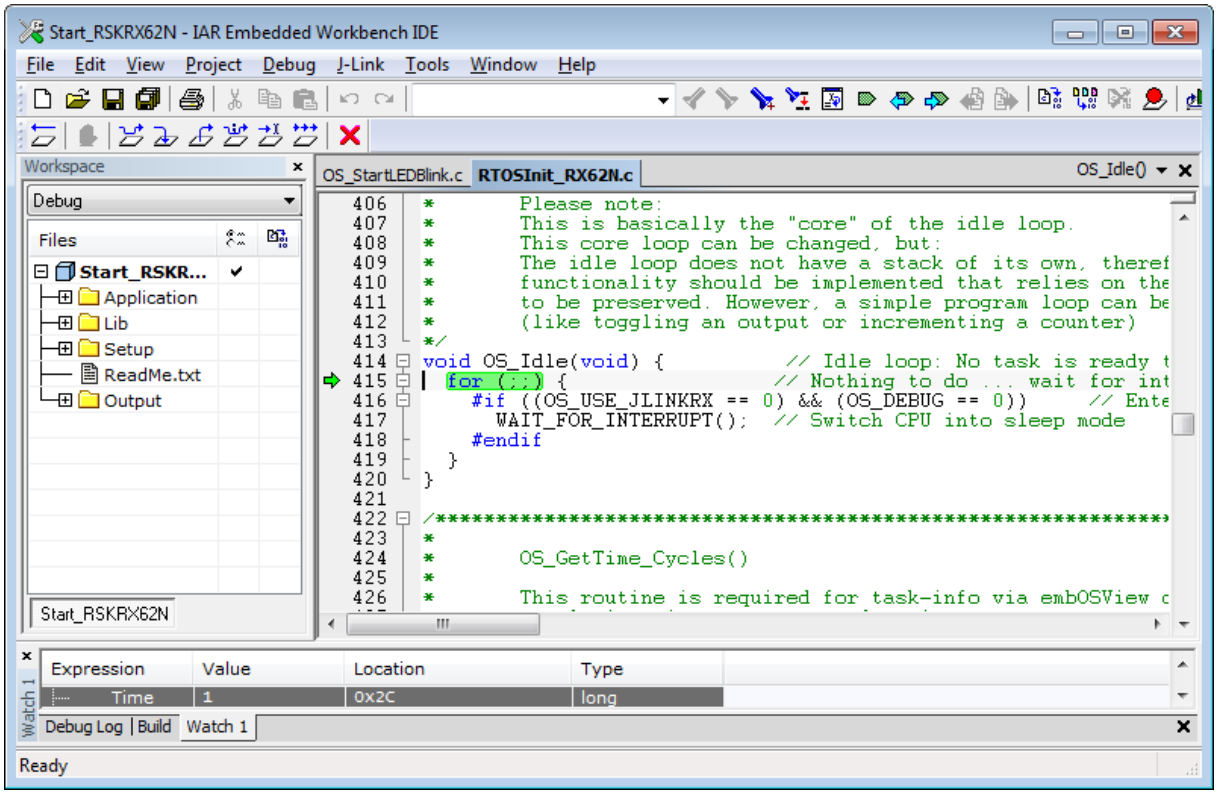


If you continue stepping, you will arrive at the task that has lower priority:



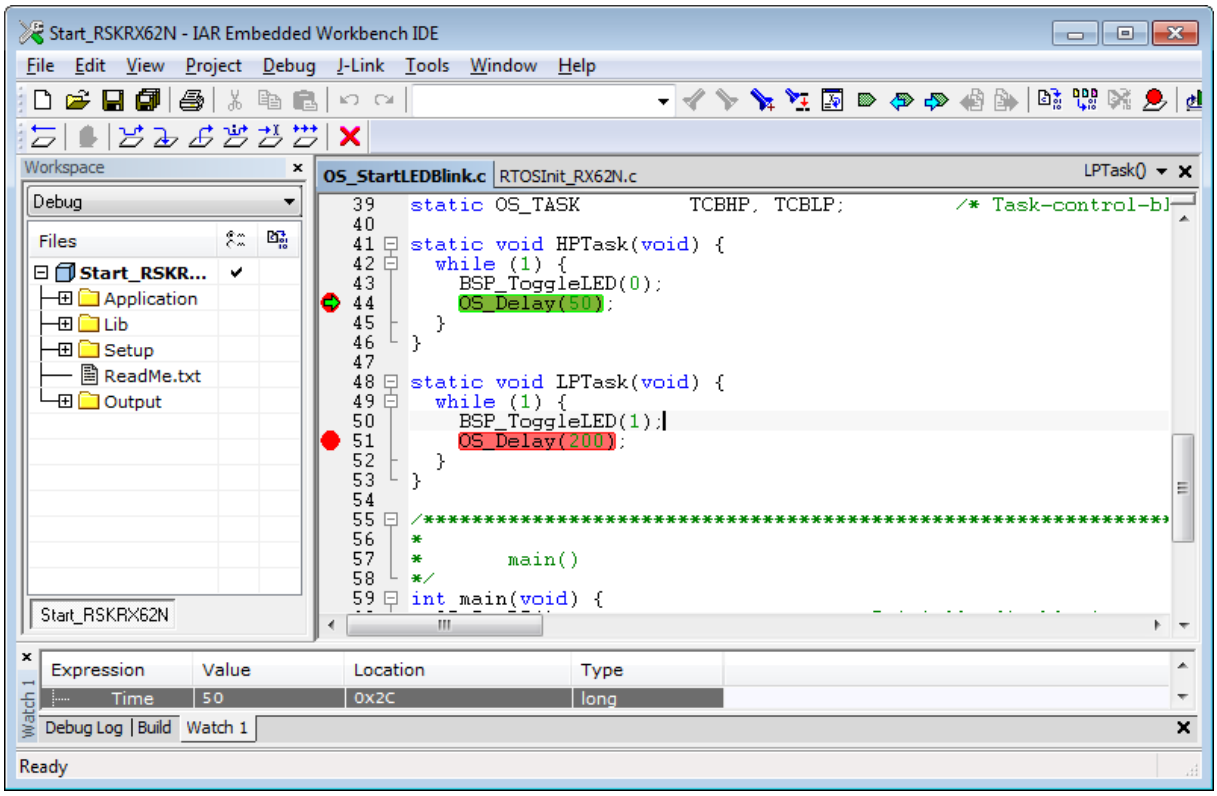
Continue to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

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()`.



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 delay.



Chapter 2

Build your own application

2.1 Introduction

This chapter provides all information to set up your own embOS project. 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 11 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 the directory `.\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** from one target specific `.\Start\BoardSupport\<Manufacturer>\<MCU>` subfolder. It contains hardware-dependent initialization code for embOS. It initializes the system timer interrupt but can also initialize or set up the interrupt controller, clocks and PLLs, the memory protection unit and its translation table, caches and so on.
- **OS_Error.c** from one target specific subfolder `.\Start\BoardSupport\<Manufacturer>\<MCU>`. The error handler is used only if a debug library is used in your project.
- One **embOS library** from the subfolder `.\Start\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 always 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 project 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 `.\Start\BoardSupport` directory. 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()`, the interrupt service routines for the embOS system tick timer and the low level initialization.

Chapter 3

Libraries

3.1 CPU modes

embOS for Renesas RX supports all memory and code model combinations that are supported by the IAR RX compiler.

Data Model	Code Model	Data Area
Near	Huge (32 bits) 0x0-0xFFFFFFFF	0x0-0x7FFF 0xFFFF8000-0xFFFFFFFF
Far	Huge (32 bits) 0x0-0xFFFFFFFF	0x0-0x7FFFF 0xFFFF8000-0xFFFFFFFF
Huge	Huge (32 bits) 0x0-0xFFFFFFFF	0x0-0xFFFFFFFF

3.2 Naming conventions for prebuilt libraries

embOS is shipped with different pre-built libraries with different combinations of features.

The libraries are named as follows:

os<Architecture><Size_of_double><Size_of_int><Endianness>_<DataModel><LibMode>.a

Parameter	Meaning	Values
Architecture	Specifies the RX core	Rx : RXv1 core Rx2: RXv2 core Rx3: RXv3 core (IAR V4.x only)
Size_of_double	Specifies the size of double data type	f : 32 bits d : 64 bits
Size_of_int	Specifies the size of integer data type	l : 32 bits
Endianness	Byte order	b : Big endian l : Little endian
DataModel	Specifies the data model	N : Near data model F : Far data model H : Huge data model
LibMode	Specifies the library mode	XR : Extreme release R : Release S : Stack check SP : Stack check + profiling D : Debug DP : Debug + stack check + profiling DT : Debug + stack check + profiling + trace

Example

osRXfll_NDP.a is the library for an RX CPU with float normal (32 bit), large integer, little endian, near memory model, with debug, stack check and profiling support.

Chapter 4

CPU and compiler specifics

4.1 IAR C-Spy stack check warning

IAR's C-Spy debugger provides a stack check feature which throws a warning when the stack pointer does not point to memory within the CSTACK scope anymore. This renders the C-Spy stack check useless, as C-Spy is not aware of any task stacks the application is using. Depending on the IAR version used, this warning can be disabled by removing the check mark for `Tools -> Options… -> Stack -> 'Warn when stack pointer is out of bounds'` or `Project -> Options… -> Debugger -> Plugins -> Stack`.

4.2 IAR C-Spy RTOS plugin

SEGGER's embOS plug-in for the IAR Embedded Workbench provides embOS awareness during debugging sessions. This enables you to inspect the state of several embOS primitives such as the task list, semaphores, mailboxes, and software timers.
SEGGER's embOS plug-in is already shipped with IAR EWARM but the most recent version can be downloaded from segger.com/products/rtos/embos/tools/plug-ins/iar-embedded-workbench.

4.3 Interrupt and thread safety

Using embOS with specific calls to standard library functions (e.g. heap management functions) may require thread-safe system libraries if these functions are called from several tasks or interrupts. IAR's system libraries provide functions, which can be overwritten to implement a locking mechanism making the system library functions thread-safe.

The Setup directory in each embOS BSP contains the file `OS_ThreadSafe.c` which overwrites these functions. By default they disable and restore embOS interrupts to ensure thread safety in tasks, embOS interrupts, `OS_Idle()` and software timers. Zero latency interrupts are not disabled and therefore unprotected. If you need to call e.g. malloc() also from within a zero latency interrupt additional handling needs to be added. If you don't call such functions from within embOS interrupts, `OS_Idle()` or software timers, you can instead use thread safety for tasks only. This reduces the interrupt latency because a mutex is used instead of disabling embOS interrupts.

You can choose the safety variant with the macro `OS_INTERRUPT_SAFE`.

- When defined to 1 thread safety is guaranteed in tasks, embOS interrupts, `OS_Idle()` and software timers.
- When defined to 0 thread safety is guaranteed only in tasks. In this case you must not call e.g. heap functions from within an ISR, `OS_Idle()` or embOS software timers.

4.3.1 Enabling thread-safe IAR system libraries

By default, IAR does not use thread-safe system libraries. As a result the implemented hook functions are not linked into the application. For more information on IAR's multithread support, please refer to the IAR Embedded Workbench manuals.

To use the thread-safe system libraries the option "`Enable thread support in library`" must be set in `Project -> Options… -> General Options -> Library Configuration`. Alternatively, the option `--threaded_lib` can be passed to the linker. Additionally the function `OS_INIT_SYS_LOCKS()` must be called.

With older IAR Embedded Workbench versions, neither the IDE option nor the linker option are available. In this case, the linker has to be told to explicitly link the hook functions by redirecting them to another symbol.

Activate the checkbox "`Use command line options`" in the dialog `Project -> Options… -> Linker -> Extra Options`. Then, in the "`Command line options:`" field, add the following lines:

```
--redirect __iar_Locksyslock=__iar_Locksyslock_mtx
```



```
--redirect __iar_Unlocksyslock=__iar_Unlocksyslock_mtx
--redirect __iar_Lockfilelock=__iar_Lockfilelock_mtx
--redirect __iar_Unlockfilelock=__iar_Unlockfilelock_mtx
--keep     __iar_Locksyslock_mtx
```

C++ thread safety

To enable thread-safe C++ constructors and destructors the option `--guard_calls` needs to be passed to the compiler.

4.4 Thread-Local Storage TLS

The DLib for IAR 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 IAR 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 separately for each task.

Only tasks that are accessing TLS variables, for instance by calling functions from the system library, need to initialize their TLS by calling an initialization function when the task is started. For each task that uses TLS the memory for the thread-local storage is allocated by the IAR runtime environment on the heap. Therefore, thread-safe heap management should be used together with TLS. For information on thread-safety, please refer to *Interrupt and thread safety* on page 22.

When the task terminates by a call of `OS_TASK_Terminate()`, the memory used for TLS is automatically freed and put back into the free heap memory.

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

- 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 API functions

Routine	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
`OS_TLS_Set()`	Initializes the thread-local storage for the current task.		•			
`OS_TLS_SetTaskContextExtension()`	Initializes the thread-local storage and sets the TLS task context extension for the current task.		•			

4.4.1.1 OS_TLS_Set()

Description

Initializes the thread-local storage for the current task.

Prototype

```
void OS_TLS_Set(void);
```

Additional information

`OS_TLS_Set()` shall be the first function called from a task when TLS should be used in this task. This function has to be used only in combination with `OS_TASK_AddContextExtension()` or `OS_TASK_SetContextExtension()` and `OS_TLS_ContextExtension` as argument to these functions. When `OS_TLS_SetTaskContextExtension()` is used, `OS_TLS_Set()` will be called automatically.

Example

```
static void Task(void) {
  OS_TLS_Set();
  OS_TASK_SetContextExtension(&OS_TLS_ContextExtension);
  while (1) {
  }
}
```


4.4.1.2 OS_TLS_SetTaskContextExtension()

Description

Initializes the thread-local storage and sets the TLS task context extension for the current task.

Prototype

```
void OS_TLS_SetTaskContextExtension(void);
```

Additional information

`OS_TLS_SetTaskContextExtension()` shall be the first function called from a task when TLS should be used in this task.

If the task already contains a task context extension, `OS_TLS_SetTaskContextExtension()` cannot be used. Instead, `OS_TASK_AddContextExtension()` needs to be called with `OS_TLS_ContextExtension` as argument. Furthermore, `OS_TLS_Set()` needs to be called to initialize TLS for this task.

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_TLS_SetTaskContextExtension();
  while (1) {
    errno = 42;  // errno specific to HPTask
    OS_TASK_Delay(50);
  }
}

static void LPTask(void) {
  OS_TLS_SetTaskContextExtension();
  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 Renesas RX CPU.

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, plus size used by exceptions.

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.

For the Renesas RX CPUs, the minimum basic task stack size is about 44 bytes. Because any function call uses some amount of stack, the task stack size has to be large enough to handle these calls. We recommend at least 128 bytes stack as a start.

5.2 System stack

The minimum system stack size required by embOS is about 60 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 also use the system-stack, the actual stack requirements depend on the application.

The size of the system stack can be configured by modifying the project option "User mode stack size" or can be defined in the linker file as `_USTACK_SIZE` in the BSP specific `Setup\` folder. We recommend a minimum stack size of 256 bytes.

5.3 Interrupt stack

The Renesas RX CU has been designed with multitasking in mind; it has 2 stack-pointer, the USP and the ISP. The U-Flag selects the active stack-pointer. During execution of a task, a software timer, or the embOS scheduler, the U-flag is set, thereby selecting the user-stack-pointer. If an interrupt occurs, the RX clears the U-flag and switches to the interrupt stack pointer automatically this way. The ISP is active during the entire ISR (interrupt service routine). This way, the interrupt does not use the stack of the task and the stack size does not have to be increased for interrupt-routines. Additional software stack-switching as for other CPUs is therefore not necessary for the RX.

The size of the interrupt stack can be configured by modifying the project option "Supervisor mode stack size" or can be defined in the linker file as `_ISTACK_SIZE` in the CPU specific `Setup\` folder. We recommend at least a minimum of 256 bytes.

Chapter 6

Interrupts

6.1 What happens when an interrupt occurs?

- The CPU receives an interrupt request.
- As soon as interrupts are enabled and the interrupt priority level (IPL) of the CPU is lower than the IPL of the interrupt, the interrupt is accepted.
- The CPU switches to the Interrupt stack.
- The CPU saves the PC and flag register on the interrupt stack.
- The CPU disables all further interrupts.
- The CPU sets its IPL to the IPL of the accepted interrupt.
- The CPU jumps to the address specified in the vector table for the interrupt service routine (ISR).
- ISR: Saves registers.
- ISR: User-defined functionality is executed.
- ISR: Restores registers.
- ISR: Executes RTE command, restoring PC, Flag register and switching back to the user stack.

For details, refer to the Renesas hard- and software manuals.

6.2 Defining interrupt handlers in C

Interrupt handlers for Renesas RX cores are written as normal C-functions which do not take parameters and do not return any value. Routines defined with the keyword `__interrupt` automatically save & restore the registers they modify and return with `RTE`.

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

For details how to write interrupt handlers using embOS functions, refer to the embOS generic manual.

For details about interrupt priorities, refer to chapter *Interrupt priorities* on page 31.

> **Note**
>
> Interrupts that use embOS API must not specify `__nested`, as this might lead to severe problems. If nestable interrupts are desired, then `OS_INT_EnterNestable()` can be used. This embOS API function ensures that interrupts are nestable. For zero latency interrupts `__nested` can be used.

Example

Simple interrupt routine:

```
//
// Interrupt handler NOT using embOS functions
//
#pragma vector=(104)
__interrupt void IntHandlerTimer(void) {
  IntCnt++;
}
//
// Interrupt function using embOS functions
//
#pragma vector=(OS_TIMER_VECT)
__interrupt void OS_ISR_Tick (void) {
  OS_INT_EnterNestable();
  OS_TICK_Handle();
  OS_INT_LeaveNestable();
}
```


6.3 Interrupt priorities

RX CPUs can have up to 16 IPLs (interrupt priority levels) reaching from 0 to 15. While most RX CPUs have 16 priority levels implemented, the RX610 CPUs only support 8 priority levels from 0 to 7.

6.4 Interrupt handling

For the Renesas RX CPU embOS delivers following functions to handle interrupts.

6.4.1 API functions

Routine	Description	main	Task	ISR	SW Timer
`OS_INT_SetPriorityThreshold()`	Sets the interrupt priority limit for zero latency interrupts.	•	•	•	•

6.4.1.1 OS_INT_SetPriorityThreshold()

Description

Sets the interrupt priority limit for zero latency interrupts.

Prototype

```
void OS_INT_SetPriorityThreshold(OS_UINT Priority);
```

Parameters

Parameter	Description
`Priority`	The highest value usable as priority for embOS interrupts. All interrupts with higher priority are never disabled by embOS. Valid range: 1 ≤ `Priority` ≤ 7 for RX610 CPUs. 1 ≤ `Priority` ≤ 15 for other RX CPUs.

Additional information

The interrupt priority limit for zero latency interrupts is set to 4 by default. This means, all interrupts with higher priority than 4 (i.e. from 5 up to the maximum CPU specific priority) will never be disabled by embOS. To disable zero latency interrupts at all, the priority limit may be set to the highest interrupt priority supported by the CPU, which is 7 for the RX610 series and 15 for other RX CPUs.

To modify the default priority limit,`OS_INT_SetPriorityThreshold()` should be called before embOS was started. In the sample start projects, `OS_INT_SetPriorityThreshold()` is not called. The start projects use the default zero latency interrupt priority limit.

Any interrupts running above the zero latency interrupt priority limit must not call any embOS function.

Note that the maximum allowed parameter is device dependent. The function will not check whether the device specific limit is exceeded. It is the users responsibility not to use a value above 7 for CPUs which do not support more than 8 priority levels.

6.4.2 Zero latency interrupts

Instead of disabling interrupts when embOS enters a critical section, the processor's IPL is increased. This prevents the execution of interrupts with an IPL lower or equal to the current IPL of the processor. All interrupts with IPL higher than the IPL threshold that embOS uses to disable interrupt are called `zero latency interrupts`.

Zero latency interrupts are never disabled by embOS.

The IPL of the processor can be increased by calling `OS_INT_Disable()`, which sets the current IPL to the IPL threshold. Initially, the IPL threshold is set to 4, but may be modified during system initialization by a call of the function `OS_INT_SetPriorityThreshold()`. Therefore, by default all interrupts with IPL 5 and greater are zero latency interrupts and can still be processed. You must not execute any embOS function from within an interrupt running on high priority.

6.4.3 embOS interrupts

Any interrupt handler using embOS API functions has to run with IPLs from 1 to the current IPL threshold. These embOS interrupt handlers have to start with a call of `OS_INT_Enter()` or `OS_INT_EnterNestable()` and must end with a call of `OS_INT_Leave()` or `OS_INT_LeaveNestable()`. Interrupt handlers running at low priorities, i.e. with priorities from 1 to the current IPL threshold, which are not calling any embOS API function are allowed, but must not re-enable interrupts!

Note

The IPL threshold between embOS interrupts and zero latency interrupts is initially set to 4, but can be changed at runtime by a call to `OS_INT_SetPriorityThreshold()`.

6.5 Interrupt nesting

The Renesas RX CPU uses a priority controlled interrupt scheduling which allows preemption and nesting of interrupts. Interrupts and exceptions with a higher priority may preempt an interrupt handler with lower priority when interrupts are enabled during execution of the interrupt service routine.

An interrupt handler calling embOS functions has to start with a call of `OS_INT_Enter()` or `OS_INT_EnterNestable()` to informs embOS that an interrupt handler is running. Using `OS_INT_EnterNestable()` enables interrupts in the interrupt handler and thus allows nesting of interrupts.

6.6 Interrupt-stack switching

Since the RX CPUs have a separate stack pointer for interrupts, there is no need for explicit software stack-switching in an interrupt routine. The routines `OS_INT_EnterIntStack()` and `OS_INT_LeaveIntStack()` are supplied for source code compatibility to other processors only and have no functionality.

6.7 Fast interrupt, RX specific

The RX CPU supports a “Fast interrupt” mode which is described in the hardware manual. The fast interrupt may be used for special purposes, but must not call any embOS function.

6.8 Non maskable interrupt, NMI

The RX CPU supports a non maskable interrupt which is described in the hardware manual. The NMI may be used for special purposes, but must not call any embOS function.

6.9 Using Register Bank Save Function with RXv3 core

Except in some products, the RXv3 CPU provides collective saving and restoring of CPU registers. In order to perform fast collective saving and restoring of CPU registers, the RXv3 CPU provides dedicated save register banks and instructions for using these banks. Using these save register banks, it is possible to perform fast register saving at the beginning of interrupt handlers, and high-speed register restoring at the end of interrupt handlers.
The save register banks can only be accessed with the SAVE instruction and RSTR instruction. Each of these banks is used to save and restore the values of the following CPU registers: all general purpose registers (R1 to R15) except for R0, the USP, the FPSW, and the accumulators (ACC0, ACC1).

The pragma directive `bank=` can be used with an interrupt function to save the values of registers to the specified register bank at the start of the interrupt, and restore them again afterward. The `SAVE` and `RSTR` instructions will be used.

The `bank=` interrupt specification can be used with any embOS or zero latency interrupt.

Example

```
//
// Interrupt function using bank pragma directive
//
#pragma vector = (TIMER_A0)
#pragma bank = 1
__interrupt void OS_ISR_Tick (void) {
  OS_INT_EnterNestable();  // Inform embOS that interrupt code is running
```



```
  OS_Tick_Handle();
  OS_INT_LeaveNestable();
}
```


Chapter 7

RTT and SystemView

7.1 SEGGER Real Time Transfer

With SEGGER's Real Time Transfer (RTT) it is possible to output information from the target microcontroller as well as sending input to the application at a very high speed without affecting the target's real time behavior. SEGGER RTT can be used with any J-Link model and any supported target processor which allows background memory access.

RTT is included with many embOS start projects. These projects are by default configured to use RTT for debug output. Some IDEs, such as SEGGER Embedded Studio, support RTT and display RTT output directly within the IDE. In case the used IDE does not support RTT, SEGGER's J-Link RTT Viewer, J-Link RTT Client, and J-Link RTT Logger may be used instead to visualize your application's debug output.

For more information on SEGGER Real Time Transfer, refer to segger.com/jlink-rtt.

7.2 SEGGER SystemView

SEGGER SystemView is a real-time recording and visualization tool to gain a deep understanding of the runtime behavior of an application, going far beyond what debuggers are offering. The SystemView module collects and formats the monitor data and passes it to RTT.

SystemView is included with many embOS start projects. These projects are by default configured to use SystemView in debug builds. The associated PC visualization application, SystemView, is not shipped with embOS. Instead, the most recent version of that application is available for download from our website.

SystemView is initialized by calling `SEGGER_SYSVIEW_Conf()` on the target microcontroller. This call is performed within `OS_InitHW()` of the respective `RTOSInit*.c` file. As soon as this function was called, the connection of the SystemView desktop application to the target can be started. In order to remove SystemView from the target application, remove the `SEGGER_SYSVIEW_Conf()` call, the `SEGGER_SYSVIEW.h` include directive as well as any other reference to `SEGGER_SYSVIEW_*` like `SEGGER_SYSVIEW_TickCnt`.

For more information on SEGGER SystemView and the download of the SystemView desktop application, refer to segger.com/systemview.

> **Note**
>
> SystemView uses embOS timing API to get at start the current system time. This requires that `OS_TIME_ConfigSysTimer()` was called before `SEGGER_SYSVIEW_Start()` is called or the SystemView PC application is started.

Chapter 8

Technical data

8.1 Resource Usage

The memory requirements of embOS (RAM and ROM) differs depending on the used features, CPU, compiler, and library model. The following values are measured using embOS library mode `OS_LIBMODE_XR`.

Module	Memory type	Memory requirements
embOS kernel	ROM	~1700 bytes
embOS kernel	RAM	~140 bytes
Task control block	RAM	36 bytes
Software timer	RAM	20 bytes
Task event	RAM	0 bytes
Event object	RAM	12 bytes
Mutex	RAM	16 bytes
Semaphore	RAM	8 bytes
RWLock	RAM	28 bytes
Mailbox	RAM	24 bytes
Queue	RAM	32 bytes
Watchdog	RAM	12 bytes
Fixed Block Size Memory Pool	RAM	32 bytes

