embOS

Real-Time Operating System

CPU & Compiler specifics for RL78 using Renesas CCRL compiler and e2 studio

Document: UM01069
Software Version: 4.32
Revision: 0
Date: March 3, 2017

A product of SEGGER Microcontroller GmbH & Co. KG

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 & Co. KG (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

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 & Co. KG

In den Weiden 11
D-40721 Hilden

Germany

Tel. +49 2103-2878-0
Fax. +49 2103-2878-28
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: March 3, 2017

Software	Revision	Date	By	Description
4.32	0	170303	MC	Initial version

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.
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
`User Input`	Text entered at the keyboard by a user in a session transcript.
`Secret Input`	Text entered at the keyboard by a user, but not echoed (e.g. password entry), in a session transcript.
Reference	Reference to chapters, sections, tables and figures or other documents.
Emphasis	Very important sections.

Table of contents

Chapter 1

Using embOS

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

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 12.

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.

1.2 Using Renesas e2 studio

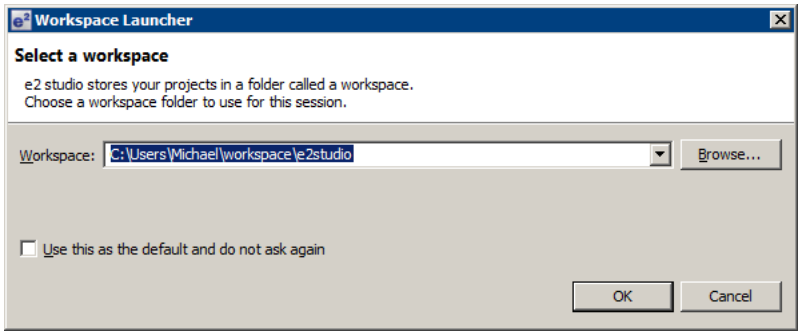
The start projects are based on e2 studio and include the necessary project files for Renesas e2studio.

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

- Create a work directory for your application, for example `C:\embOS`
- Copy the whole folder `Start` which is part of your embOS distribution into your work directory.
- Start Renesas e2studio and select and create a workspace.
- Import the sample start project into the workspace.
- Build the start project
- Run the application using e2studio HardwareDebug configuration using the E1 emulator for downloading and debugging.

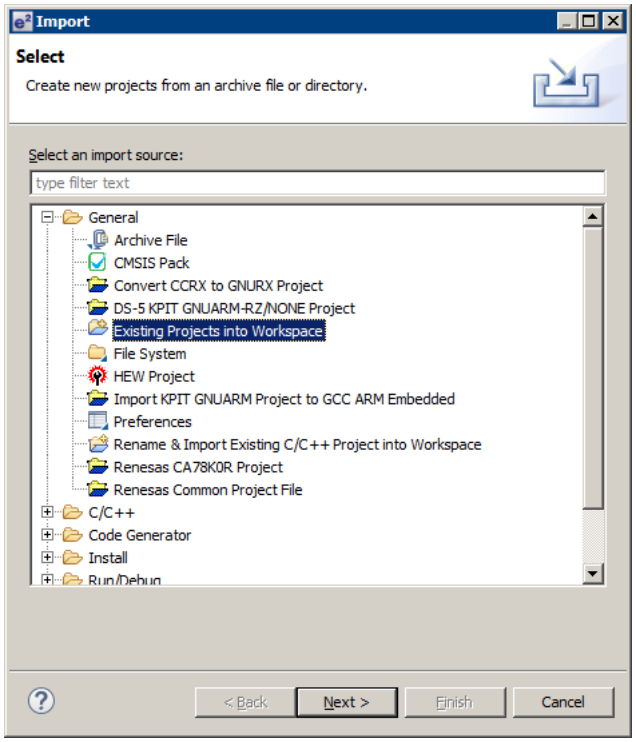
Start Renesas e2studio and in the Workspace Launcher click `Browse…` to select the workspace. If the Workspace Launcher is not shown on startup, select it by menu `File -> Switch Workspace`.

Select the workspace directory `c:\workspace` or any other folder of your choice:

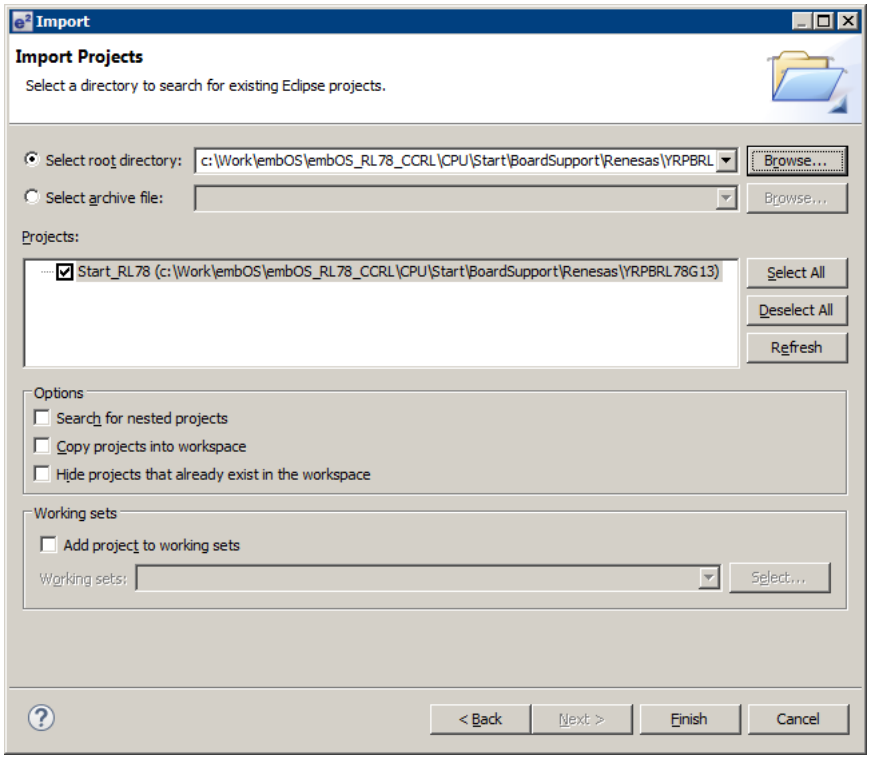


The workspace will then be created in the selected folder.

Now import the sample start project from one board support folder. Choose menu `File -> Import` and in the `Import` dialog select `General -> Existing Projects into Workspace`.

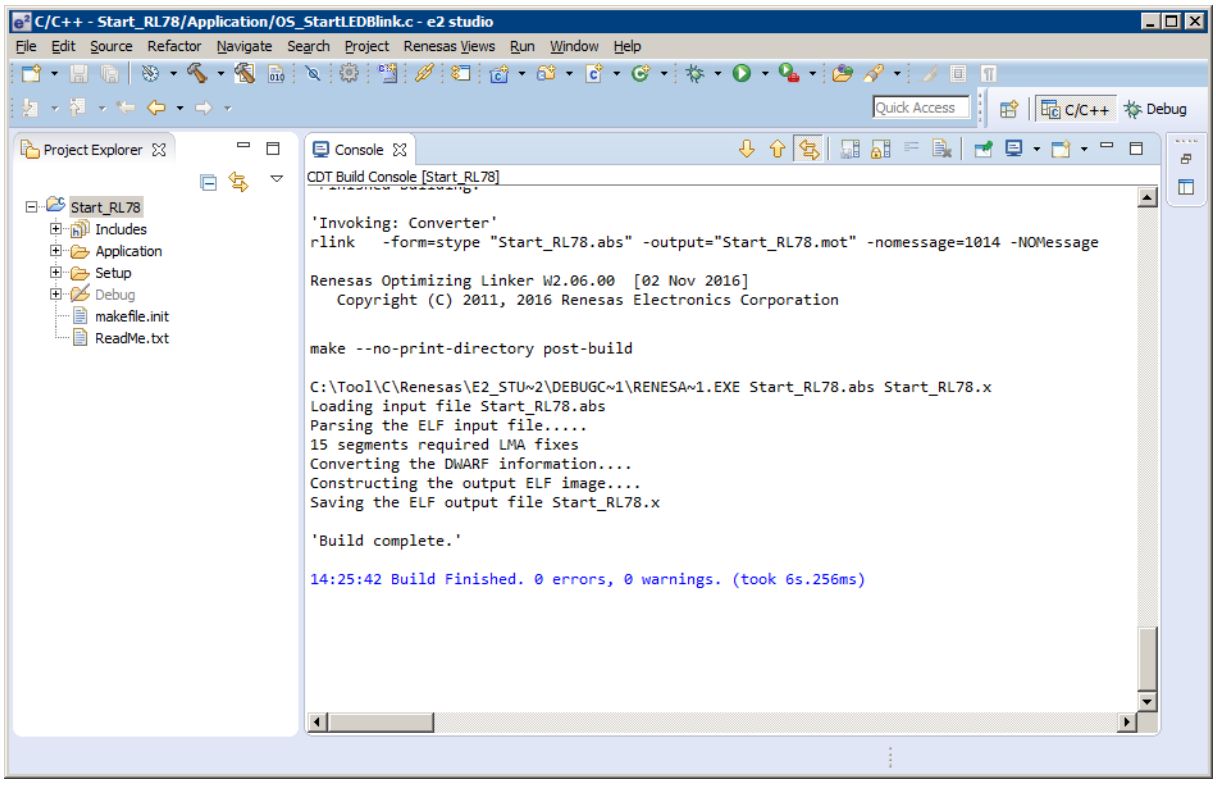


Press `Next`, the `Import Projects` dialog shows up.
Press `Browse…` and select `Start\BoardSupport\` or any project subfolder as the root directory for the project to import:



Do **not** select the `Copy projects into workspace` option.

Refresh the project and build it:



For latest information you may read the `ReadMe.txt` file which is part of every start project.

1.3 First Steps

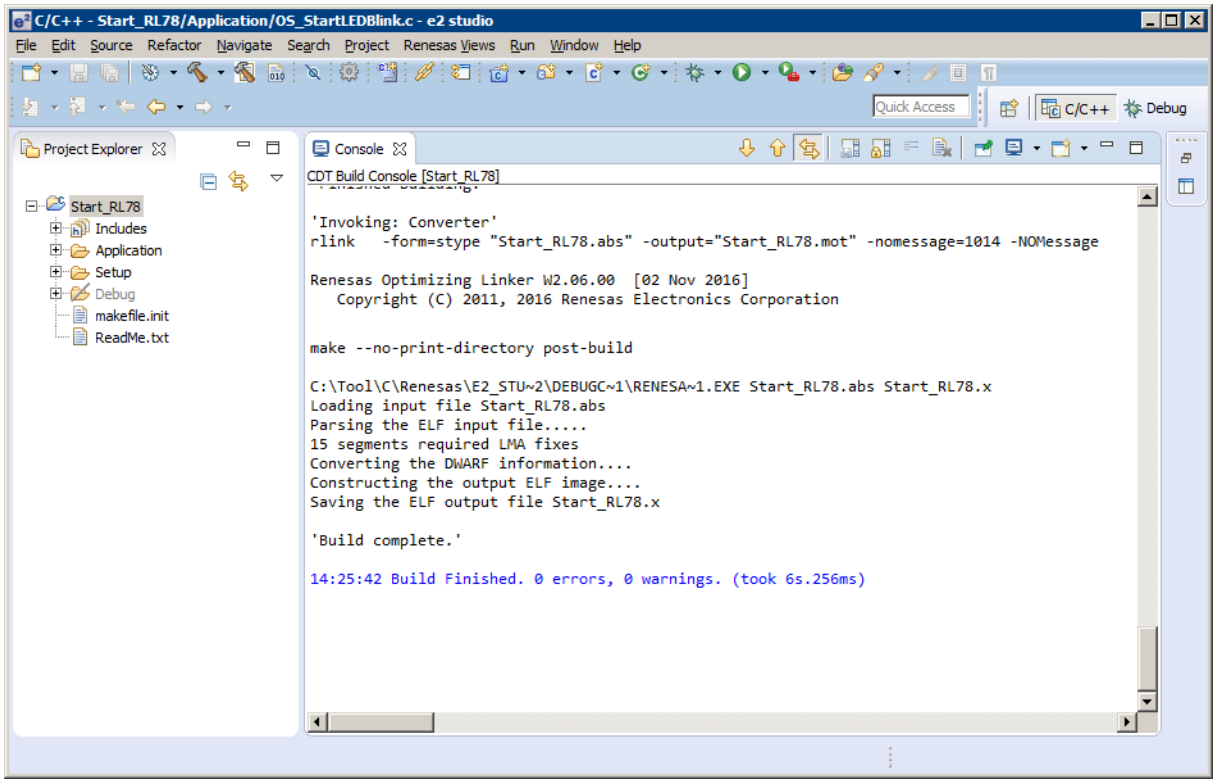
After installation of embOS you can create your first multitasking application. You have received several ready to go sample start workspaces and projects and every other files needed in the subfolder `Start`. It is a good idea to use one of them as a starting point for all of your applications. The 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\<DeviceManufactor>\<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.

1.4 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 & Co. KG                *
*                         The Embedded Experts                        *
**********************************************************************
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).
--------- END-OF-HEADER ------------------------------------------------
*/

#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_Delay (50);
  }
}

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

/*********************************************************************
*
*       main()
*/
int main(void) {
  OS_InitKern();                  /* Initialize OS                */
  OS_InitHW();                    /* Initialize Hardware for OS   */
  BSP_Init();                     /* Initialize LED ports         */
  /* You need to create at least one task before calling OS_Start() */
  OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
  OS_Start();                     /* Start multitasking           */
  return 0;
}

/****** End Of File *************************************************/
```

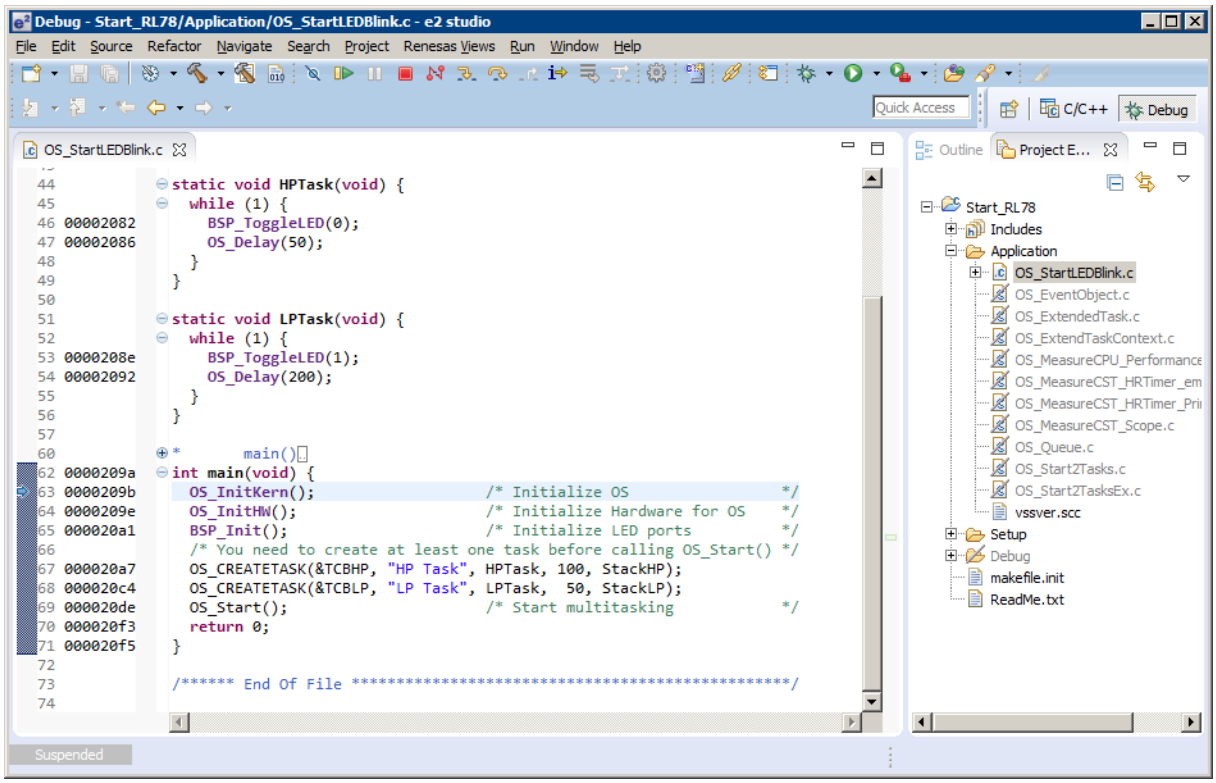
1.5 Stepping through the sample application

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_IncDI()` initially disables interrupts.

`OS_InitKern()` 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. Because of the previous call of `OS_IncDI()`, interrupts are not enabled during execution of `OS_InitKern()`.

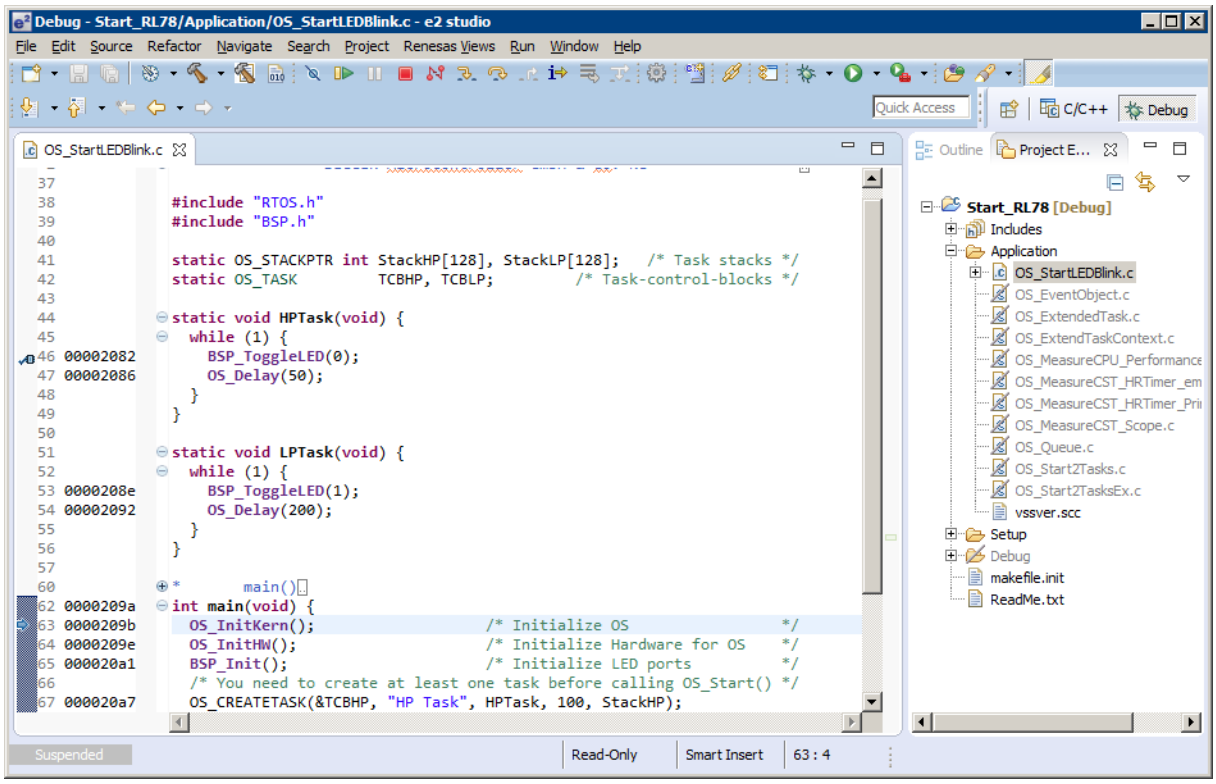
`OS_InitHW()` is part of `RTOSInit.c` and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the system tick interrupt for embOS. Step through it to see what is done.

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



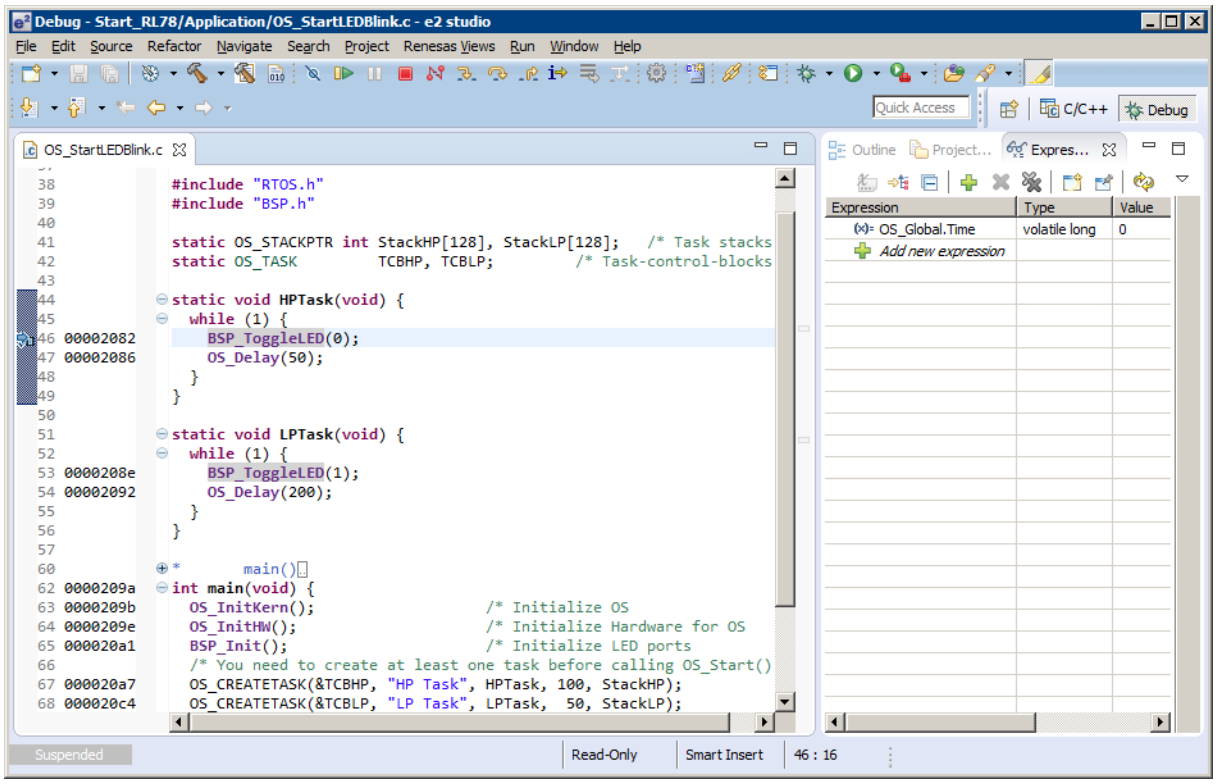

```
44                static void HPTask(void) {
45                  while (1) {
46 00002082         BSP_ToggleLED(0);
47 00002086         OS_Delay(50);
48                  }
49                }
50
51                static void LPTask(void) {
52                  while (1) {
53 0000208e         BSP_ToggleLED(1);
54 00002092         OS_Delay(200);
55                  }
56                }
57
60                *       main()
62 0000209a       int main(void) {
63 0000209b         OS_InitKern();                /* Initialize OS                */
64 0000209e         OS_InitHW();                  /* Initialize Hardware for OS   */
65 000020a1         BSP_Init();                   /* Initialize LED ports         */
66                  /* You need to create at least one task before calling OS_Start() */
67 000020a7         OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
68 000020c4         OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
69 000020de         OS_Start();                   /* Start multitasking           */
70 000020f3         return 0;
71 000020f5       }
72
73                /****** End Of File *************************************************/
74
```


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

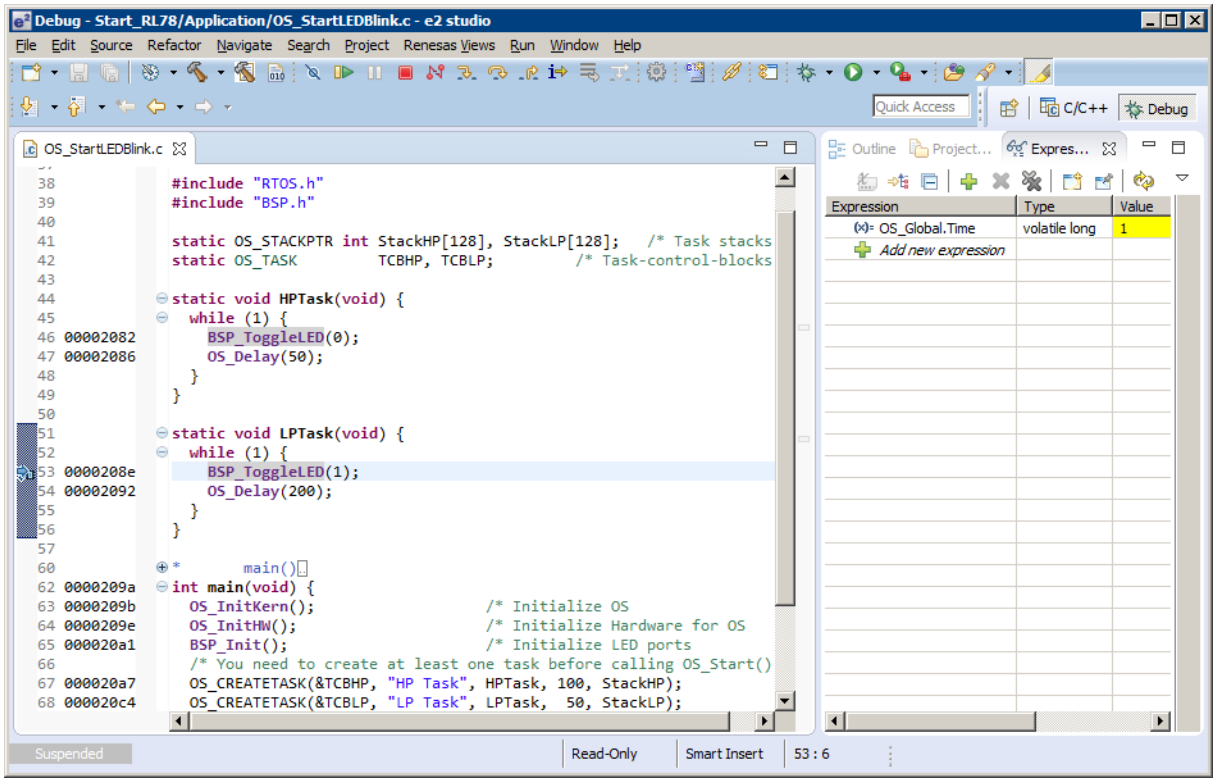


As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only.

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

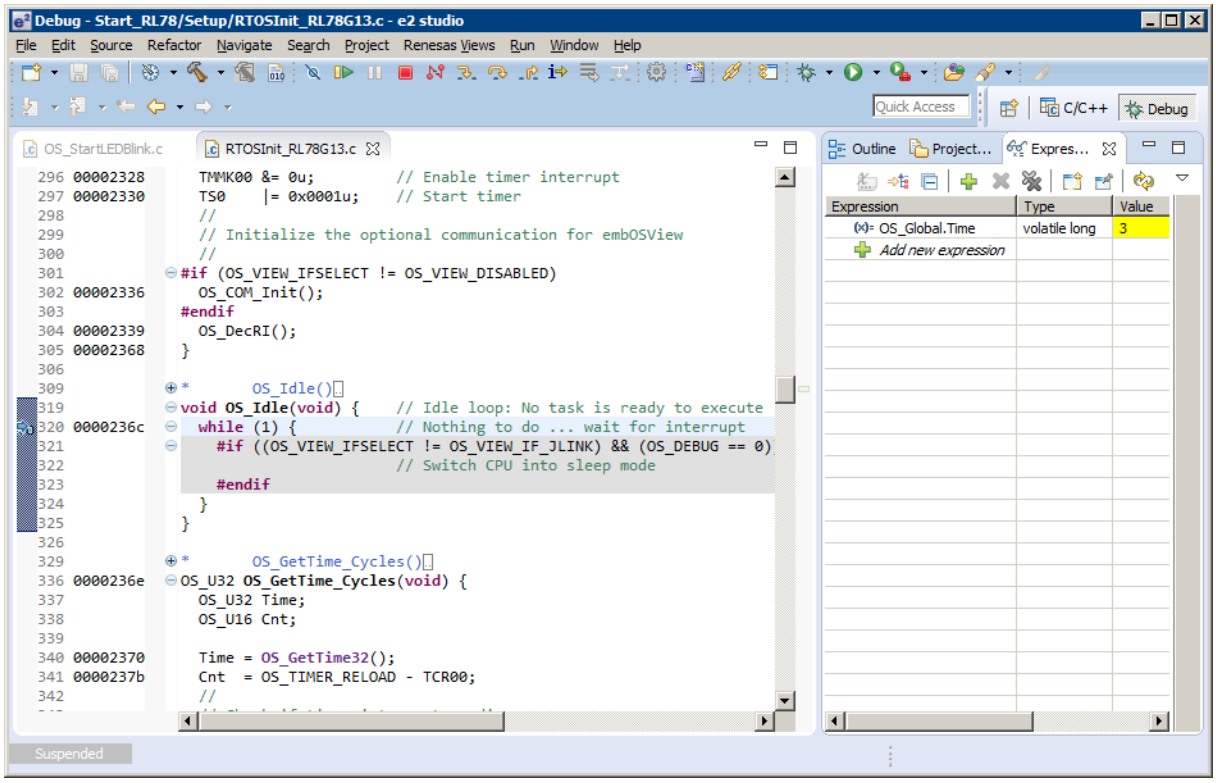


If you continue stepping, you will arrive at the task that has lower priority:



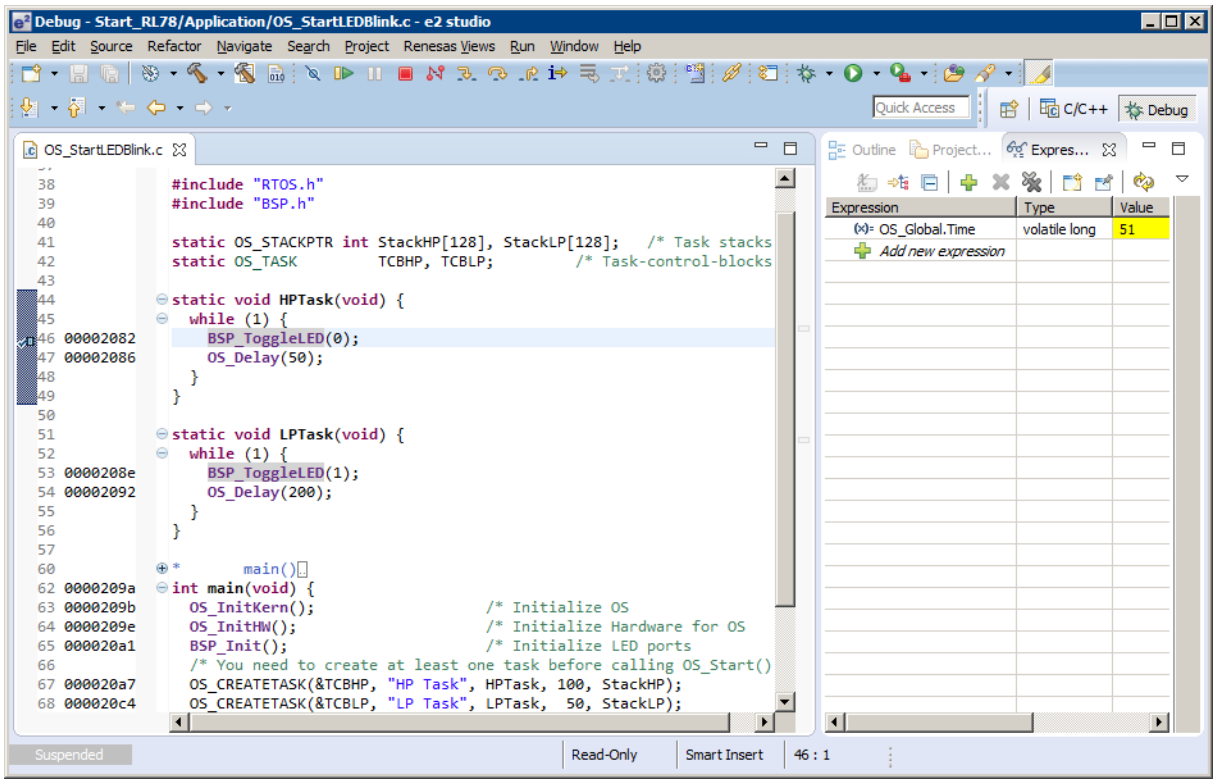
Continue to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

You will arrive there when you step into the `OS_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 50 system tick delay.



Chapter 2

Build your own application

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

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 12 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.
- 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 a call of `OS_InitKern()` and `OS_InitHW()` prior any other embOS functions are called. You should then modify or replace the `OS_StartLEDBlink.c` source file in the subfolder `Application\`.

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.

3.1 Naming conventions for prebuilt libraries

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

The libraries are named as follows:

`osRL78<Memory_Model><Far_ROM><Core>_<LibMode>.lib`

Parameter	Meaning	Values
`Memory_Model`	Specifies the selected memory model	s : small m : medium
`Far_ROM`	Default attribute of ROM data	f : Sets near/far attribute for ROM data to far n : Sets near/far attribute for ROM data to near
`Core`	CPU core variant	2 : RL78 core without instructions to support a hardware multiplier/divider (S2) 3 : RL78 core with instructions to support a hardware multiplier/divider (S3)
`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

`osRL78sf2_dp.lib` is the library for a project using small memory model and far ROM data for the `RL78_2` core variant with debug and profiling support.

3.2 List of available libraries

Memory model	Far ROM	Core	Library types	Library
small	near	RL78_2	Extreme release	osRL78sn2_xr.lib
small	near	RL78_2	Release	osRL78sn2_r.lib
small	near	RL78_2	Stack-check	osRL78sn2_s.lib
small	near	RL78_2	Stack-check + Profiling	osRL78sn2_sp.lib
small	near	RL78_2	Debug	osRL78sn2_d.lib
small	near	RL78_2	Debug + Profiling	osRL78sn2_dp.lib
small	near	RL78_2	Debug + Profiling + Trace	osRL78sn2_dt.lib
small	far	RL78_2	Extreme release	osRL78sf2_xr.lib
small	far	RL78_2	Release	osRL78sf2_r.lib
small	far	RL78_2	Stack-check	osRL78sf2_s.lib
small	far	RL78_2	Stack-check + Profiling	osRL78sf2_sp.lib
small	far	RL78_2	Debug	osRL78sf2_d.lib
small	far	RL78_2	Debug + Profiling	osRL78sf2_dp.lib
small	far	RL78_2	Debug + Profiling + Trace	osRL78sf2_dt.lib

Memory model	Far ROM	Core	Library types	Library
medium	near	RL78_2	Extreme release	osRL78mn2_xr.lib
medium	near	RL78_2	Release	osRL78mn2_r.lib
medium	near	RL78_2	Stack-check	osRL78mn2_s.lib
medium	near	RL78_2	Stack-check + Profiling	osRL78mn2_sp.lib
medium	near	RL78_2	Debug	osRL78mn2_d.lib
medium	near	RL78_2	Debug + Profiling	osRL78mn2_dp.lib
medium	near	RL78_2	Debug + Profiling + Trace	osRL78mn2_dt.lib
medium	far	RL78_2	Extreme release	osRL78mf2_xr.lib
medium	far	RL78_2	Release	osRL78mf2_r.lib
medium	far	RL78_2	Stack-check	osRL78mf2_s.lib
medium	far	RL78_2	Stack-check + Profiling	osRL78mf2_sp.lib
medium	far	RL78_2	Debug	osRL78mf2_d.lib
medium	far	RL78_2	Debug + Profiling	osRL78mf2_dp.lib
medium	far	RL78_2	Debug + Profiling + Trace	osRL78mf2_dt.lib
small	near	RL78_3	Extreme release	osRL78sn3_xr.lib
small	near	RL78_3	Release	osRL78sn3_r.lib
small	near	RL78_3	Stack-check	osRL78sn3_s.lib
small	near	RL78_3	Stack-check + Profiling	osRL78sn3_sp.lib
small	near	RL78_3	Debug	osRL78sn3_d.lib
small	near	RL78_3	Debug + Profiling	osRL78sn3_dp.lib
small	near	RL78_3	Debug + Profiling + Trace	osRL78sn3_dt.lib
small	far	RL78_3	Extreme release	osRL78sf3_xr.lib
small	far	RL78_3	Release	osRL78sf3_r.lib
small	far	RL78_3	Stack-check	osRL78sf3_s.lib
small	far	RL78_3	Stack-check + Profiling	osRL78sf3_sp.lib
small	far	RL78_3	Debug	osRL78sf3_d.lib
small	far	RL78_3	Debug + Profiling	osRL78sf3_dp.lib
small	far	RL78_3	Debug + Profiling + Trace	osRL78sf3_dt.lib
medium	near	RL78_3	Extreme release	osRL78mn3_xr.lib
medium	near	RL78_3	Release	osRL78mn3_r.lib
medium	near	RL78_3	Stack-check	osRL78mn3_s.lib
medium	near	RL78_3	Stack-check + Profiling	osRL78mn3_sp.lib
medium	near	RL78_3	Debug	osRL78mn3_d.lib
medium	near	RL78_3	Debug + Profiling	osRL78mn3_dp.lib
medium	near	RL78_3	Debug + Profiling + Trace	osRL78mn3_dt.lib
medium	far	RL78_3	Extreme release	osRL78mf3_xr.lib
medium	far	RL78_3	Release	osRL78mf3_r.lib
medium	far	RL78_3	Stack-check	osRL78mf3_s.lib
medium	far	RL78_3	Stack-check + Profiling	osRL78mf3_sp.lib
medium	far	RL78_3	Debug	osRL78mf3_d.lib
medium	far	RL78_3	Debug + Profiling	osRL78mf3_dp.lib
medium	far	RL78_3	Debug + Profiling + Trace	osRL78mf3_dt.lib

Chapter 4

CPU and compiler specifics

4.1 CPU modes

embOS for Renesas RL78 supports all memory models that the CCRL compiler supports. For the RL78 CPUs, there are two memory models and two ROM data models, which results in four different combinations for the memory model options.

The CCRL compiler offers two memory models:

Memory Model	Default memory attribute	Code location
small	__near	0x000000 to 0x00FFFF
medium	__far	0x000000 to 0x0FFFFF

The CCRL compiler offers two ROM data models:

ROM Data Model	Default memory attribute	Data placement
near	__near	0x0F0000 to 0x0FFFFF
far	__far	0x000000 to 0x0FFFFF

4.2 Core options

The CCRL compiler supports two different core variants which are also supported by embOS.

Chapter 5

Interrupts

5.1 What happens when an interrupt occurs?

- The CPU-core receives an interrupt request from the interrupt controller.
- As soon as the interrupts are enabled, the interrupt is accepted and executed.
- The corresponding interrupt service routine (ISR) is started.
- The first thing you should do in the ISR is to call `OS_EnterInterrupt()` or `OS_EnterNestableInterrupt()`. These functions tell embOS, that you are executing an ISR. In case of calling `OS_EnterNestableInterrupt()` embOS will reenable interrupts again to allow nesting.
- The ISR stores all registers which are modified by the ISR on the current stack. Current stack is either a task stack or the system stack.
- If you are using `OS_EnterIntStack()` in the ISR, it will switch the stack pointer to the system stack. Please be aware, that a function calling `OS_EnterIntStack()` is not allowed to have local variables.
- If you used `OS_EnterIntStack()` at the beginning of your ISR, you must call `OS_LeaveIntStack()` at the end of this function. The stack pointer will be restored to its original value.
- Depending on which function you have called at the beginning of your ISR, you must call `OS_LeaveInterrupt()` or `OS_LeaveNestableInterrupt()` and the ISR will return from interrupt. If the ISR caused a task switch, it will take place immediately when leaving the ISR.

5.2 Defining interrupt handlers in C

The definition of an interrupt function using embOS calls is very much the same as for a normal interrupt service routine (ISR). If your ISR will use embOS system calls, or if you enable interrupts again in your ISR, you will have to call `OS_EnterInterrupt()` or `OS_EnterNestableInterrupt()` at the start and `OS_LeaveInterrupt()` or `OS_LeaveNestableInterrupt()` at the end of your ISR. In case you want to execute the ISR on the system stack, you must call `OS_EnterIntStack()` right after `OS_EnterInterrupt()` and `OS_LeaveIntStack()` right before `OS_LeaveInterrupt()`.

Example

Simple interrupt routine:

```
#pragma interrupt OS_ISR_Tick (vect=INTTM00)
static void OS_ISR_Tick (void) {
  OS_EnterNestableInterrupt();
  OS_ENTER_INT_STACK();
  OS_TICK_Handle();
  OS_LEAVE_INT_STACK();
  OS_LeaveNestableInterrupt();
}
```


5.3 Interrupt-stack

The routines `OS_EnterIntStack()` and `OS_LeaveIntStack()` can be used to switch the stack pointer to the system stack during execution of the ISR. If you are not using these routines, the ISR uses the active stacks. The active stack is either a task stack or the system stack.

5.4 Interrupt-stack switching

Since the Renesas RL78 CPUs do not have a separate stack pointer for interrupts, every interrupt runs on the current stack. To reduce the stack load of tasks, embOS offers its own interrupt stack which is located in the system stack. To use the embOS interrupt stack, call `OS_EnterIntStack()` at the beginning of an interrupt handler just after the call of `OS_EnterInterrupt()` and call `OS_LeaveIntStack()` at the end just before `OS_LeaveInterrupt()`.

Please note, that an interrupt handler using interrupt stack switching must not use local variables. It should call a function.

Example

Interrupt-routine using embOS interrupt stack:

```
static void OS_ISR_Rx_Handler(void) {
  int Dummy;
  if (ASIS0 & 0x07) {           /* Check any reception error        */
    Dummy = RXB0;               /* Reset error, discard Byte        */
  } else {
    OS_OnRx(RXB0);              /* Process data                     */
  }
}

#pragma interrupt OS_COM_ISR_RxErr (vect=INTSRE2)
static void OS_COM_ISR_RxErr(void) {
  OS_EnterNestableInterrupt();  /* We will enable interrupts        */
  OS_EnterIntStack();           /* We will use interrupt stack      */
  OS_ISR_Rx_Handler();          /* A call to a handler is required! */
  OS_LeaveIntStack();           /* Interrupt stack switching does   */
  OS_LeaveNestableInterrupt();  /* not allow local variables in ISR */
}
```

5.5 Zero latency interrupts

Instead of disabling interrupts when embOS does atomic operations, the inter- rupt level of the CPU is set per default to 1. Therefore all interrupts with the priorities 0 and 1 can still be processed. Please note, that lower priority numbers define a higher priority. All interrupts with priority levels 0 and 1 are never disabled. These interrupts are named zero latency interrupts.
You must not execute any embOS function from within a zero latency interrupt function.

5.6 OS_SetFastIntPriorityLimit()

The interrupt priority limit for zero latency interrupts is set to 1 by default. This means, all interrupts with priority 0 and 1 will never be disabled by embOS.

Description

`OS_SetFastIntPriorityLimit()` is used to set the interrupt priority limit between zero latency interrupts and lower priority embOS interrupts.

Prototype

```
void OS_SetFastIntPriorityLimit (OS_UINT Priority);
```

Parameters

Parameter	Description
`Priority`	The lowest value useable as priority for zero latency interrupts. All interrupts with higher priority are never disabled by embOS. Valid range: 0 ≤ Priority ≤ 2.

Additional Information

To modify the default priority limit, `OS_SetFastIntPriorityLimit()` should be called before embOS was started.

This table shows which interrupt priority values are valid for a given priority limit.

Priority limit	embOS interrupts	Zero latency interrupts
0	1, 2, 3	0
1 (default)	2, 3	0, 1
2	3	0, 1, 2

Chapter 6

Stacks

6.1 Task stack for Renesas RL78

The stack pointer of the RL78 CPUs is a 16bit register and can therefore point to any near memory location. The stacks for the tasks may be located in any RAM location which can be addressed by the stack pointer. The required amount of stack for a task depends on the embOS library mode, the application and functions called by the task. As long as interrupt stack switching is not used, all interrupts may also run on the task stack. The minimum amount of stack required by embOS to save the task specific registers is about 24 bytes. We recommend at a minimum task stack size of 64 bytes. Using embOSView together with a stack check library may be used to analyze the amount of stack used and needed for every task.

6.2 System and Interrupt stack for Renesas RL78

The main stack is used as system stack. Your application uses this stack before executing `OS_Start()`, during execution of embOS internal functions and during the timer tick routines. Furthermore, software timers use the system stack. If your interrupt service routines perform stack switching by calling `OS_EnterIntStack()`, they will also use the system stack. The stack segment also has to be located in the internal RAM which is addressable by the stack pointer.

Chapter 7

Technical data

This chapter lists technical data of embOS used with RL78 CPUs.

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 1.500 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	14
Software timer	12
Resource semaphore	8
Counting semaphore	4
Mailbox	14
Queue	18
Task event	0
Event object	16

