embOS

Real-Time Operating System

CPU & Compiler specifics for Renesas RL78 using IAR Embedded Workbench

Document: UM01032
Software Version: 5.20.0.0
Revision: 0
Date: April 3, 2025

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: April 3, 2025

Software	Revision	Date	By	Description
5.20.0.0	0	250403	MC	New software version.
5.18.1.0	0	230405	MC	Chapter “Libraries” updated.
5.18.0.0	0	230308	TS/MC	New software version.
5.16.0.0	0	220107	TS	New software version.
5.10.2.1	0	200922	TS	New software version.
5.10.2.0	0	200909	MC	New software version.
5.02	0	180702	TS	New software version.
4.40	0	180105	MC	New software version.
4.36	0	170728	MC	New software version.
4.34	0	170329	TS	New software version.
4.16	0	160308	TS	New software version.
4.14a	0	160115	TS	New software version.
4.14	0	151130	TS	New software version.
4.10b	0	150609	TS	Chapter “Interrupts” updated.
4.04a	0	150303	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.
`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 10.

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.

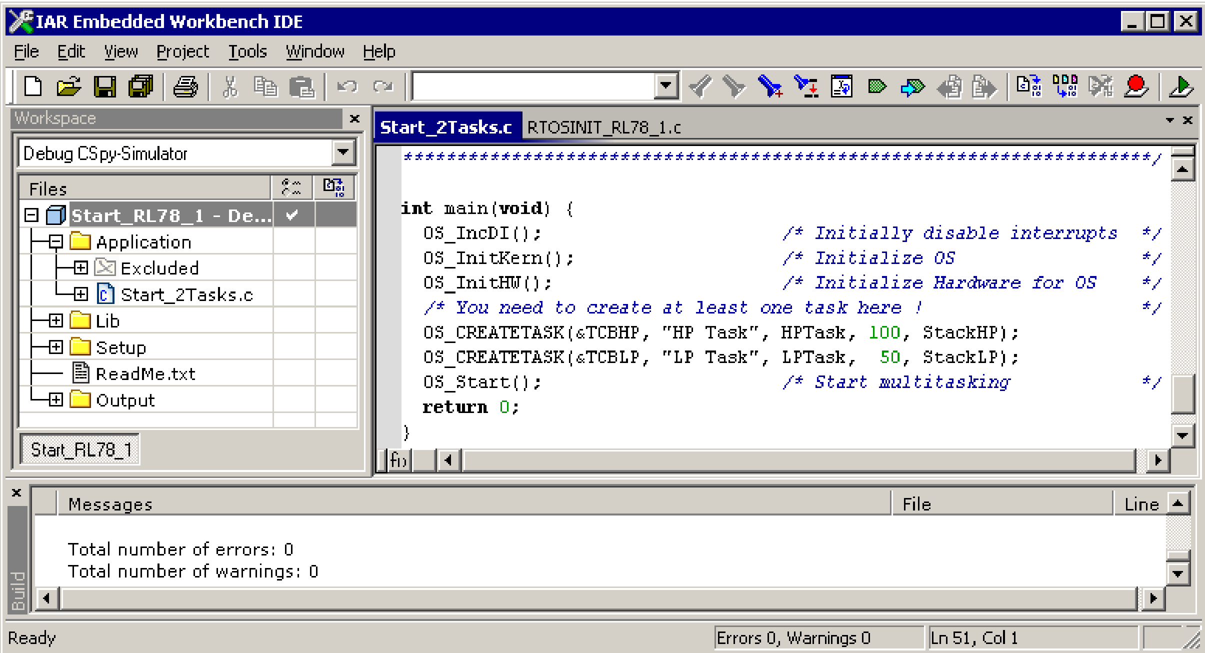
1.2 First Steps

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 ****************************/
```

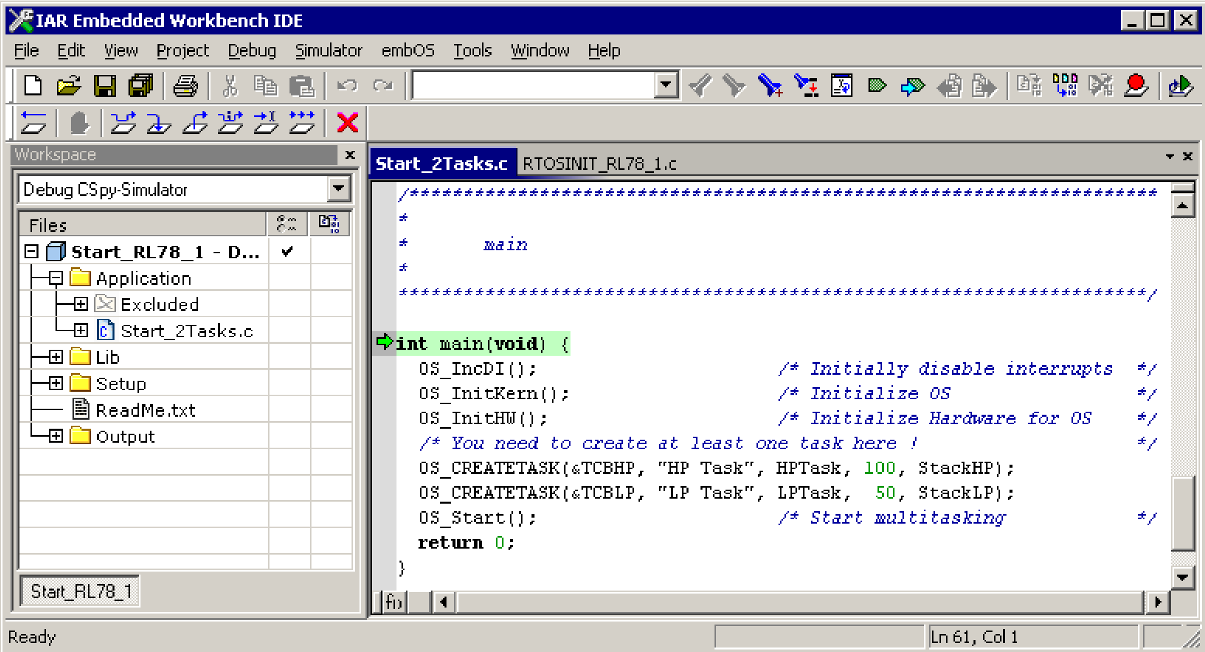

1.4 Stepping through the sample application

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.

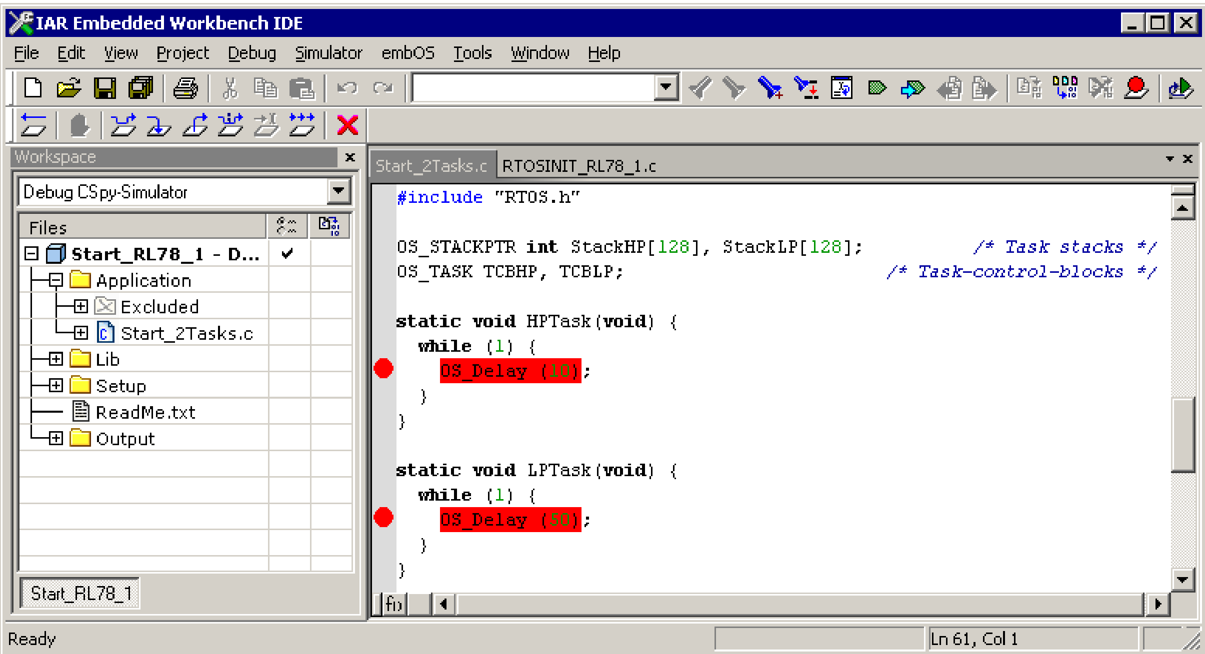
`OS_Init()` is part of the embOS library and written in assembler; you can therefore only step into it in disassembly mode. It initializes the relevant OS variables.

`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.

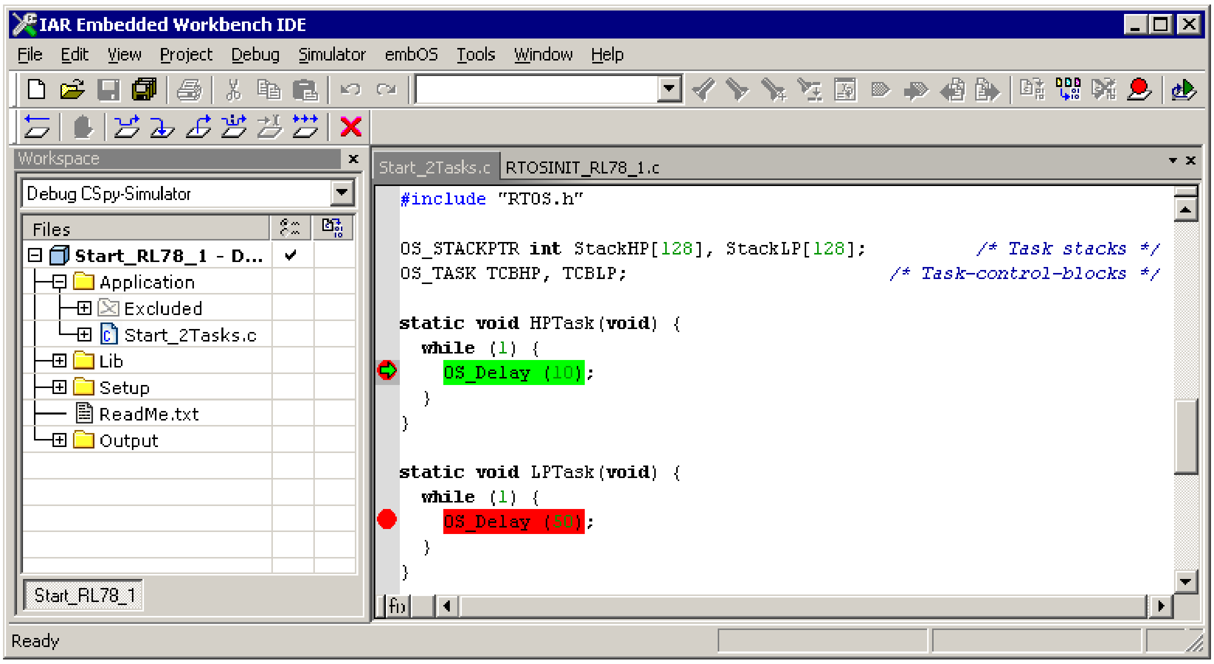


Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.

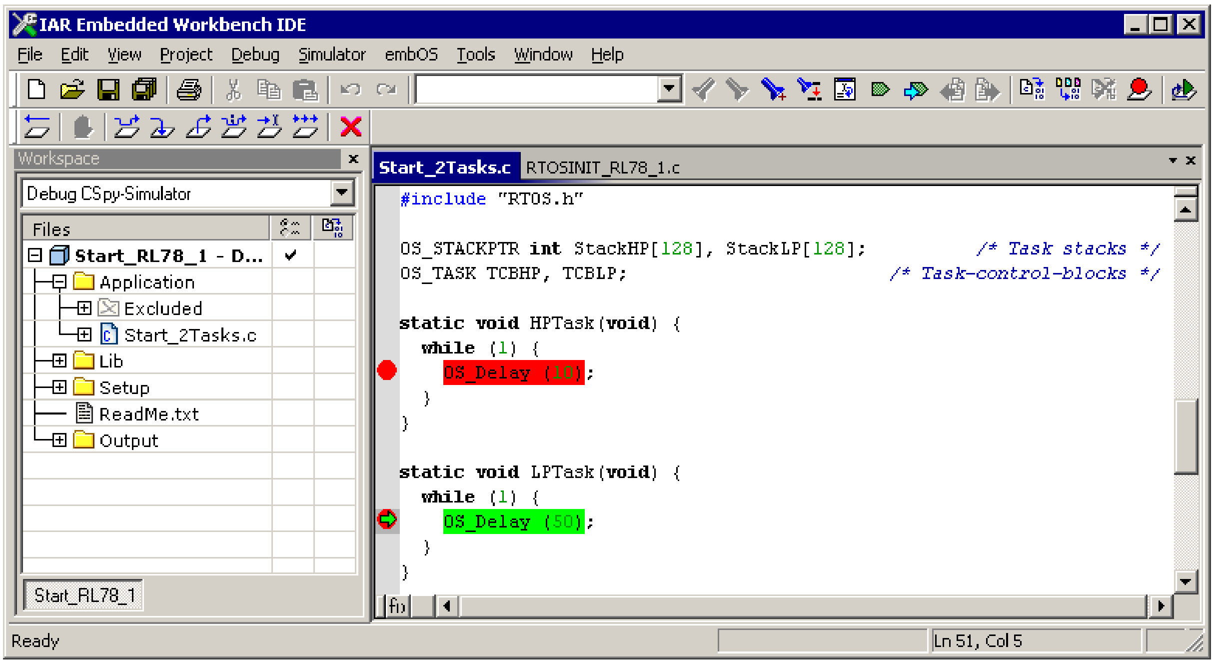


As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only.

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

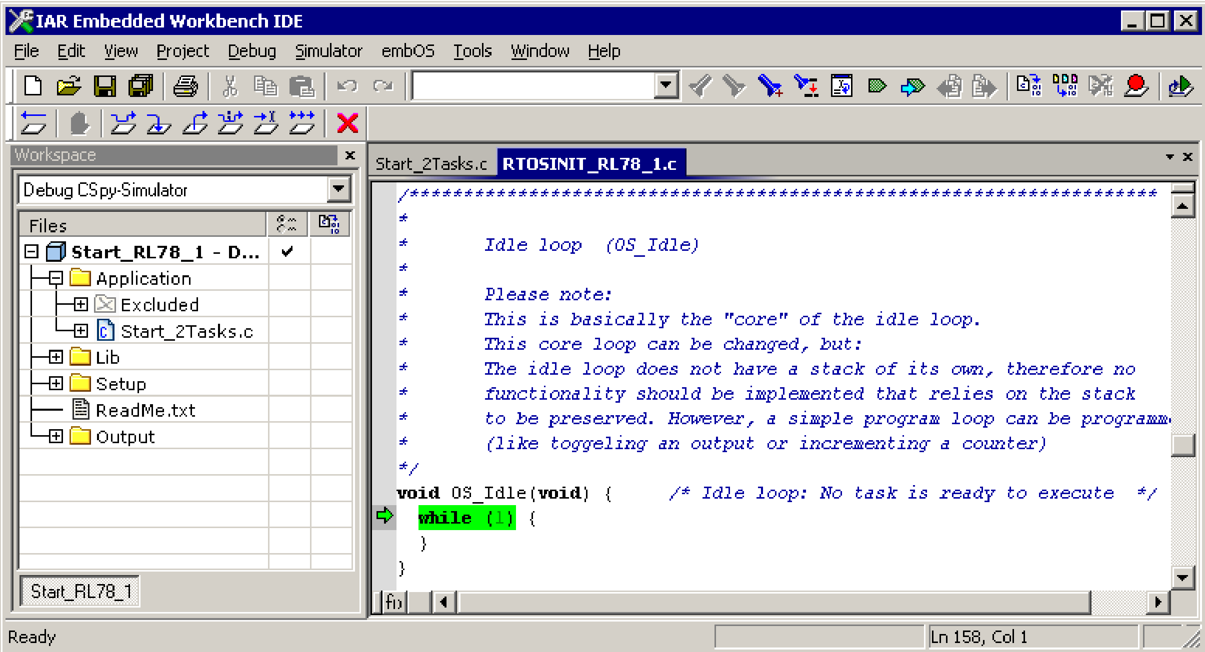


If you continue stepping, you will arrive at the task that has lower priority:



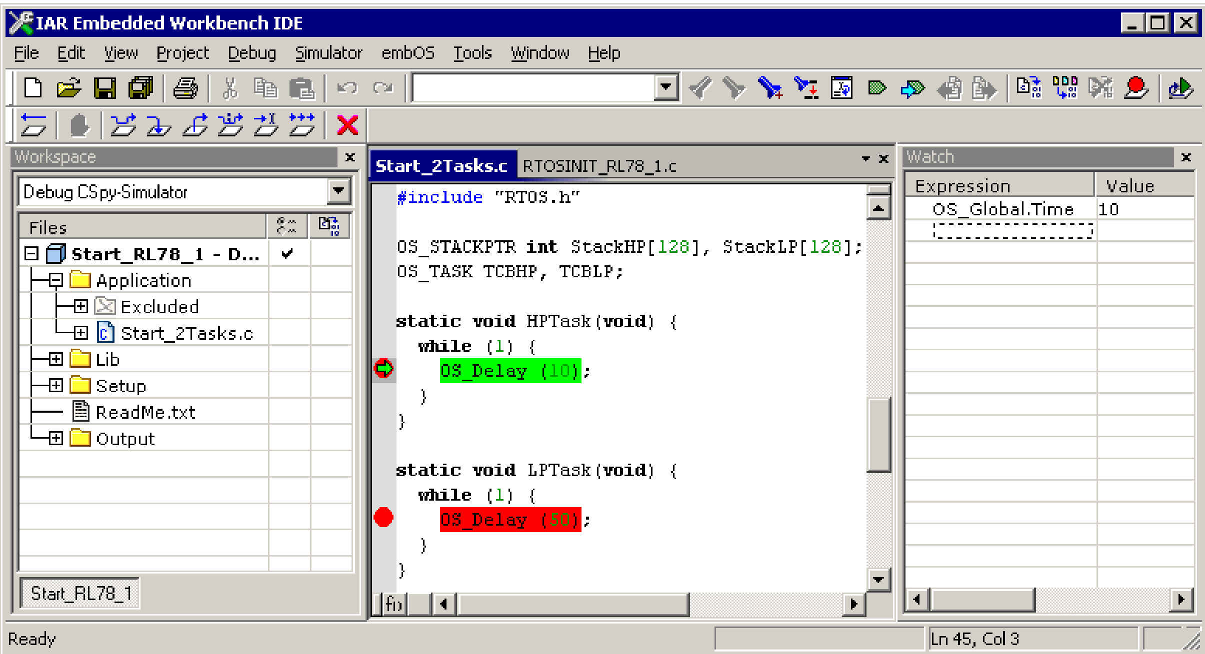
Continue to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

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 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 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 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<code_model><data_model><const_location><core>_<libmode>.a`

Parameter	Meaning	Values
code_model	Specifies the selected code model	n: near f : far
data_model	Specifies the selected data model	n: near data model f : far data model
const_location	Specifies the location of constants	0: constants are located in ROM0 1: constants are located in ROM1 r : constants are located in RAM
core	CPU core variant	1: RL78 core without instructions to support a hardware multiplier/ divider (S2) 2: 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

`osRL78nn1_SP.a` is the library for a project using near code model and near data model for the `RL78_1` core variant with 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 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.

4.2.1 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. 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.

To use the automatic thread-safe locking functions the function `OS_INIT_SYS_LOCKS()` must be called.

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

For more information on IAR's multithread support, please refer to the IAR Embedded Workbench manuals.

4.3 CPU modes

embOS for RENESAS RL78 supports all memory models that the IAR C/C++ Compiler supports. For the RL78 CPUs, there are two code memory models and two data models which results in four different combinations for the memory model options.

The IAR compiler offers two code models:

Code Model	Default memory attribute	Code location
near	`__near_func`	`0x000000` to `0x00FFFF`
far	`__far_func`	`0x000000` to `0xFFFFFF`

The IAR compiler offers two data models:

Data Model	Default memory attribute	Data placement
near	`__near`	The highest 64KB of memory.
far	`__far`	The entire 1MB memory space.

Chapter 5

Stacks

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.

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 least a minimum task stack size of 64 bytes. Using the embOS IAR plugin or embOSView together with a stack check library may be used to analyze the amount of stack used and needed for every task.

5.2 System and Interrupt stack

The IAR `CSTACK` 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. Also software timers use the system stack. If your interrupt service routines perform stack switching by calling `OS_INT_EnterIntStack()`, they will also use the system stack.

The `CSTACK` segment also has to be located in the internal RAM which is addressable by the stack pointer.

Chapter 6

Interrupts

6.1 What happens when an interrupt occurs?

- The CPU-core receives an interrupt request.
- As soon as the interrupts are enabled, the interrupt is executed.
- The corresponding interrupt service routine (ISR) is started.
- The first thing you should do in the ISR, is to call `OS_INT_Enter()` or `OS_INT_EnterNestable()`. These functions tell embOS, that you are executing an ISR. In case of calling `OS_INT_EnterNestable()` 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 your are using `OS_INT_EnterIntStack()` in the ISR, it will switch the stack pointer to the system stack. Please be aware, that a function calling `OS_INT_EnterIntStack()` is not allowed to have local variables.
- If you used `OS_INT_EnterIntStack()` at the beginning of your ISR, you have to call `OS_INT_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 will have to call `OS_INT_Leave()` or `OS_INT_LeaveNestable()` and the ISR will return from interrupt. If the ISR caused a task switch, it will take place immediately when leaving the ISR.

6.2 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_INT_Enter()` or `OS_INT_EnterNestable()` at the start and `OS_INT_Leave()` or `OS_INT_LeaveNestable()` at the end of your ISR. In case you want to execute the ISR on the system stack, you will have to call `OS_INT_EnterIntStack()` right after e.g. `OS_INT_Enter()` and `OS_INT_LeaveIntStack()` right before e.g. `OS_INT_Leave()`.

Example

Simple interrupt routine:

```
#pragma vector= INTTM00_vect
__interrupt void OS_ISR_Tick(void) {
  OS_INT_EnterNestable();
  OS_INT_EnterIntStack();
  OS_HandleTick();
  OS_INT_LeaveIntStack();
  OS_INT_LeaveNestable();
}
```

6.3 Interrupt stack

The routines `OS_INT_EnterIntStack()` and `OS_INT_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.

6.4 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_INT_EnterIntStack()` at the beginning of an interrupt handler just after the call of `OS_INT_Enter()` and call `OS_INT_LeaveIntStack()` at the end just before `OS_INT_Leave()`.

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

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_COM_OnRx(RXB0);      // Process data
  }
}
__interrupt [INTSR0_vect] void OS_ISR_rx(void) {
  OS_INT_EnterNestable();  // We will enable interrupts
  OS_INT_EnterIntStack();  // We will use interrupt stack
  OS_ISR_Rx_Handler();     // A call to a handler is required!
  OS_INT_LeaveIntStack();  // Interrupt stack switching does
  OS_INT_LeaveNestable();  // not allow local variables in ISR
}
```

6.5 Zero latency interrupts with RL78

Instead of disabling interrupts when embOS does atomic operations, the interrupt 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.

6.6 OS_INT_SetPriorityThreshold()

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

Prototype

```
void OS_INT_SetPriorityThreshold(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: $0 \leq$ Priority ≤ 2

Return value

None.

Additional information

To modify the default priority limit, `OS_INT_SetPriorityThreshold()` 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 7

Technical data

7.1 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	~94 bytes
Task control block	RAM	14 bytes
Software timer	RAM	10 bytes
Task event	RAM	0 bytes
Event object	RAM	6 bytes
Mutex	RAM	8 bytes
Semaphore	RAM	4 bytes
RWLock	RAM	14 bytes
Mailbox	RAM	14 bytes
Queue	RAM	16 bytes
Watchdog	RAM	6 bytes
Fixed Block Size Memory Pool	RAM	16 bytes