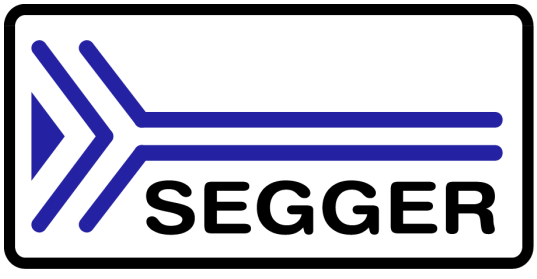
embOS

Real-Time
Operating System

CPU & Compiler
specifics for Renesas
R32C using IAR com-
piler for R32C

Document: UM01046
Software version 3.88e
Revision: 0
Date: September 6, 2013



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: http://www.segger.com

Manual versions

This manual describes the current software version. If any error occurs, inform us and we will try to assist you as soon as possible.
Contact us for further information on topics or routines not yet specified.

Print date: September 6, 2013

Software	Revision	Date	By	Description
3.88e	0	130906	TS	First 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 programm 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 2.1: Typographic conventions

SEGGER Microcontroller GmbH & Co. KG develops and distributes software development tools and ANSI C software components (middleware) for embedded systems in several industries such as telecom, medical technology, consumer electronics, automotive industry and industrial automation.

SEGGER's intention is to cut software development time for embedded applications by offering compact flexible and easy to use middleware, allowing developers to concentrate on their application.

Our most popular products are emWin, a universal graphic software package for embedded applications, and embOS, a small yet efficient real-time kernel. emWin, written entirely in ANSI C, can easily be used on any CPU and most any display. It is complemented by the available PC tools: Bitmap Converter, Font Converter, Simulator and Viewer. embOS supports most 8/16/32-bit CPUs. Its small memory footprint makes it suitable for single-chip applications.

Apart from its main focus on software tools, SEGGER develops and produces programming tools for flash micro controllers, as well as J-Link, a JTAG emulator to assist in development, debugging and production, which has rapidly become the industry standard for debug access to ARM cores.

Corporate Office:
http://www.segger.com

United States Office:
http://www.segger-us.com

EMBEDDED SOFTWARE (Middleware)

emWin
Graphics software and GUI
emWin is designed to provide an efficient, processor- and display controller-independent graphical user interface (GUI) for any application that operates with a graphical display.

embOS
Real Time Operating System
embOS is an RTOS designed to offer the benefits of a complete multitasking system for hard real time applications with minimal resources.

embOS/IP
TCP/IP stack
embOS/IP a high-performance TCP/IP stack that has been optimized for speed, versatility and a small memory footprint.

emFile
File system
emFile is an embedded file system with FAT12, FAT16 and FAT32 support. Various Device drivers, e.g. for NAND and NOR flashes, SD/MMC and CompactFlash cards, are available.

USB-Stack
USB device/host stack
A USB stack designed to work on any embedded system with a USB controller. Bulk communication and most standard device classes are supported.

SEGGER TOOLS

Flasher
Flash programmer
Flash Programming tool primarily for micro controllers.

J-Link
JTAG emulator for ARM cores
USB driven JTAG interface for ARM cores.

J-Trace
JTAG emulator with trace
USB driven JTAG interface for ARM cores with Trace memory. supporting the ARM ETM (Embedded Trace Macrocell).

J-Link / J-Trace Related Software
Add-on software to be used with SEGGER's industry standard JTAG emulator, this includes flash programming software and flash breakpoints.

Table of Contents

Chapter 4

Using embOS for Renesas R32C

This chapter describes how to start with and use embOS for Renesas R32C cores and the IAR compiler for R32C. You should follow these steps to become familiar with embOS.

4.1 Installation

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

To install it, proceed as follows:

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

Assuming that you are using the IAR Embedded Workbench project manager 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 11.

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

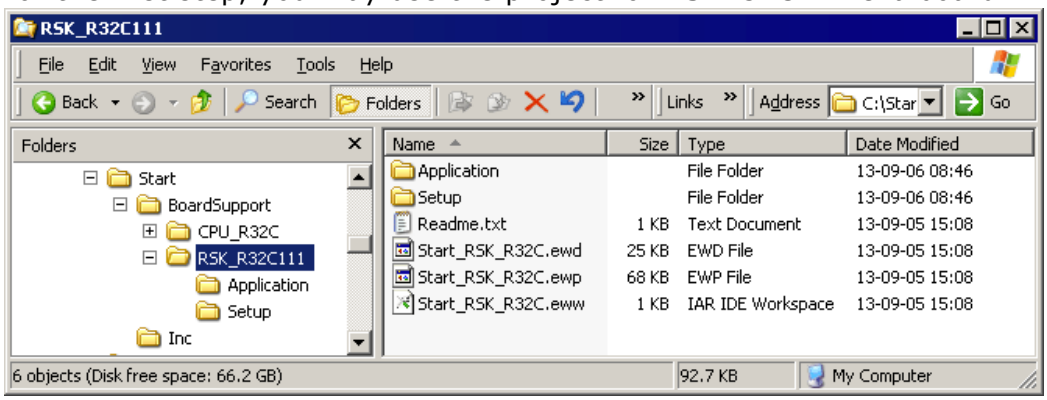
If you will not work with the embedded workbench, you should:
Copy either all or only the library-file that you need to your work-directory. This has the advantage that when you switch to an updated version of embOS later in a project, you do not affect older projects that use embOS also.
embOS does in no way rely on the IAR Embedded Workbench project manager, it may be used without the project manager using batch files or a make utility without any problem.

4.2 First steps

After installation of embOS you can create your first multitasking application. You 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.

For the first step, you may use the project for RSKR32C111 evalboard:

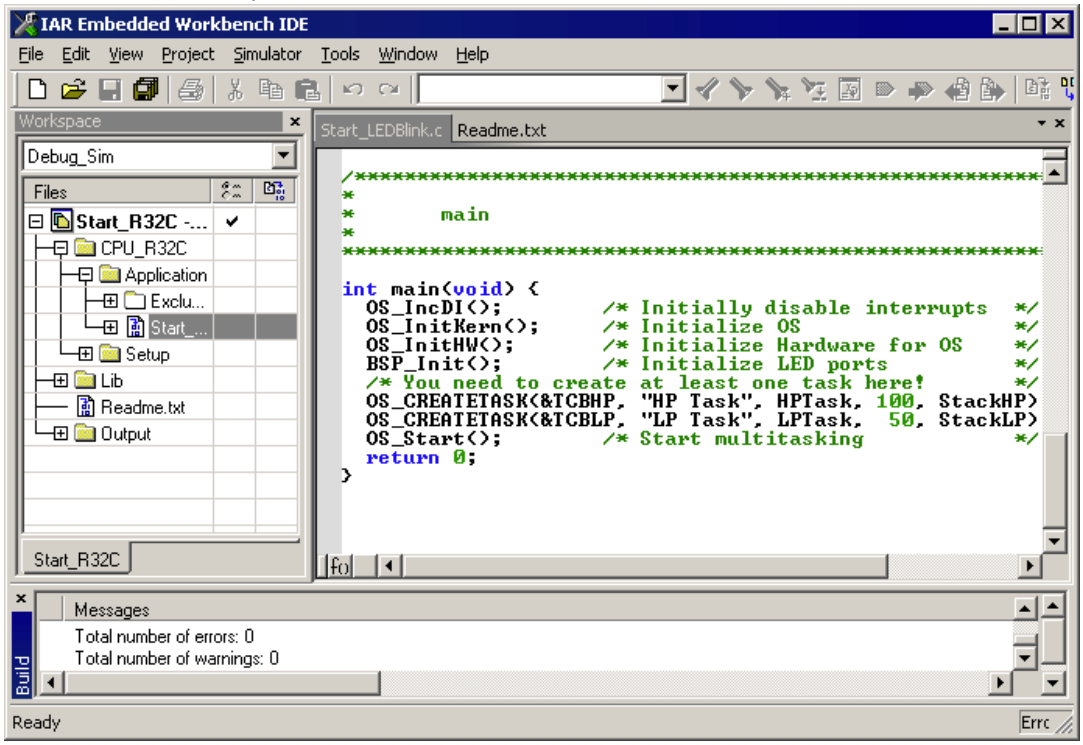


To get your new application running, you should proceed as follows:

- 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 the sample workspace **Start\BoardSupport\CPU_R32C\Start_R32C.eww** with the IAR Embedded Workbench project manager (for example, by double clicking it).
- Build the project. It should be build 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.



4.3 The example application Start_2Tasks.c

The following is a printout of the example application `Start_2Tasks.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 SYSTEME GmbH & Co.KG
* Solutions for real time microcontroller applications
**********************************************************
File    : Start2Tasks.c
Purpose : Skeleton program for embOS
--------- END-OF-HEADER ---------------------------------*/

#include "RTOS.h"

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

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

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

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

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


4.4 Stepping through the sample application

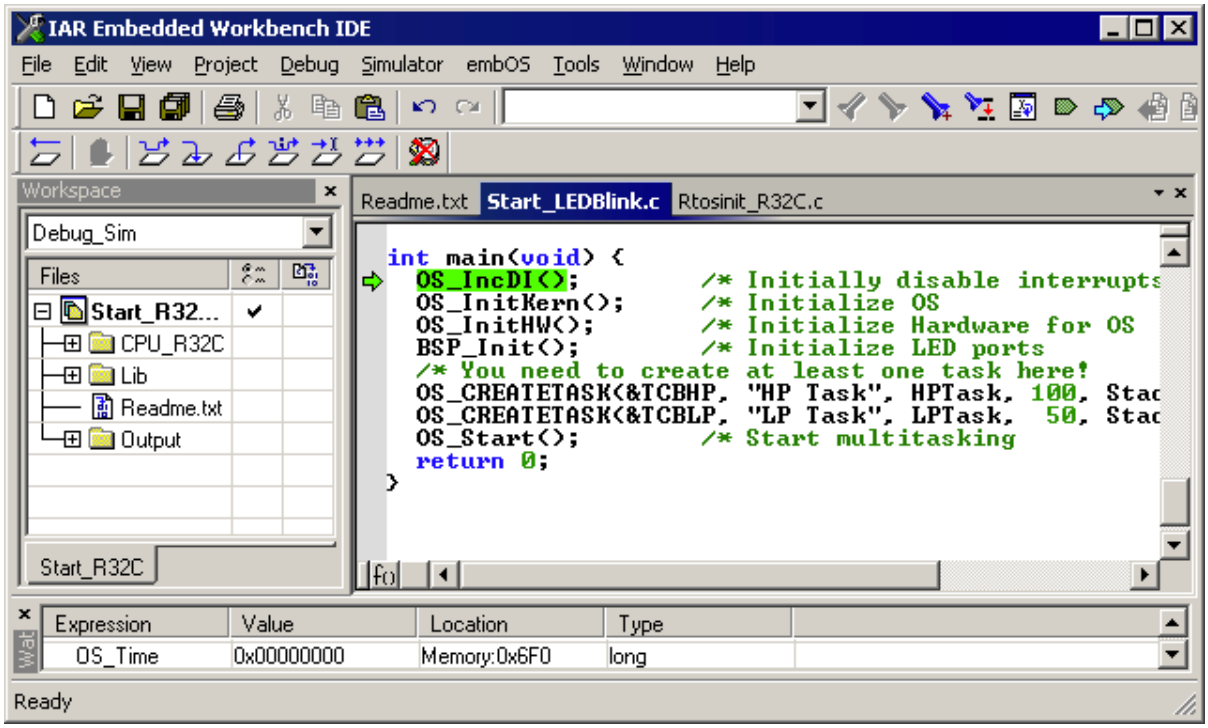
The embOS start project contains configurations which are already setup for the following debugging tools:

- IAR's simulator C-SPY. This configuration is named "Debug-Sim".
- RENESAS's emulator E8A, E30A or newer. This configuration is named "Debug".

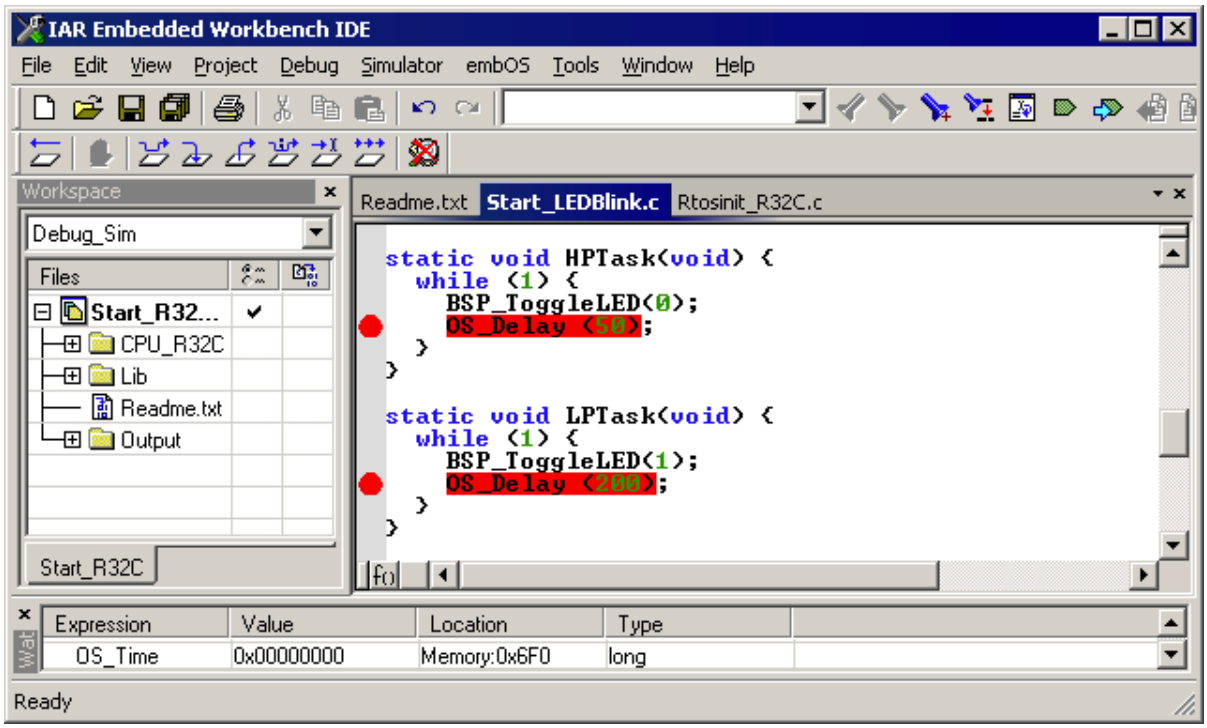
All configurations are prepared to produce the appropriate output files required by the selected debugger or emulator debugger.
The following chapters describe a sample session based on our sample application Start_LEDBlink, using C-SPY Simulator. Using the E8A emulator, the debugging session is very similar.

When starting C-SPY simulator after building the Debug_Sim configuration target, you will see the `main` function (see example screenshot below). The `main` function appears as long as the C-SPY option **Run to main** is selected, which it is by default. Now you can step through the program.

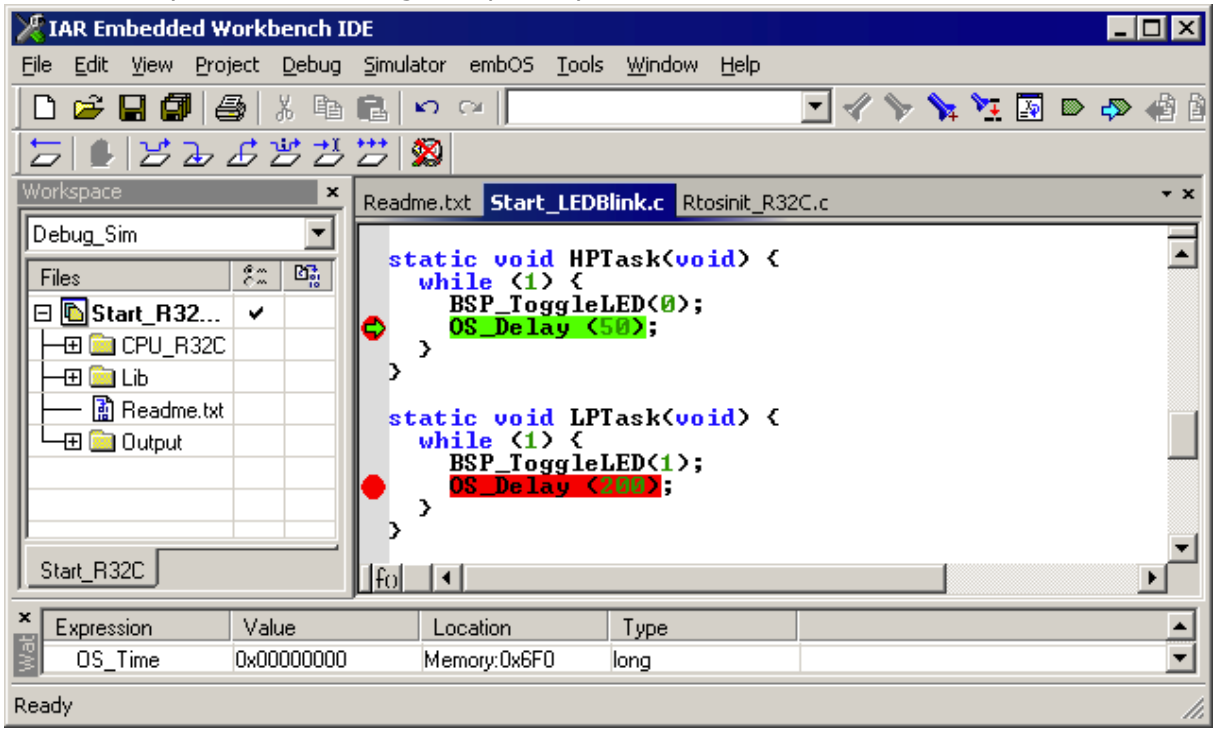
OS_IncDI() initially disables interrupts.
OS_InitKern() is part of the embOS library and written in assembler; you can therefore only step into it in disassembly mode. It initializes the relevant OS variables. Because of the previous call of OS_IncDI(), interrupts are not enabled during execution of 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 timer-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.



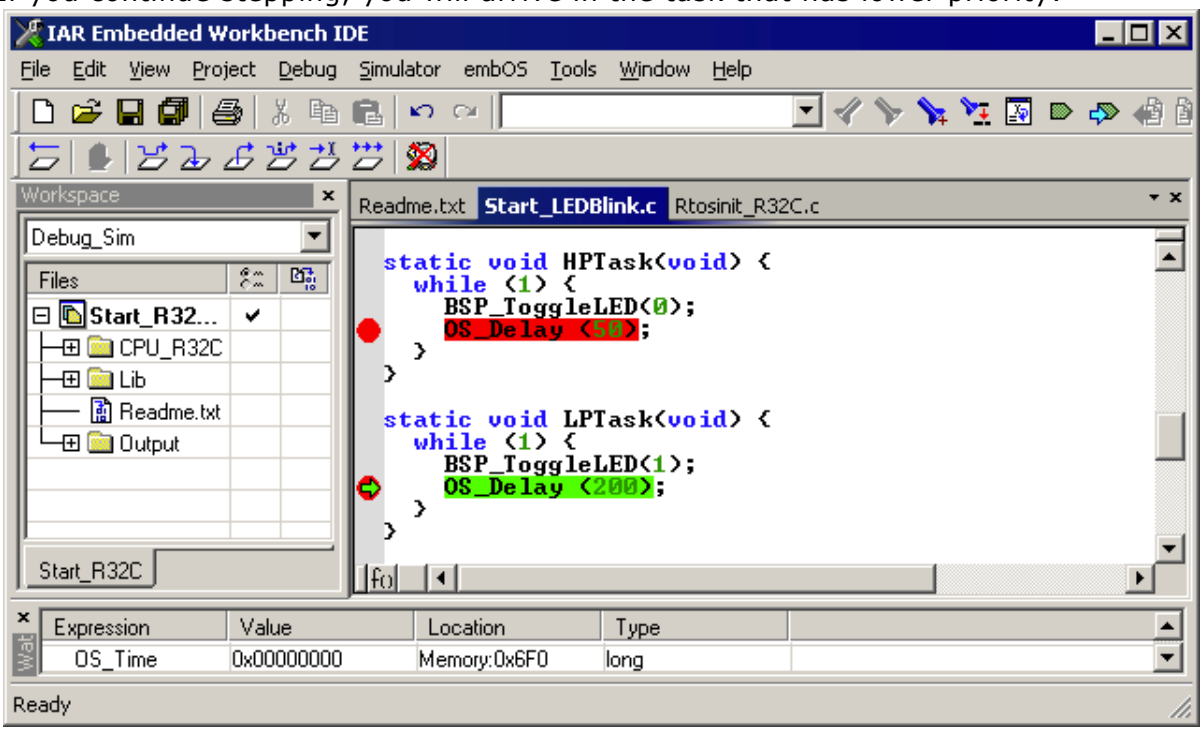
Before you step into OS_Start(), you should set two breakpoints in the two tasks as shown below.



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.

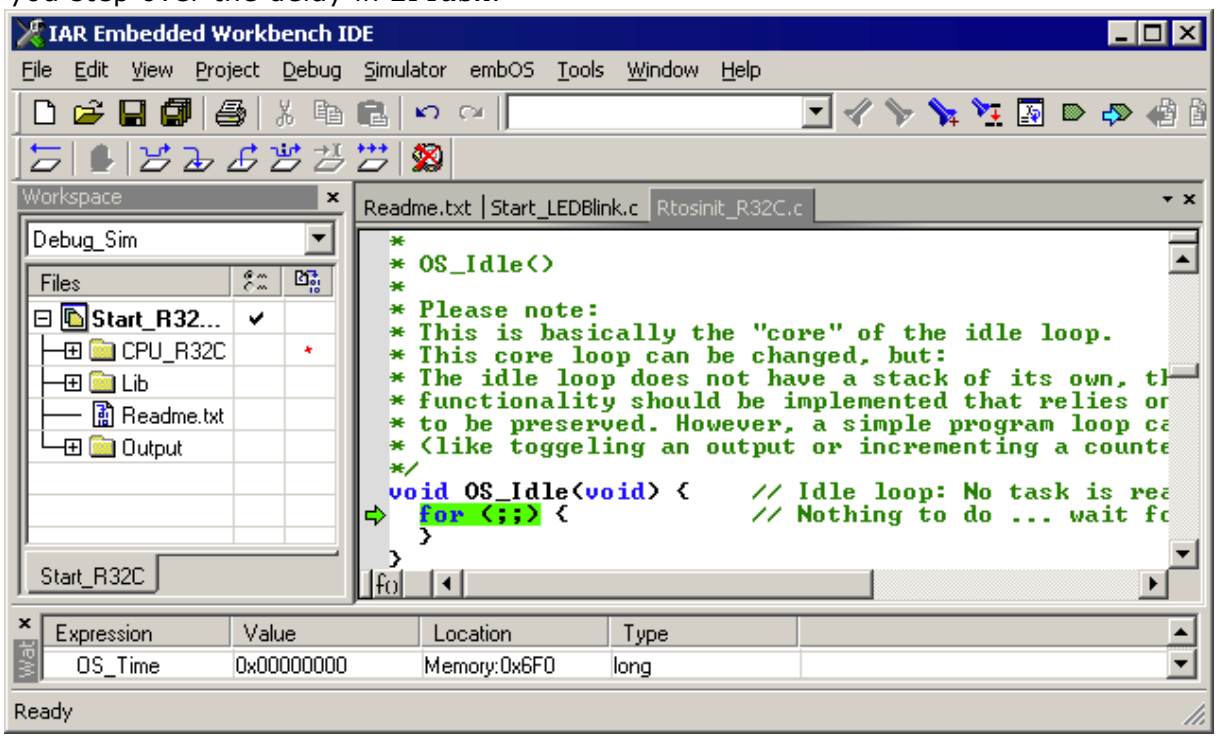


If you continue stepping, you will arrive in the task that has lower priority:



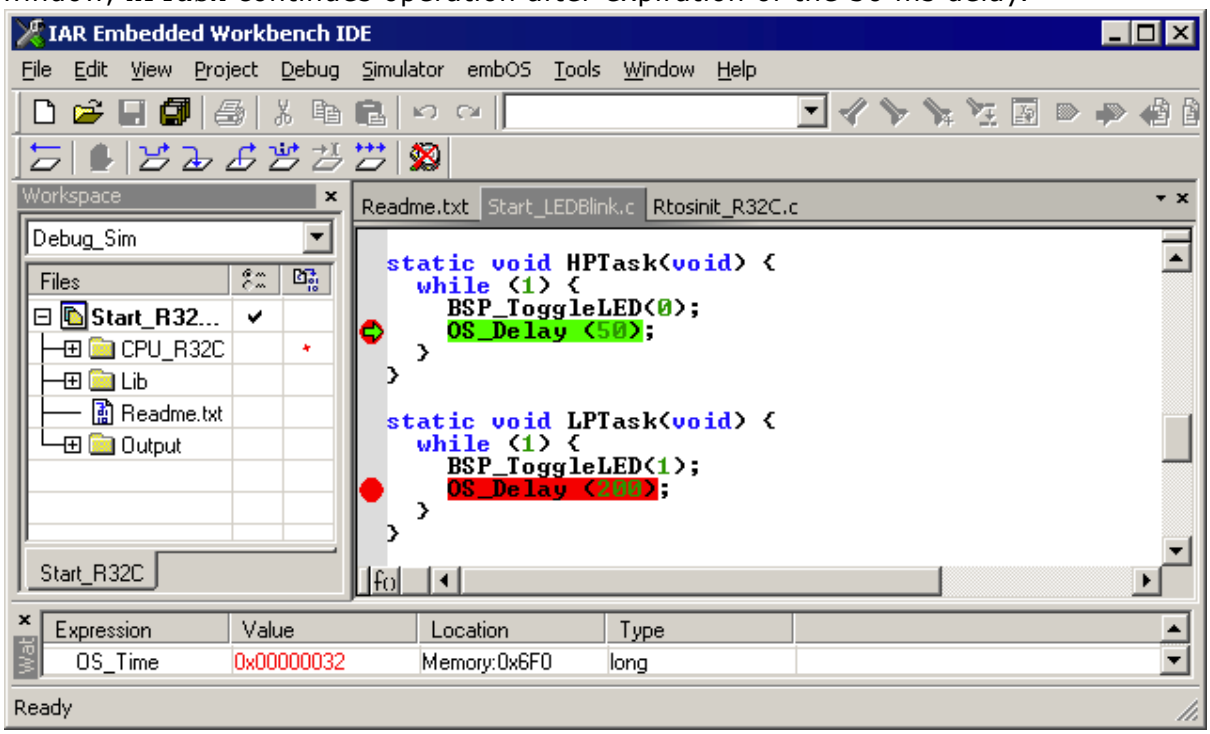
Continue to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

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 you step 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_Time, shown in the Watch window, `HPTask` continues operation after expiration of the 50 ms delay.



4.5 Using E8A or other in circuit emulators

The standard distribution of embOS for R32C and IAR compiler contains a configuration for RENESAS's emulator E8A.
This configuration is named "Debug" and it produces an output file with debug information which may be loaded into the target CPU's internal flash memory via C-SPY.
The sample start project for the RSK_R32C is setup to use the E8A emulator as debugging tool.

4.6 Common debugging hints

For debugging your application, you should use a debug build, e.g. use the debug build libraries in your projects if possible. The debug build contains additional error check functions which are called during runtime.
When an error is detected, the debug libraries call OS_Error(), which is defined in the separate file OS_Error.c.
Using an emulator you should set a breakpoint there. The actual error code is assigned to the global variable OS_Status. The program then waits for this variable to be reset. This allows to get back to the program-code that caused the problem easily: Simply reset this variable to 0 using your in circuit-emulator, and you can step back to the program sequence causing the problem. Most of the time, a look at this part of the program will make the problem clear.
How to select an other library with debug code for your projects is described later on in this manual.

Chapter 5

Build your own application

This chapter provides all information to setup your own embOS project.

5.1 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 "First steps" on page 11 and modify the project to fit your needs. Using a sample project as starting point has the advantage that all necessary files are included and all settings for the project are already done.

5.2 Required files for an embOS for RENESAS R32C application

To build an application using embOS, the following files from your embOS distribution are required and have to be included in your project:

- `RTOS.h` from 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 the target specific subfolder **BoardSupport\CPU_R32C\Setup**.
  It contains hardware-dependent initialization code for embOS. It initializes the system timer, timer interrupt and optional communication for embOSView via UART.
- One embOS library from the subfolder `Lib\`.
- OS_Error.c from the target specific subfolder **BoardSupport\CPU_R32C\Setup**.
  The error handler is used if any library other than Release build library is used in your project.
- Additional low level init code 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.
Also ensure, that `main()` is called with the CPU running in supervisor or system mode.
Your `main()` function has to initialize embOS by a call of `OS_InitKern()` and `OS_InitHW()` prior any other embOS embOS functions are called.
You should then modify or replace the `Start_2Task.c` source file in the subfolder `Application\`.

5.3 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 a new `Lib` group to your project and add this library to the new group. Exclude all other library groups from build, delete unused `Lib` groups 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.

5.4 Select another CPU

embOS contains CPU-specific code for various Renesas R32C 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.

Chapter 6

Renesas R32C version specifics

6.1 Memory models

embOS for Renesas R32C supports all memory and code model combinations that IAR compiler for R32C supports.
For the R32C, 2 code and 1 data memory models are available:

Data model	Code model	Data area
Near	Far (24 bits) 0xFF800000-0xFFFFFFFF	0x0-0x7FFF; 0xFFFF8000 – 0xFFFFFFFF
Near	Huge (32 bits) 0x0-0xFFFFFFFF	0x0-0x7FFF; 0xFFFF8000 – 0xFFFFFFFF
Far	Far (24 bits) 0xFF800000-0xFFFFFFFF	0x0-0x7FFFF; 0xFF800000 – 0xFFFFFFFF
Far	Huge (32 bits) 0x0-0xFFFFFFFF	0x0-0x7FFFF; 0xFF800000 – 0xFFFFFFFF
Huge	Far (24 bits) 0xFF800000-0xFFFFFFFF	0x0-0xFFFFFFFF
Huge	Huge (32 bits) 0x0-0xFFFFFFFF	0x0-0xFFFFFFFF

6.2 Available libraries

The files to use depend on the target CPU, code model and library type which should be used.
The library files are located in the subfolder 'Lib' in the start project folder.
The CPU type selection and memory model settings for your target application have to confirm to the library used in your application.

6.3 Naming conventions for prebuild libraries

The naming convention for library files follows the naming convention of the IAR system libraries an is as follows:

rtos<CPU><code model><float><double>_<TYPE>.r53

<CPU> specifies the CPU family: R32C

<code model> specifies the code model:
·**f** for Far code model
·**h** for Huge code model

<float> specifies the implementation of floating point calculation:
·**h** uses the floating point hardware of the CPU.
·**s** for software floating point calculation functions, fully IEEE compliant.

<double> specifies whether 32 or 64 bit doubles are used:
·**f** uses standard floating point doubles of 32bit.
·**d** uses 64bit floating point doubles.

<TYPE> specifies the type of the embOS -library:
·**XR** stands for eXtreme Release build library.
·**R** stands for Release build library.
·**S** stands for Stack check library, which performs stack checks during runtime.
·**SP** stands for Stack check and Profiling library, which performs stack checking and additional runtime (Profiling) calculations
·**D** stands for Debug library which performs error checking during runtime.
·**DP** stands for Debug and Profiling library which performs error checking and addi-

tional Profiling during runtime.
·DT stands for Debug and Trace library which performs error checking and additional Trace functionality during runtime.

Example:

rtosR32Cfsf_SP.r53 is the embOS library for an **R32C** CPU with **f**ar code model, full (**s**oftware) **f**loating point calculations, 32bit doubles, with **S**tack check and **P**rofiling functionality.
Note that the embOS libraries can be used for any data model.

6.4 CPU specific settings

embOS may be used with any R32C CPU variant. Our start projects are set up for a "generic" CPU. You should select your specific CPU as project option.

Chapter 7

Stacks

7.1 Task stack for Renesas R32C

Each task uses its individual stack. The stack poiter 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 R32C CPUs, this minimum basic task stack size is about 52 bytes in the near memory model. Because any function call uses some amount of stack the task stack size has to be large enough. We recommend at least 256 bytes stack as a start.

7.2 System stack for Renesas R32C

The minimum system stack size required by embOS is about 40 bytes (stack check & profiling build) However, since the system stack is also used by the application before the start of multitasking (the call to OS_Start()), and because software-timers and .C.-level interrupt handlers also use the system-stack, the actual stack requirements depend on the application. We recommend at least a minimum of 128 bytes. *embOS* uses IARs CSTACK as system stack.
The size of the system stack can be changed by modifying the CSTACK size in your *.icf file. We recommend a minimum stack size of 256 bytes for the CSTACK.

7.3 Interrupt stack for Renesas R32C

The R32C has been designed with multitasking in mind; it has 2 stack-pointers, the USP and the ISP. The U-Flag selects the active stack-pointer. During exe-cution of a task or timer, the U-flag is set thereby selecting the user-stack-pointer. If an interrupt occurs, the R32C 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 R32C.
IAR defines the interrupt stack as ISTACK in the linker files. The size of the interrupt stack is defined in the link-file as _ISTACK_SIZE or may be set up as project option “Interrupt stack size”. (We recommend at least 256 bytes if multiple nested nested interrupts are allowed)

7.4 Stack specifics of the Renesas R32C family

Because the stack-pointer of the R32C CPUs can address the entire memory area, stacks can be located anywhere in RAM. For performance reasons you should try to locate stacks in fast internal RAM.

Chapter 8

Interrupts

8.1 What happens when an interrupt occurs?

- The CPU-core receives an interrupt request
- As soon as the interrupts are enabled and the processor interrupt priority level is below or equal to the interrupt priority level, the interrupt is executed
- the CPU switches to the Interrupt stack
- the CPU saves PC and flags on the stack
- the IPL is loaded with the priority of the interrupt
- the CPU jumps to the address specified in the vector table for the interrupt service routine (ISR)
- ISR: Save registers
- ISR: User-defined functionality
- ISR: Restore registers
- ISR: Execute REIT command, restoring PC, Flags and switching to User stack
- For details, refer to the Renesas users manual.

8.2 Defining interrupt handlers in C

Routines defined with the keyword __interrupt automatically save & restore the registers they modify and return with REIT.
For a detailed description on how to define an interrupt routine in "C", refer to the IAR R32C C-Compiler reference guide.
For details how to write interrupt handler using embOS functions, refer to the embOS generic manual.
For details about interrupt priorities, refer to chapter “Interrupt priorities”.

Example:

Simple interrupt routine:

```
//
//Interrupt handler NOT using embOS functions
//

#pragma vector= (TIMER_A1)
__interrupt void IntHandlerTimerA1(void) {
  IntCnt++;
}

//
//Interrupt function using embOS functions
//

#pragma vector = (TIMER_A0)
__interrupt void OS_ISR_Tick (void) {
  OS_EnterNestableInterrupt();  // Inform embOS that interrupt code is running
  OS_HandleTick();
  OS_LeaveNestableInterrupt();
}
```

8.3 Interrupt-stack switching

Since the R32C CPUs have a separate stack pointer for interrupts, there is no need for explicit software stack-switching in an interrupt routine. The routines OS_EnterIntStack() and OS_LeaveIntStack() are supplied for source compatibility to other processors only and have no functionality.

8.4 Zero latency interrupts with Renesas R32C

Instead of disabling interrupts when embOS does atomic operations, the interrupt level of the CPU is set to a specific value. Initially, this value is initialized to 4, but may be modified during system initialization by a call of the function OS_SetFastIntPriorityLimit().

Therefore all interrupts with level 5 or above can still be processed.
These interrupts are named *Zero latency interrupts*. You must not execute any embOS function from within an interrupt running on high priority.

8.5 Interrupt priorities

This chapter describes interrupt priorities supported by the RENESAS R32C cores.

8.5.1 Interrupt priorities with Renesas R32C cores

With introduction of Zero latency interrupts, interrupt priorities useable for interrupts using embOS API functions are limited.

- Any interrupt handler using embOS API functions has to run with interrupt priorities from 1 to 4. These embOS interrupt handlers have to start with OS_EnterInterrupt() or OS_EnterNestableInterrupt() and have to end with OS_LeaveInterrupt() or OS_LeaveNestableInterrupt().
- Any Zero latency interrupt (running at priorities from 5 to 7) must not call any embOS API function. Even OS_EnterInterrupt() and OS_LeaveInterrupt() must not be called.
- Interrupt handler running at low priorities (from 1 to 4) not calling any embOS API function are allowed, but must not reenable interrupts!

The priority limit between embOS interrupts and Zero latency interrupts is initially set to 4, but can be changed at runtime by a call of the function OS_SetFastIntPriorityLimit().

8.6 OS_SetFastIntPriorityLimit()

The interrupt priority limit for fast interrupts is set to 4 by default. This means, all interrupts with higher priority from 4 to 7 will never be disabled by embOS.

Description

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

Prototype

```
void OS_SetFastIntPriorityLimit(unsigned int Priority)
```

Parameter	Description
Priority	The highest value useable as priority for embOS interrupts. All interrupts with higher priority are never disabled by em-bOS. Valid range: 1 <= Priority <= 7.

Return value

None.

Additional Information

To disable fast interrupts at all, the priority limit may be set to 7 which is the highest interrupt priority for interrupts.
To modify the default priority limit, OS_SetFastIntPriorityLimit() should be called before embOS was started.
In the default projects, OS_SetFastIntPriorityLimit() is not called. The start projects use the default fast interrupt priority limit.
Any interrupts running above the fast interrupt priority limit must not call any embOS function.

Chapter 9

STOP / WAIT Mode

9.1 Introduction

In case your controller does support some kind of power saving mode, it should be possible to use it also with embOS, as long as the timer keeps working and timer interrupts are processed. To enter that mode, you usually have to implement some special sequence in the function `OS_Idle()`, which is part of the hardware dependent module RTOSInit.c.
The stop-mode works without a problem; however the real-time operating system is halted during the execution of the stop-instruction if the timer that the scheduler uses is supplied from the internal clock. With external clock, the scheduler keeps working.

Chapter 10

Technical data

10.1 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 2.500 bytes.

In the table below, which is for 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	44
Resource semaphore	16
Counting semaphore	8
Mailbox	24
Software timer	20

Chapter 11

Files shipped with embOS

List of files shipped with embOS

Directory	File	Explanation
root	*.pdf	Generic API and target specific documentation.
root	Release.html	Version control document.
root	embOSView.exe	Utility for runtime analysis, described in generic documentation.
Start\Inc\	RTOS.h BSP.h	Include file for embOS, to be included in every C-file using embOS functions.
Start\Lib\	rtos*.r53	embOS libraries for IAR compiler for R32C.
Start\ BoardSupport\		Sample workspaces and project files for IAR Embedded Workbench, contained in manufacturer specific sub folders.
Start\BoardSupport\CPU_R32C\	Start_R32C.ew*	CPU specific sample workspace and project.
Start\BoardSupport\RSK_R32C111\	Start_RSK_R32C.ew*	CPU specific workspace and project for the RSK starter kit.
Start\BoardSupport\...\Setup\	*.*	CPU specific hardware routines for various CPUs.
Start\BoardSupport\..\Setup	OS_Error.c	embOS runtime error handler used in stack check or debug builds.

Any additional files shipped serve as example.

Index

