embOS

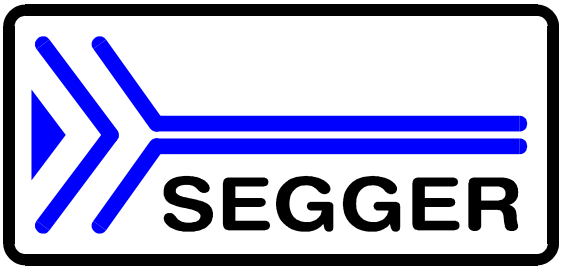
Real Time Operating System

CPU & Compiler specifics for

RENESAS R32C CPUs

and RENESAS HEW 4

Document Rev. 0



A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Contents

1. About this document

This guide describes how to use ***embOS*** for R32C Real Time Operating System for the RENESAS R32C series of microcontroller using RENESAS HEW for R32C.

1.1. How to use this manual

This manual describes all CPU and compiler specifics for ***embOS*** R32C for the HEW workbench and compiler. Before actually using ***embOS***, you should read or at least glance through this manual in order to become familiar with the software.
Chapter 2 gives you a step-by-step introduction, how to install and use ***embOS*** using the HEW workbench. If you have no experience using ***embOS***, you should follow this introduction, the HEW Workbench, because it is the easiest way to learn how to use ***embOS*** in your application.
Most of the other chapters in this document are intended to provide you with detailed information about functionality and fine-tuning of ***embOS*** for the RENESAS R32C using the RENESAS compiler.

2. Using ***embOS*** with HEW Workbench

2.1. Installation

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

In order 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, please keep all files in their respective sub directories. Make sure the files are not read only after copying.
If you received a zip-file, please extract it to any folder of your choice, preserving the directory structure of the zip-file.

Assuming that you are using the HEW Workbench to develop your application, no further installation steps are required. You will find a prepared sample start project for R32C CPUs, which you should use and modify to write your application. So follow the instructions of the next chapter 'First steps'.

You should do this even if you do not intend to use the HEW Embedded Workbench for your application development in order to become familiar with ***embOS.***

If for some reason you will not work with the HEW 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 HEW Embedded Workbench, it may be used without the workbench using batch files or a make utility without any problem.

2.2. First steps

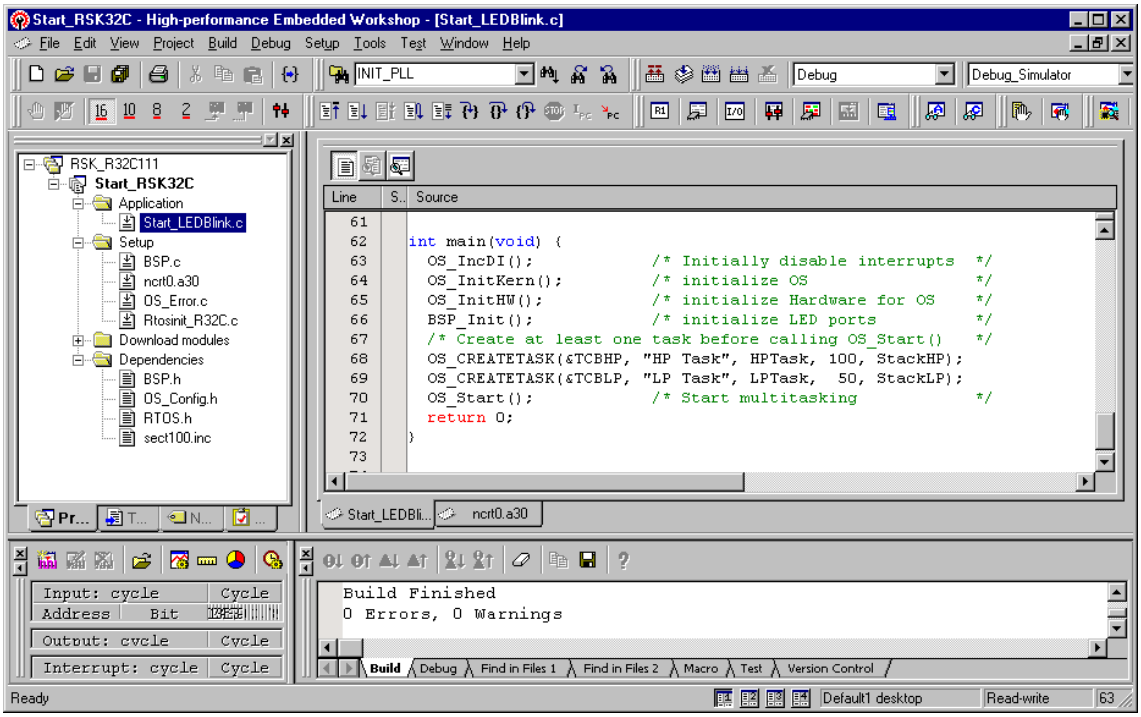
After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received a ready to go sample start workspace and project for the R32C CPU and it is a good idea to use this as a starting point of all your applications.

Your ***embOS*** distribution contains one folder ‘Start’ which contains the sample start workspace and project and every additional files used to build your first ***embOS*** application.

To get your application running, you should proceed as follows.

- Create a work directory for your application, for example c:\work
- Copy all files and subdirectories from the ***embOS*** distribution into your work directory.
- Clear the read only attribute of all files in the new ‘Start’-folder in your working directory.
- Open the folder ‘Start’.
- Open the start workspace ‘RSK_R32C111.hws’. (e.g. by double clicking it)
- Select a configuration, for example ‘Debug_Simulator’.
- Build the start project

After building the start project your screen should look like follows:



For latest information you should open the ReadMe.txt which is part of your project.

2.3. The sample application Start_LEDBlink.c

The following is a printout of the sample application Start_LEDBlink.c. It is a good starting-point for your application.
The start project may contain an other application which is very similar.
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.

```
----------------------------------------------------------------------
File    : Start_LEDBlink.c
Purpose : Sample program for OS running on EVAL-boards with LEDs
--------- END-OF-HEADER ----------------------------------------------*/

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

OS_STACKPTR int StackHP[128], StackLP[128];          /* Task stacks */
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_IncDI();                      /* Initially disable interrupts  */
  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;
}
```

3. Using debugging tools to debug the application

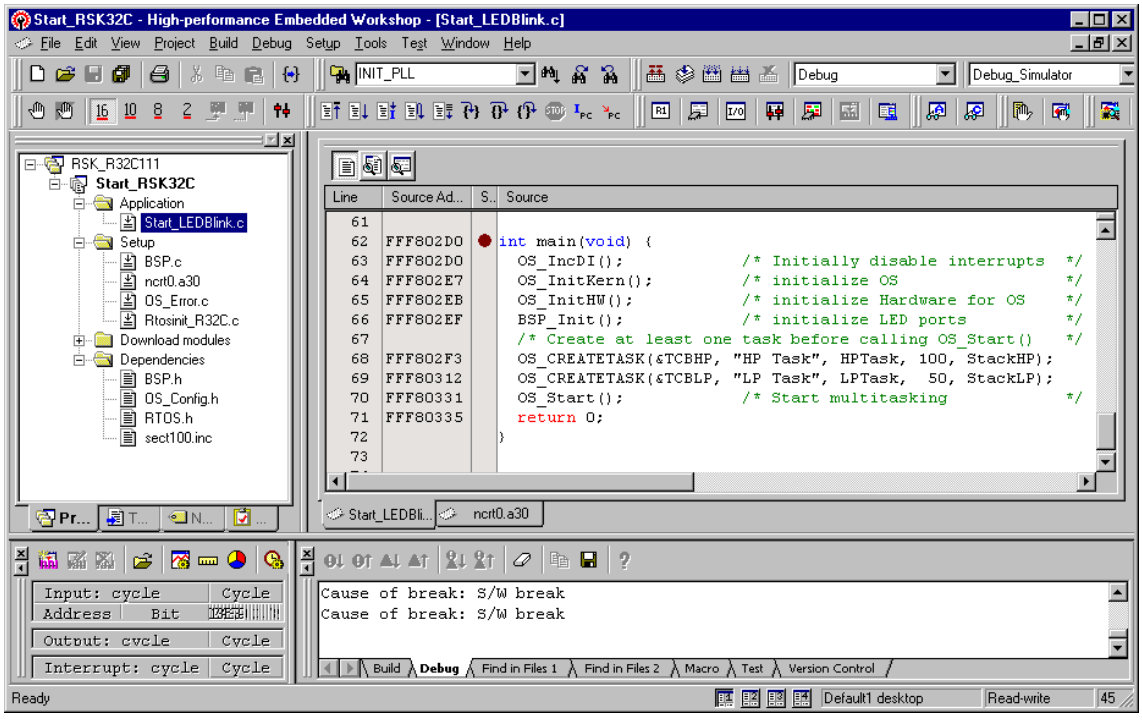
The ***embOS*** start project contains configurations which are already setup for the following debugging tools:

- The R32C simulator. This configuration is named “Debug_Simulator”.
- RENESAS’s emulator E8a. This configuration is named “Debug_E8a”.

All configurations are prepared to produce the appropriate output files required by the selected debugger or emulator debugger.
The following chapter describe a sample session based on our sample application Start_LEDBlink, using the simulator. Using the E8a emulator, the debugging session is very similar.

3.1. Using the R32C simulator

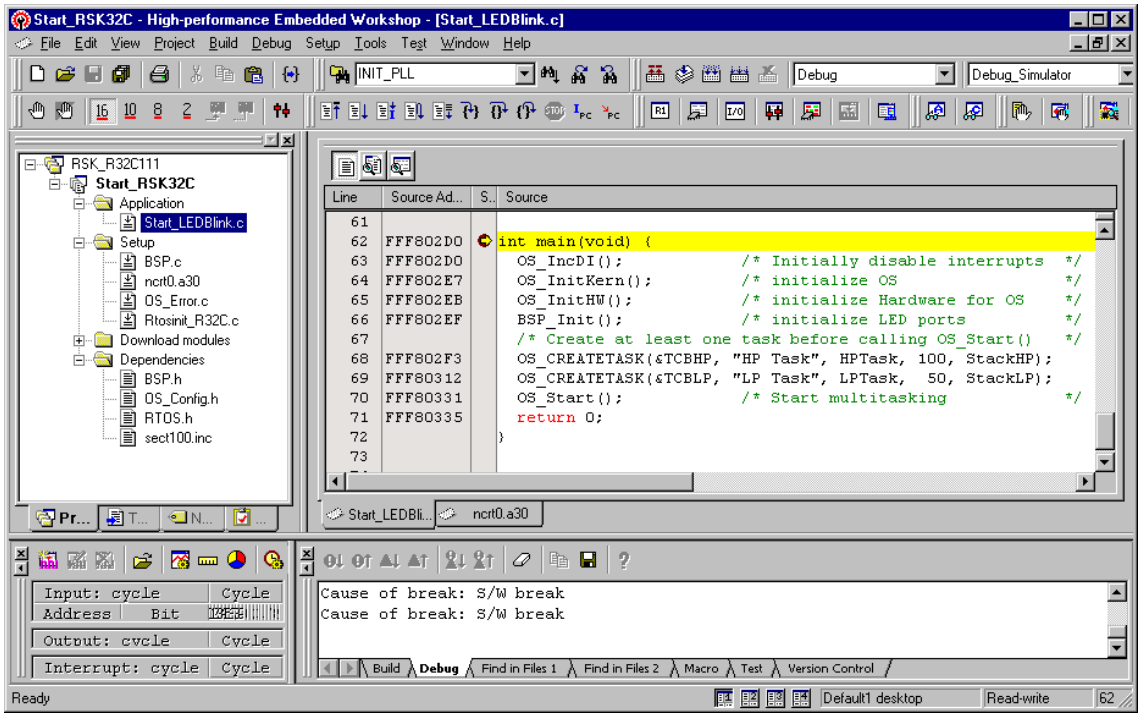
According to project settings, the simulator may start automatically after building the project. Otherwise, activate the connection and download the output module of the project.
After download, perform a CPU reset.
You will usually see the startup function.
Open the source file Start_LEDBilnk.c and place a break point at the main() function:



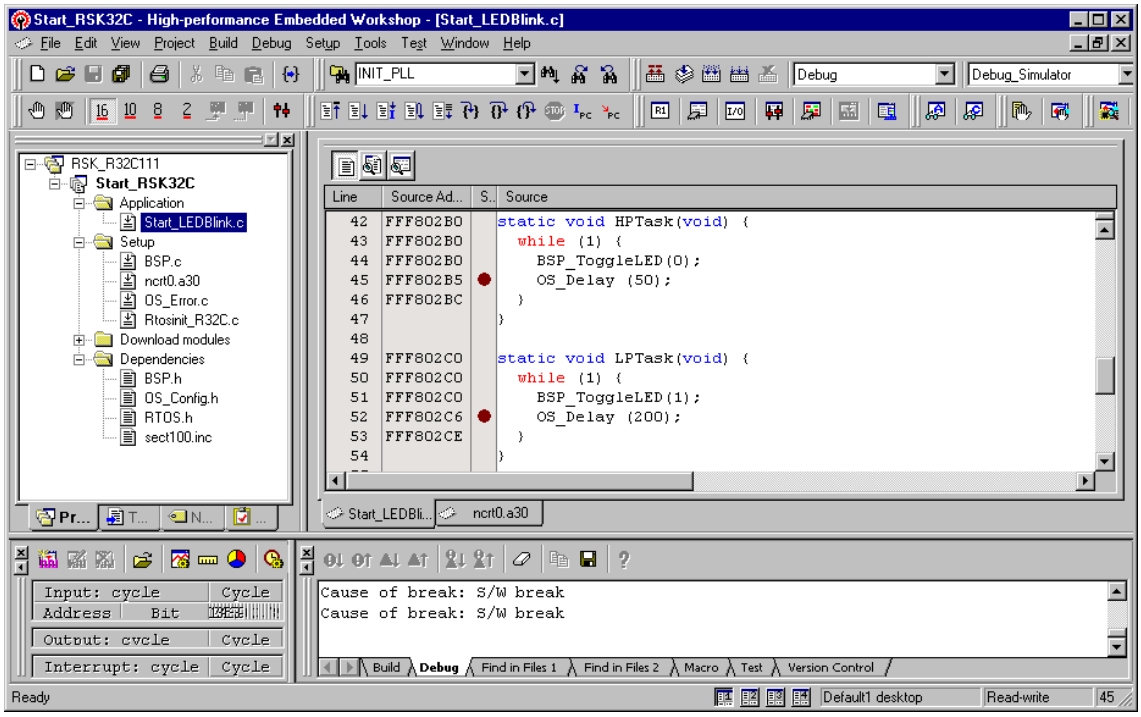
Then start the application by “Debug -> Go”, e.g. press “F5”.
The simulator will stop at the main function.
Now you can step through the sample application program.

`OS_IncDI()`initially disables interrupts and prevents OS_InitKern() from re-enabling them.
`OS_InitKern()`initializes ***embOS*** -Variables. As this function is part of the ***embOS*** library, you may step into it in disassembly mode only.
`OS_InitHW()` is part of RTOSINIT_R32C.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.

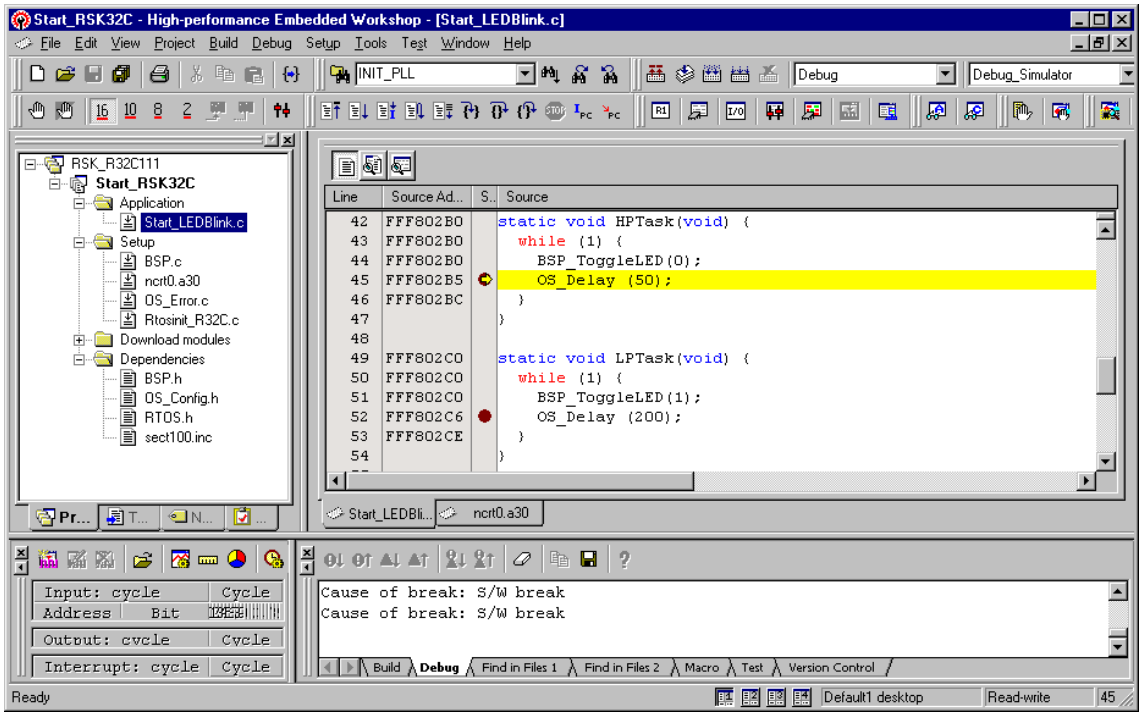
`OS_Start()` should be the last line in main, since it starts multitasking and does not return.



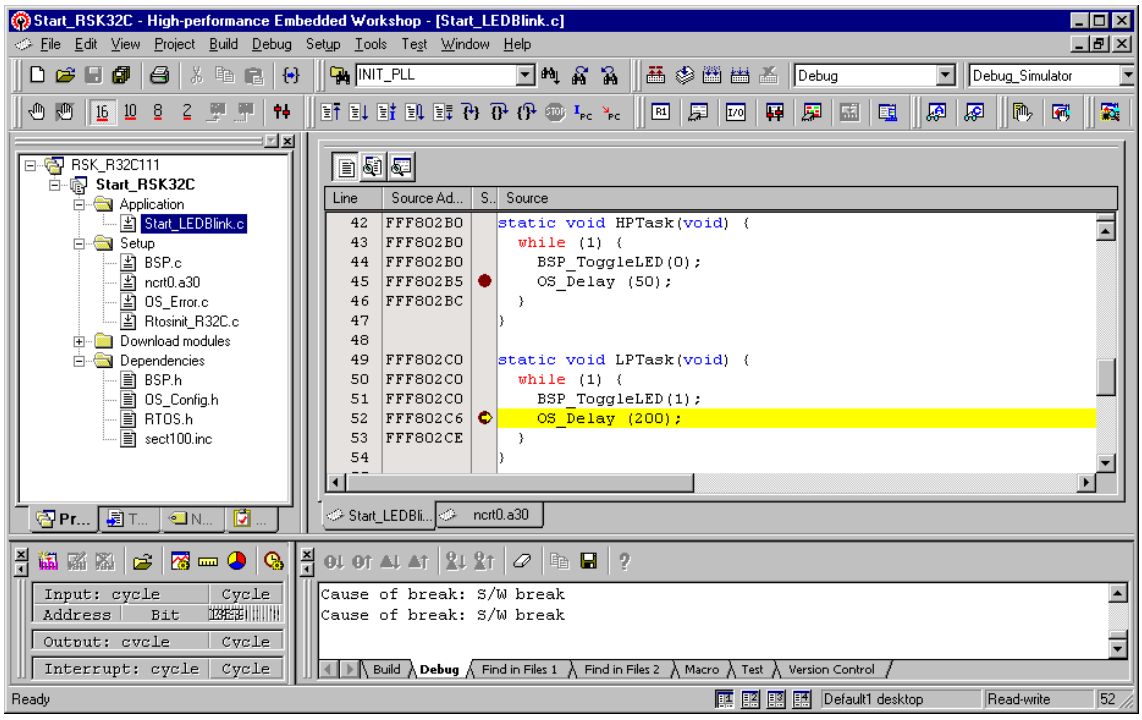
Before you step into `OS_Start()`, you should set breakpoints in the two tasks:



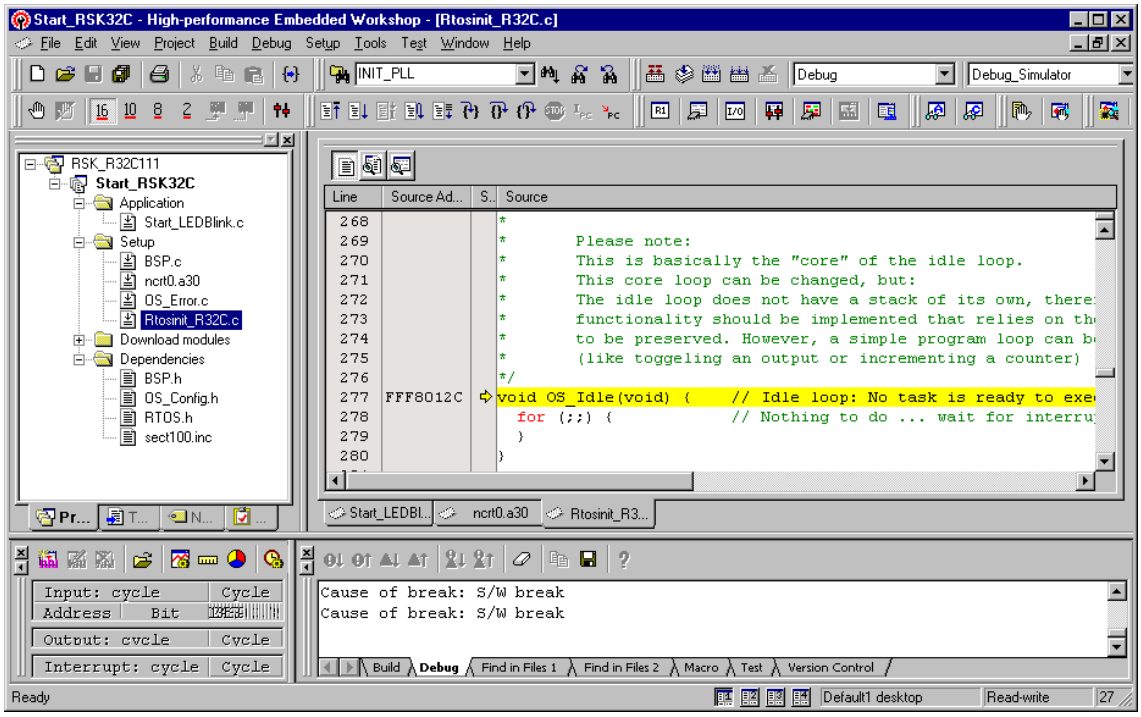
When you step over `OS_Start()`, the next line executed is already in the highest priority task created. (you may also step into `OS_Start()`, then stepping through the task switching process in disassembly mode). In our small start program, `HPTask()` is the highest priority task and is therefore active.



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

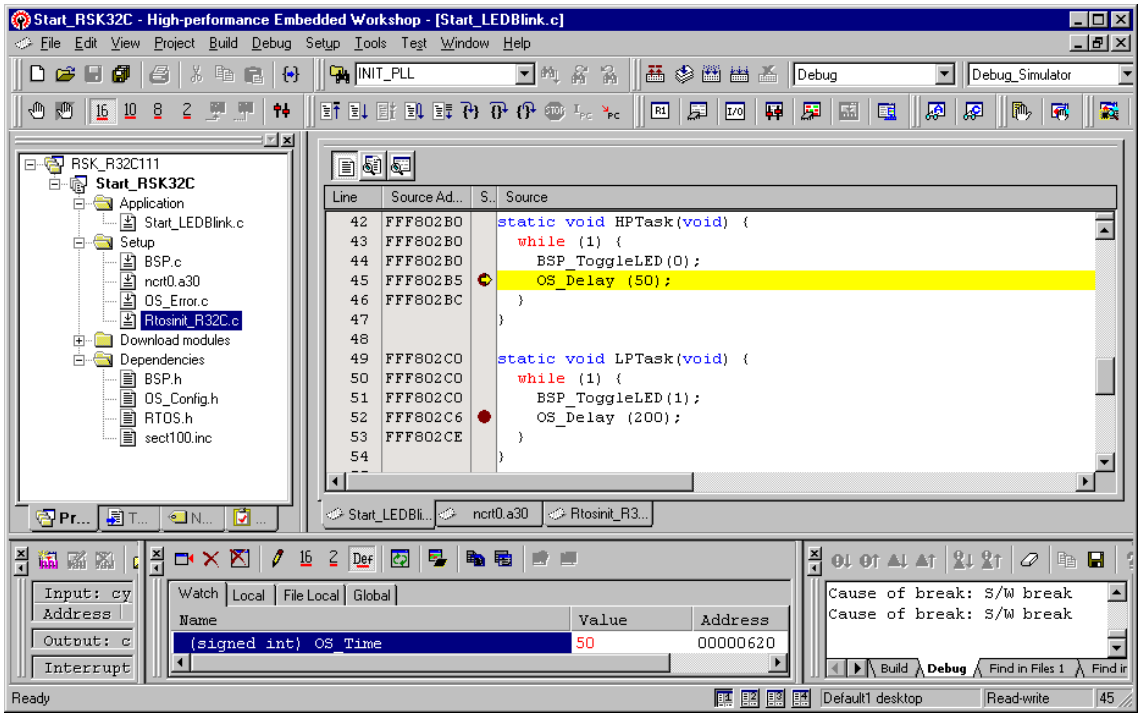


Continuing to step through the program, there is no other task ready for execution. ***embOS*** will suspend LPTask and switch to the idle-loop, which is an endless loop which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).
`OS_Idle()` is found in RTOSInit_R32C.c
You will arrive there, when you step through the task switching process in disassembly mode:



If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay.
Coming from `OS_Idle()`, you should execute the ‘Go’ command to arrive at the highest priority task after its delay is expired.

The watch window shows the system variable `OS_Time`, which shows how much time has expired in the target system.



Note that the I/O simulation window has to be open, and the simple timer “embOS_Timer.stm” from the “Setup” folder has to be started to simulate the ***embOS*** timer interrupts.

3.2. Using E8A or other in circuit emulators

The standard distribution of ***embOS*** for R32C and HEW contains a configuration for the RENESAS emulator E8a.
This configuration is named “Debug_E8a” and it produces an output file with debug information which may be loaded into the target CPU’s internal flash memory. The sample start project is built for the RSK32C_111 eval kit with an R32C111 CPU and may have to be adapted, if an other board or CPU is used.

3.3. 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.

4. Build your own application

To build your own application, you should start with the sample start project. This has the advantage, that all necessary files are included and all settings for the project are already done.

4.1. Required files for an *embOS* 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 sub folder Inc\
  This header file declares all ***embOS*** API functions and data types and has to be included in any source file using ***embOS*** functions.
- **ncrt0.a30** from the subfolder RSK_R32C111\Setup\.
  This is the startup code for the CPU. It initializes the stack poiters, the vector base register and the variables. It is almost identical to the startup files which came with the workbench or starterkit software. For ***embOS***, some variables are declared global to export information about the stack addresses and sizes.
- **RTOSInit_R32C.c** from the subfolder RSK_R32C111\Setup\.
  It contains the hardware dependent initialization code for the ***embOS*** timer and optional code for the UART for embOSView.
- **OS_Error.c** from the subfolder RSK_R32C111\Setup\.
  It contains the ***embOS*** runtime error handler `OS_Error()`which is used in stack check or debug builds.
- One ***embOS*** **library** from the Lib\ subfolder

When you decide to write your own startup code, please 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 call of `OS_InitKern()` and `OS_InitHW()` prior any other ***embOS*** functions except `OS_incDI()` are called.

4.2. Select a start project

embOS comes with one start project which includes different configurations for different debug tools. The start project was built and tested with one specific R32C CPU. For your specific CPU variant there may be modifications required.
If you have to modify the code for your specific CPU, you may copy and rename the whole RSK_R32C111 folder and use the new sources in your project.

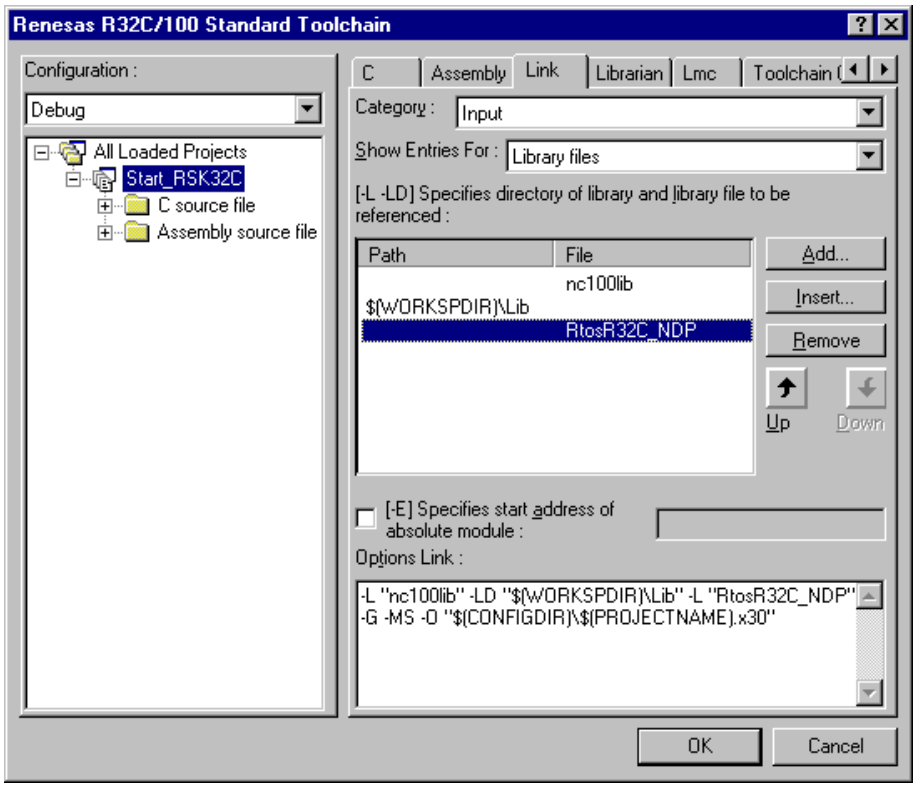
4.3. Add your own code

For your own code, you may add a new group to the project.
You should then modify or replace the source file containing the main() function.

4.4. Change memory model or library mode

For your application you may have to choose an other memory-model. 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.

To replace the ***embOS*** library in your project, you have to modify the linker settings in the project options:



Double-click on the ***embOS*** library name and type the name of the new library which you want to use in the project. Don't enter the extension.

To change the data memory model, select the "C" options, Category "Code Modification":

- check the "[-fFRAM] option to select the far data model
- un-check the "[-fFRAM] option to select the near data model

Ensure, the corresponding ***embOS*** library is used in the project.
Refer to chapter 5 about the library naming conventions to select the correct library.

5. R32C specifics

5.1. Memory models

embOS supports the near and far data memory models that the RENESAS compiler supports.
The near code model is not supported.
For the R32C, 2 data memory models are available:

Data Model	Code area (always far)	Data area
Near	0x0-0x7FFFFF; 0xFF800000-0xFFFFFFFF	0x0-0x7FFF; 0xFFFF8000 – 0xFFFFFFFF
Far	0x0-0x7FFFFF; 0xFF800000-0xFFFFFFFF	0x0-0x7FFFFF; 0xFF800000-0xFFFFFFFF

5.2. Available libraries

The files to use depend on the required data memory model and the library type which should be used.
The library files are located in the subfolder 'Lib' in the start project folder.

The naming convention for the embOS library files is as follows:
Rtos<CPU>_<DATA MODEL><TYPE>.lib

<CPU> specifies the CPU family: **R32C**

< DATA MODEL > specifies the data model to be used:

- **N** for Near data model
- **F** for Far data model

<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 additional Profiling during runtime.
- **DT** stands for Debug and Trace library which performs error checking and additional Trace functionality during runtime.

Example:

RtosR32C_NSP.lib is the ***embOS*** library for an **R32C** CPU with **N**ear memory model, with **S**tack check and **P**rofiling functionality.

5.3. CPU specific settings

embOS may be used with any R32C CPU variant. Our start projects are set up for the R32C/111 CPU. You may have to modify the segment settings in the sect100.inc file according to the memory layout of your specific CPU.

You may also have to verify and modify the PLL initialization code found in the `OS_InitHW()` function in the RTOSInit-file.

6. Stacks

6.1. Task stack for R32C

Every ***embOS*** task has to have its own stack. Task stacks can be located in any RAM memory location.
The stack-size required is the sum of the stack-size of all routines called from the task, all local variables used in the functions, plus basic stack size.
The basic stack size is the size of memory required to store the registers of the CPU plus the stack size required by ***embOS*** -routines.
For the R32C, this minimum stack size is about 60 bytes in the near memory model.

6.2. System stack for R32C

The system stack size required by ***embOS*** is about 60 bytes (80 bytes in debug or profiling builds) 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. We recommend at least a minimum of 128 bytes.
embOS uses the user stack as system stack. The `main()` function has to be called with the user stack selected.
The size of the stack is configured using the __USTACKSIZE__ definition in the Assembly settings in the project, or by the USTACKSIZE definition in the startup code ncrt0.a30as.

6.3. Interrupt stack for 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 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 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.
The size of the stack is configured using the __ISTACKSIZE__ definition in the Assembly settings in the project, or by the ISTACKSIZE definition in the startup code ncrt0.a30as.
We recommend at least a minimum of 256 bytes if multiple nested interrupts are allowed.

6.4. 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.

7. Interrupts

7.1. 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 accepted
- the CPU switches to the Interrupt stack
- the CPU saves PC and flags on the stack
- the CPU disables all further interrupts
- 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.

7.2. Defining interrupt handlers in "C"

Routines defined with the #pragma 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 C-Compiler User's manual.
For details how to write interrupt handler using ***embOS*** functions, also refer to the ***embOS*** generic manual.
For details about interrupt priorities, refer to chapter "Interrupt priorities".

Example

"Simple" interrupt-routines

```
//
// Interrupt handler NOT using embOS functions
//
#pragma INTERRUPT IntHandlerTimerA1(vect=13);
void IntHandlerTimerA1(void);
void IntHandlerTimerA1(void) {
  IntCnt++;
}

//
// Interrupt function using embOS functions
//
#pragma INTERRUPT OS_ISR_Tick (vect=12);
void OS_ISR_Tick (void);
void OS_ISR_Tick (void) {
  OS_EnterNestableInterrupt();
  OS_HandleTick();
  OS_LeaveNestableInterrupt();
}
```

7.3. Interrupt-stack

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.

7.4. Zero latency interrupts with 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 pre-set 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.

7.5. Interrupt priorities

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 must 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 re-enable 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()`.

7.6. OS_SetFastIntPriorityLimit(): Set the interrupt priority limit for Zero Latency (fast) Interrupts

The interrupt priority limit for zero latency 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 zero latency interrupts and lower priority *embOS* interrupts.

Prototype

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

Parameter	Meaning
`Priority`	The highest value useable as priority for *embOS* interrupts. All interrupts with higher priority are never disabled by *embOS*. Valid range: 1 <= `Priority` <= 7

Return value

NONE.

Add. information

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

7.7. Fast interrupt, R32C specific

The R32C 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.

7.8. Non Maskable Interrupt, NMI

The R32C 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.

8. STOP / WAIT Mode

Usage of the wait instruction is one possibility to save power consumption during idle times. If required, you may modify the `OS_Idle()` routine, which is part of the hardware dependent module Rtosinit_R32C.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.

9. Technical data

9.1. 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 ***embOS*** library. The values in the table are for the far memory model and release build library.

Short description	ROM [byte]	RAM [byte]
Kernel	approx.1600	37
Event-management	< 200	---
Mailbox management	< 550	---
Single-byte mailbox management	< 300	---
Resource-semaphore management	< 250	---
Timer-management	< 250	---
Add. Task	---	28
Add. Semaphore	---	6
Add. Mailbox	---	14
Add. Timer	---	14
Power-management	---	---

10. Files shipped with ***embOS*** for R32C and HEW

Directory	File	Explanation
root	`*.pdf`	Generic API and target specific documentation.
root	`Release.html`	Release notes of ***embOS*** R32C.
root	`embOSView.exe`	Utility for runtime analysis, described in generic documentation.
Start\	`*.hws`	Start workspace and project for R32C.
Start\	`Readme.txt`	Latest information about ***embOS*** R32C.
Start\Inc\	`RTOS.h`	To be included in any file using ***embOS*** functions.
Start\Lib\	`Rtos*.lib`	***embOS*** libraries
Start\RSK_R32C111\	`*.*`	CPU specific files, start project and sample applications.

11. Index