embOS

Real-Time Operating System

CPU & Compiler specifics for Renesas RX using GNURX

Document: UM01019
Software Version: 5.16.1.0
Revision: 0
Date: March 10, 2022

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

Copyright notice

Trademarks

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel.	+49 2173-99312-0
Fax.	+49 2173-99312-28
E-mail:	support@segger.com[*]
Internet:	*www.segger.com*

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

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please inform us and we will try to assist you as soon as possible. Contact us for further information on topics or functions that are not yet documented.

Print date: March 10, 2022

Software	Revision	Date	By	Description
5.16.1.0	0	220310	MM	New software version.
4.24	0	160824	RH	Chapter "SEGGER RTT and SystemView" added.
3.88	0	130319	AW	New generic embOS sources V3.88.
3.86n	0	121213	TS	New generic embOS sources V3.86n. Chapter "Stacks" updated.
3.86d	0	120515	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 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 start 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, 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 many 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 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 First Steps

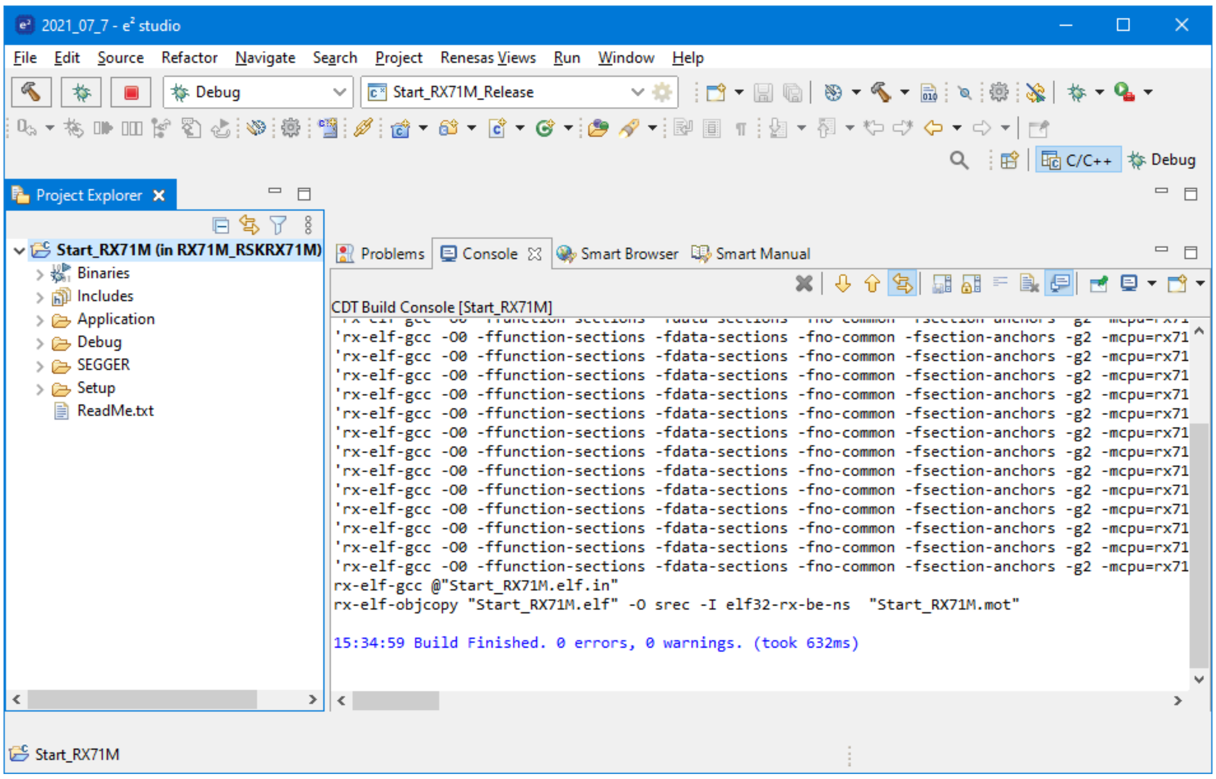
After installation of embOS you can create your first multitasking application. You have received several ready to go sample start workspaces and projects and every other files needed in the subfolder `Start`. It is a good idea to use one of them as a starting point for all of your applications. The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



```
'rx-elf-gcc -O0 -ffunction-sections -fdata-sections -fno-common -fsection-anchors -g2 -mcpu=rx71
'rx-elf-gcc -O0 -ffunction-sections -fdata-sections -fno-common -fsection-anchors -g2 -mcpu=rx71
'rx-elf-gcc -O0 -ffunction-sections -fdata-sections -fno-common -fsection-anchors -g2 -mcpu=rx71
'rx-elf-gcc -O0 -ffunction-sections -fdata-sections -fno-common -fsection-anchors -g2 -mcpu=rx71
'rx-elf-gcc -O0 -ffunction-sections -fdata-sections -fno-common -fsection-anchors -g2 -mcpu=rx71
'rx-elf-gcc -O0 -ffunction-sections -fdata-sections -fno-common -fsection-anchors -g2 -mcpu=rx71
'rx-elf-gcc -O0 -ffunction-sections -fdata-sections -fno-common -fsection-anchors -g2 -mcpu=rx71
'rx-elf-gcc -O0 -ffunction-sections -fdata-sections -fno-common -fsection-anchors -g2 -mcpu=rx71
'rx-elf-gcc -O0 -ffunction-sections -fdata-sections -fno-common -fsection-anchors -g2 -mcpu=rx71
'rx-elf-gcc -O0 -ffunction-sections -fdata-sections -fno-common -fsection-anchors -g2 -mcpu=rx71
'rx-elf-gcc -O0 -ffunction-sections -fdata-sections -fno-common -fsection-anchors -g2 -mcpu=rx71
'rx-elf-gcc -O0 -ffunction-sections -fdata-sections -fno-common -fsection-anchors -g2 -mcpu=rx71
'rx-elf-gcc -O0 -ffunction-sections -fdata-sections -fno-common -fsection-anchors -g2 -mcpu=rx71
rx-elf-gcc @"Start_RX71M.elf.in"
rx-elf-objcopy "Start_RX71M.elf" -O srec -I elf32-rx-be-ns  "Start_RX71M.mot"

15:34:59 Build Finished. 0 errors, 0 warnings. (took 632ms)
```

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.

1.3 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. (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                    *
*                        The Embedded Experts                        *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------
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).
*/

#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 ****************************/
```

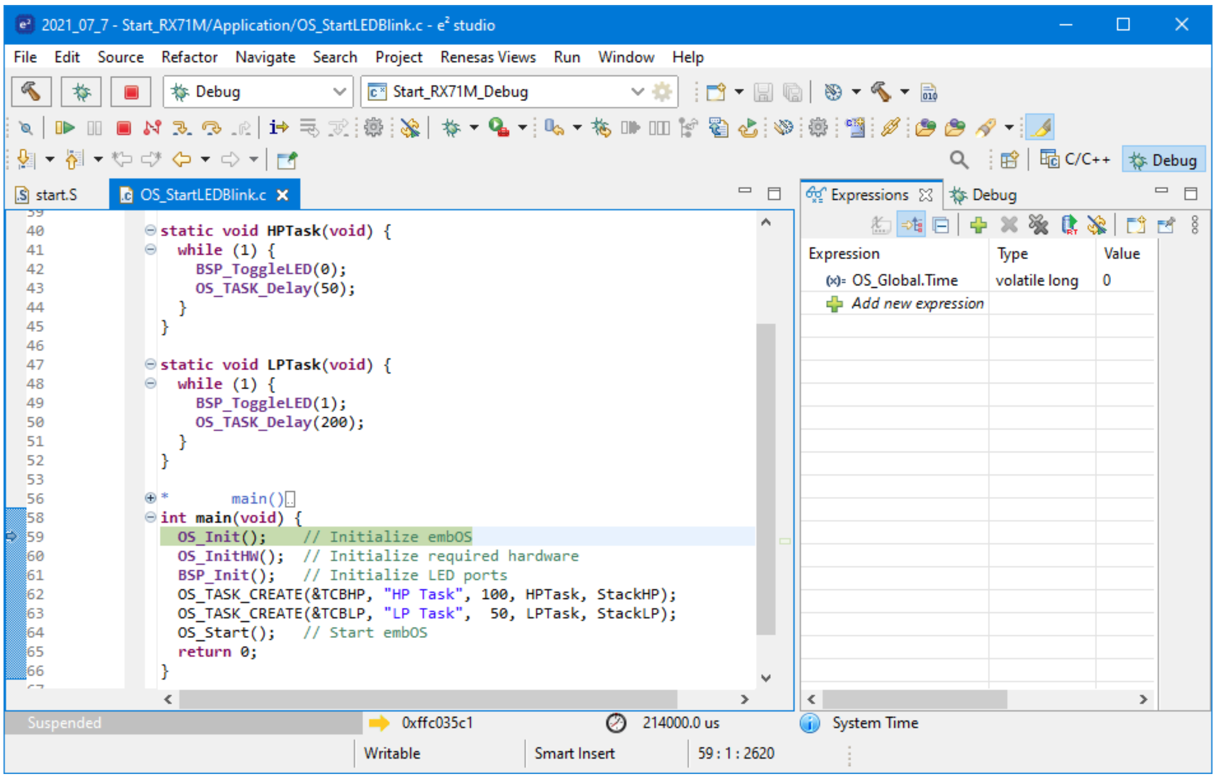
1.4 Stepping through the sample application

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.


2021_07_7 - Start_RX71M/Application/OS_StartLEDBlink.c - e² studio
File Edit Source Refactor Navigate Search Project Renesas Views Run Window Help
Debug
Start_RX71M_Debug
C/C++
Debug
start.S
OS_StartLEDBlink.c
Expressions
Debug

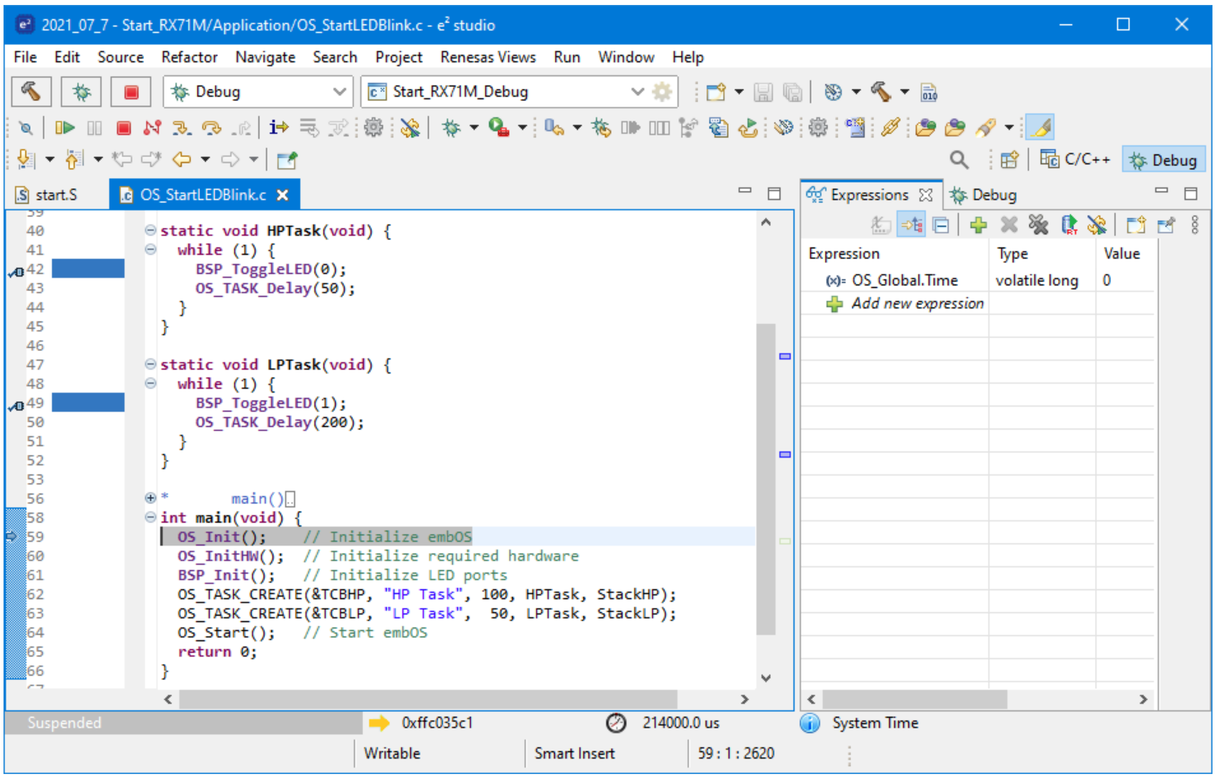
```
static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_TASK_Delay(50);
  }
}

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;
}
```

Expression	Type	Value
OS_Global.Time	volatile long	0
Add new expression		

Suspended | 0xffc035c1 | 214000.0 us | System Time

Writable | Smart Insert | 59 : 1 : 2620

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


2021_07_7 - Start_RX71M/Application/OS_StartLEDBlink.c - e² studio
File Edit Source Refactor Navigate Search Project Renesas Views Run Window Help
Debug
Start_RX71M_Debug
C/C++
Debug
start.S
OS_StartLEDBlink.c
Expressions
Debug

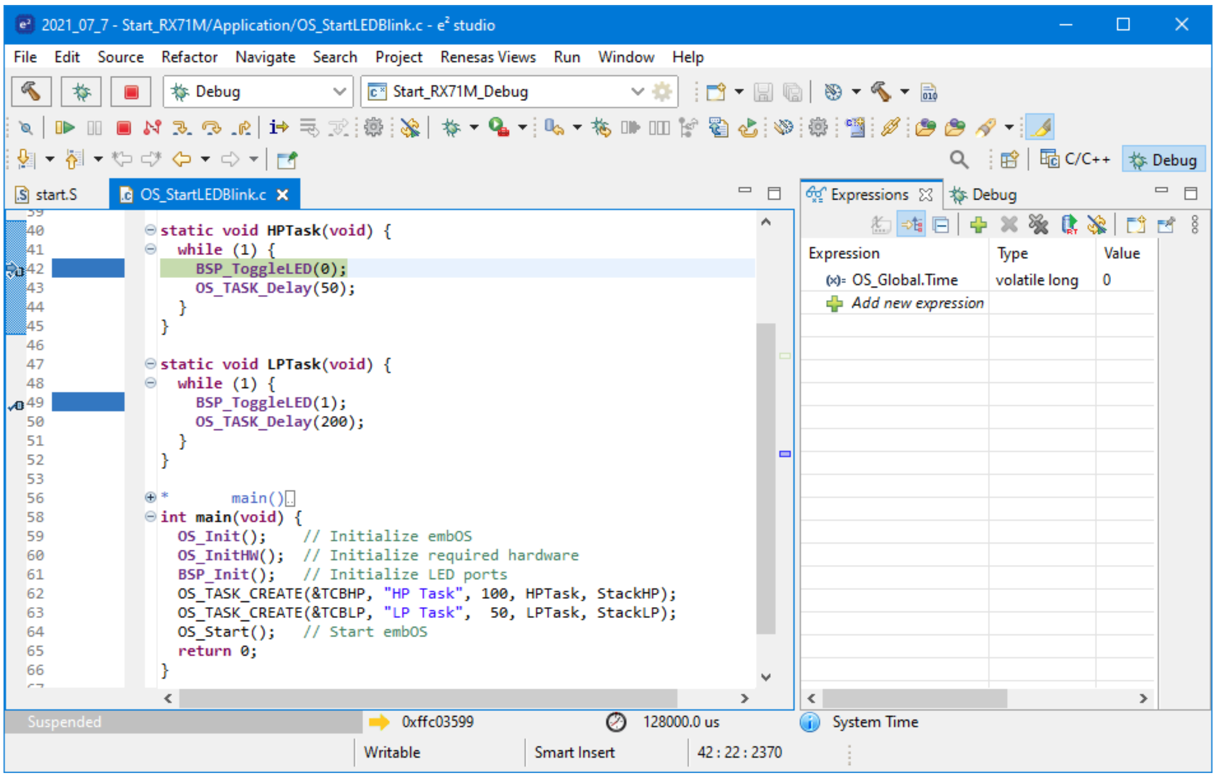
```
static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_TASK_Delay(50);
  }
}

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;
}
```

Expression	Type	Value
OS_Global.Time	volatile long	0
Add new expression		

```
Suspended    0xffc035c1    214000.0 us    System Time
Writable    Smart Insert    59 : 1 : 2620
```

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.

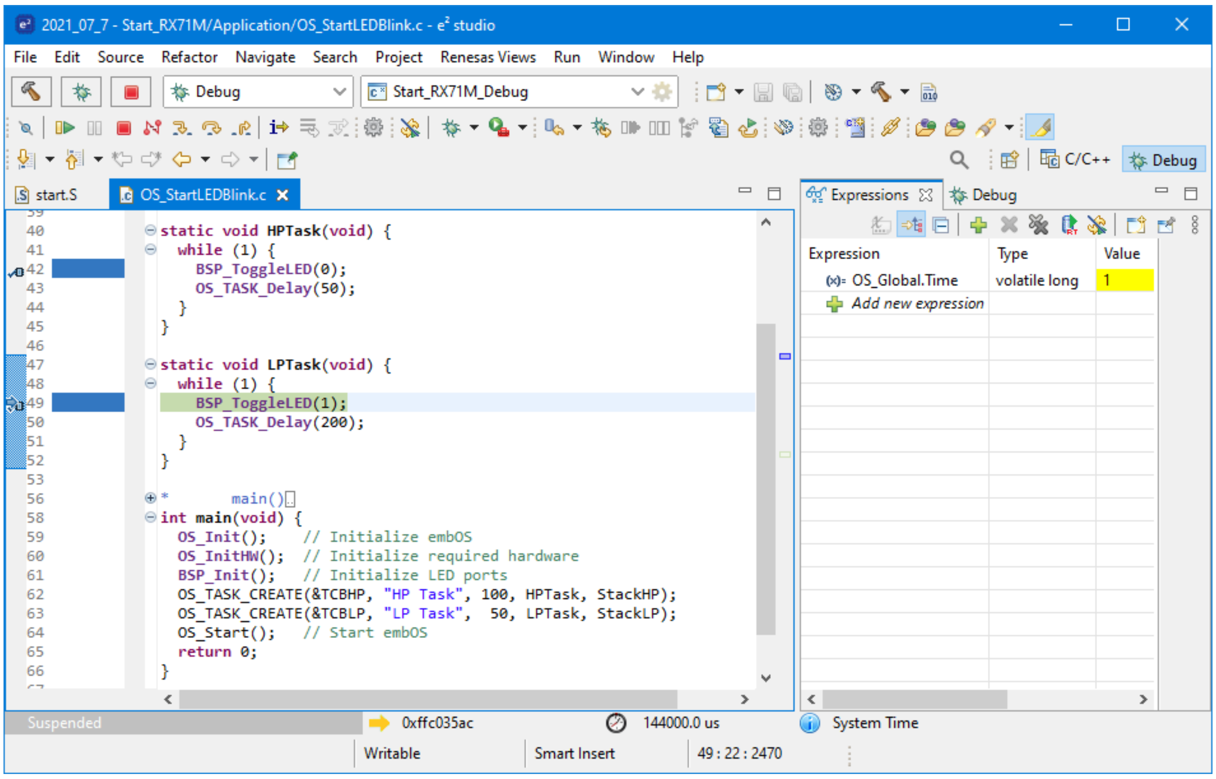

2021_07_7 - Start_RX71M/Application/OS_StartLEDBlink.c - e² studio
File Edit Source Refactor Navigate Search Project Renesas Views Run Window Help
Debug
Start_RX71M_Debug
C/C++
Debug
start.S
OS_StartLEDBlink.c
Expressions
Debug

```
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;
}
```

Expression	Type	Value
OS_Global.Time	volatile long	0
Add new expression		

```
Suspended    0xffc03599    128000.0 us    System Time
Writable    Smart Insert    42 : 22 : 2370
```

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


2021_07_7 - Start_RX71M/Application/OS_StartLEDBlink.c - e² studio

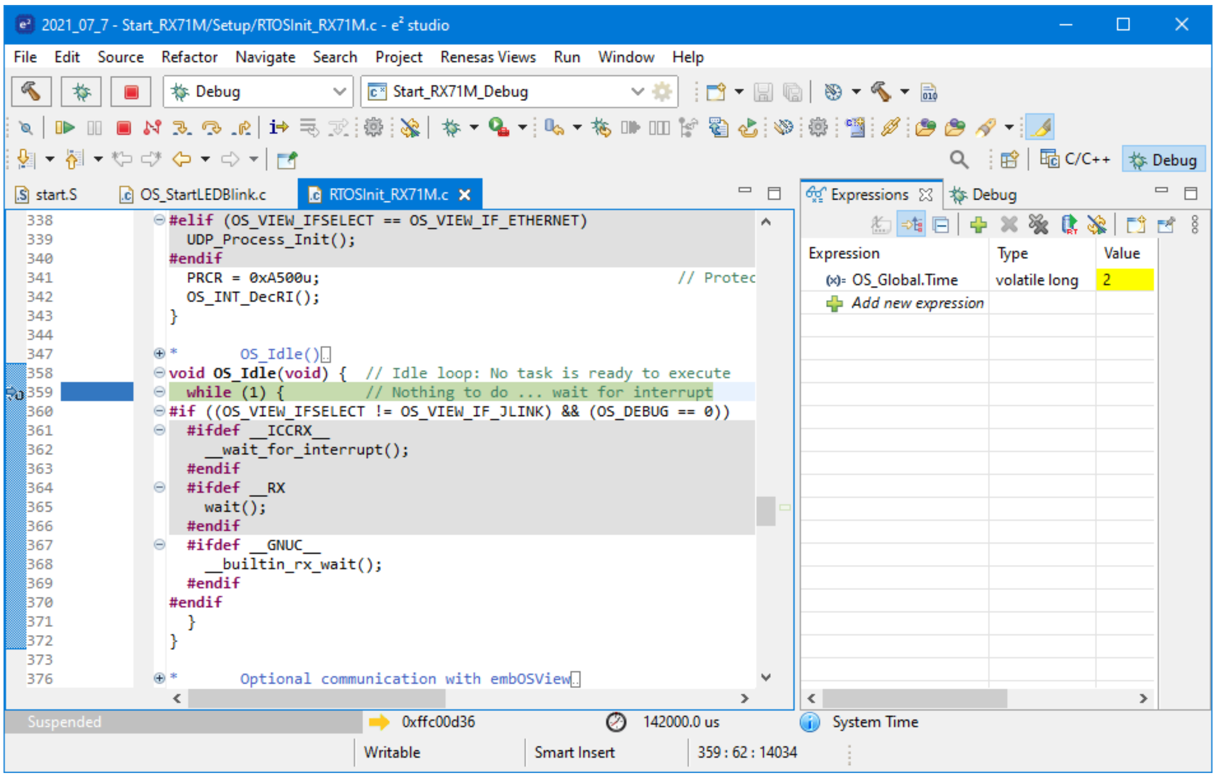
```
static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_TASK_Delay(50);
  }
}

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;
}
```

Expression	Type	Value
OS_Global.Time	volatile long	1
Add new expression		

```
Suspended    0xffc035ac    144000.0 us    System Time
Writable    Smart Insert    49 : 22 : 2470
```

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()`.


2021_07_7 - Start_RX71M/Setup/RTOSInit_RX71M.c - e² studio

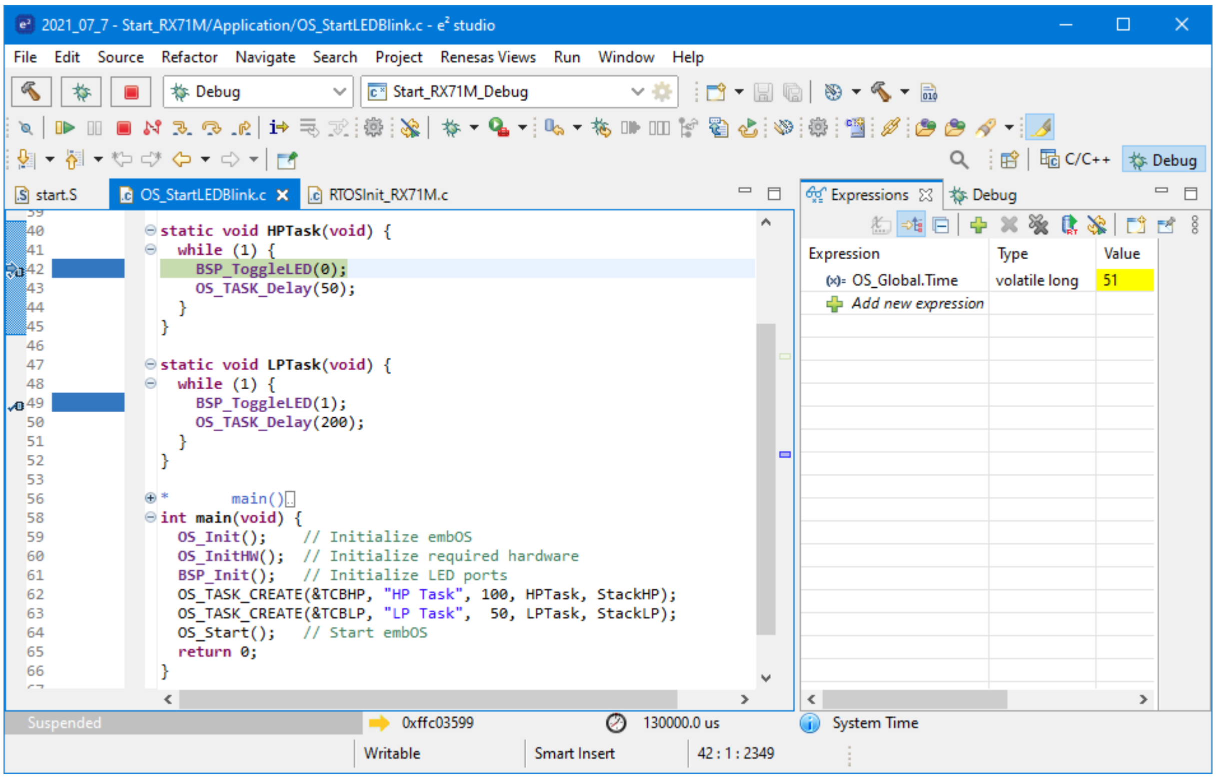
```
#elif (OS_VIEW_IFSELECT == OS_VIEW_IF_ETHERNET)
  UDP_Process_Init();
#endif
  PRCR = 0xA500u;                                    // Protec
  OS_INT_DecRI();
}

*       OS_Idle()
void OS_Idle(void) {  // Idle loop: No task is ready to execute
  while (1) {         // Nothing to do ... wait for interrupt
#if ((OS_VIEW_IFSELECT != OS_VIEW_IF_JLINK) && (OS_DEBUG == 0))
  #ifdef __ICCRX__
    __wait_for_interrupt();
  #endif
  #ifdef __RX
    wait();
  #endif
  #ifdef __GNUC__
    __builtin_rx_wait();
  #endif
#endif
  }
}

*       Optional communication with embOSView
```

Expression	Type	Value
OS_Global.Time	volatile long	2
Add new expression		

```
Suspended    0xffc00d36    142000.0 us    System Time
Writable    Smart Insert    359 : 62 : 14034
```

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.


2021_07_7 - Start_RX71M/Application/OS_StartLEDBlink.c - e² studio
File Edit Source Refactor Navigate Search Project Renesas Views Run Window Help
Debug
Start_RX71M_Debug
C/C++
Debug
start.S
OS_StartLEDBlink.c
RTOSInit_RX71M.c
Expressions
Debug

```
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;
}
```

Expression	Type	Value
OS_Global.Time	volatile long	51
Add new expression		

```
Suspended    0xffc03599    130000.0 us    System Time
Writable    Smart Insert    42 : 1 : 2349
```

Chapter 2

Build your own application

2.1 Introduction

This chapter provides all information to set up your own embOS project. To build your own application, you should always start with one of the supplied sample workspaces and projects. Therefore, select an embOS workspace as described in chapter *First Steps* on page 11 and modify the project to fit your needs. Using an embOS start project as starting point has the advantage that all necessary files are included and all settings for the project are already done.

2.2 Required files for an embOS

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

- **`RTOS.h`** from the directory `.\Start\Inc`. This header file declares all embOS API functions and data types and has to be included in any source file using embOS functions.
- **`RTOSInit*.c`** from one target specific `.\Start\BoardSupport\<Manufacturer>\<MCU>` subfolder. It contains hardware-dependent initialization code for embOS. It initializes the system timer interrupt but can also initialize or set up the interrupt controller, clocks and PLLs, the memory protection unit and its translation table, caches and so on.
- **`OS_Error.c`** from one target specific subfolder `.\Start\BoardSupport\<Manufacturer>\<MCU>`. The error handler is used only if a debug library is used in your project.
- One **embOS library** from the subfolder `.\Start\Lib`.
- Additional CPU and compiler specific files may be required according to CPU.

When you decide to write your own startup code or use a low level `init()` function, ensure that non-initialized variables are initialized with zero, according to C standard. This is required for some embOS internal variables. Your `main()` function has to initialize embOS by calling `OS_Init()` and `OS_InitHW()` prior to any other embOS functions that are called.

2.3 Change library mode

For your application you might want to choose another library. For debugging and program development you should always use an embOS debug library. For your final application you may wish to use an embOS release library or a stack check library.

Therefore you have to select or replace the embOS library in your project or target:

- If your selected library is already available in your project, just select the appropriate project configuration.
- To add a library, you may add the library to the existing Lib group. Exclude all other libraries from your build, delete unused libraries or remove them from the configuration.
- Check and set the appropriate `OS_LIBMODE_*` define as preprocessor option and/or modify the `OS_Config.h` file accordingly.

2.4 Select another CPU

embOS contains CPU-specific code for various CPUs. Manufacturer- and CPU-specific sample start workspaces and projects are located in the subfolders of the `.\Start\BoardSupport` directory. To select a CPU which is already supported, just select the appropriate workspace from a CPU-specific folder.

If your CPU is currently not supported, examine all `RTOSInit.c` files in the CPU-specific subfolders and select one which almost fits your CPU. You may have to modify `OS_InitHW()`, the interrupt service routines for the embOS system tick timer and the low level initialization.

Chapter 3

CPU and compiler specifics

3.1 Memory models

The GNURX compiler provides only one memory model.

3.2 Heap memory

embOS for Renesas RX and GNU provides an own implementation of the function `sbrk()`. This function is required if the standard library Newlib is used without default libraries, i.e. linked with "`-nostdlib`".

`sbrk()` is implemented in the file `sbrk.c` which is located in the `Setup` directory of each BSP. This implementation requires specific linker symbols in order to link properly.

Symbol	Description
`__heap_start`	Contains the start address (lower address) of the heap
`__heap_end`	Contains the end address (higher address) of the heap

The heap section, its size, location and the linker symbols are defined in the linker script. The heap can be defined like this in the linker script file:

```
  .heap _end (NOLOAD) :
  {
    __heap_start = .;
    . = ORIGIN(RAM) + LENGTH(RAM);
    __heap_end = .;
  } > RAM
```

The `_end` symbol contains the end address of the RAM's content, so that the remaining memory, from `_end` to the end of the RAM memory block, can be used for the heap. The heap section has to be placed after the linker symbol `_end` has been defined.

Chapter 4

Interrupts

4.1 What happens when an interrupt occurs?

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

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

4.2 Defining interrupt handlers in C

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

For a detailed description on how to define an interrupt routine in “C”, refer to the GNURX documentation.

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

For details about interrupt priorities, refer to chapter *Interrupt priorities* on page 23.

Example

Simple interrupt routines:

```
//
// Interrupt handler NOT using embOS functions
//
void __attribute__ ((interrupt)) IntHandlerTimer(void){
  IntCnt++;
}
//
// Interrupt function using embOS functions
//
void __attribute__ ((interrupt)) OS_ISR_Tick(void) {
  OS_INT_EnterNestable();
  OS_TICK_Handle();
  OS_INT_LeaveNestable();
}
```

4.2.1 Interrupt vector table

The vector table is written in “C”. It is located in the file `vects.c` which is part of the BSPs. Please make sure that the vector tables has an entry for each of your interrupt handlers.

4.3 Interrupt priorities

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

4.3.1 Zero latency interrupts

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

Zero latency interrupts are never disabled by embOS.

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

4.3.2 embOS interrupts

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

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

4.3.3 OS_INT_SetPriorityThreshold()

Description

`OS_INT_SetPriorityThreshold()` is used to set the IPL threshold between zero latency interrupts and lower priority embOS interrupts.

Prototype

```
void OS_INT_SetPriorityThreshold(unsigned int Priority);
```

Parameters

Parameter	Description
`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 for RX CPUs with 8 priority levels. 1 ≤ `Priority` ≤ 15 for RX CPUs with 16 priority levels.

Additional information

The IPL threshold for zero latency interrupts is set to 4 by default. This means, all interrupts with IPLs from 5 up to the maximum CPU specific IPL will never be disabled by embOS. To modify the default priority limit, `OS_INT_SetPriorityThreshold()` should be called before embOS was started. In the sample start projects, `OS_INT_SetPriorityThreshold()` is not called. The start projects use the default IPL threshold.

Interrupts running above the IPL threshold must not call any embOS function.

To disable zero latency interrupts at all, the IPL threshold may be set to the highest interrupt priority level supported by the CPU. Note that the maximum allowed parameter is device dependent. The function will not check whether the device specific limit is exceeded. It is the user’s responsibility not to use a value above 7 for CPUs which do not support more than 8 priority levels.

4.4 Interrupt nesting

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

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

4.5 Interrupt-stack switching

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

4.6 Fast interrupt, RX specific

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

4.7 Non Maskable Interrupt, NMI

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

Chapter 5

Libraries

5.1 Naming conventions for prebuilt libraries

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

The libraries are named as follows:

`libos<CpuMode><Endianness><LibMode>.a`

Parameter	Meaning	Values
`CpuMode`	Specifies the RX core	RXv2: RXv2 core : RXv1 core otherwise
`Endianness`	Byte order	B: Big endian L: Little endian
`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

`libosLDP.a` is the library for a project using little endian mode with debug and profiling support.

Chapter 6

RTT and SystemView

6.1 SEGGER Real Time Transfer

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

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

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

6.2 SEGGER SystemView

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

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

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

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

Note

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

Chapter 7

Stacks

7.1 Stack pointers

RX CPUs have two stack pointers, the user stack pointer (USP) and the interrupt stack pointer (ISP). The U-flag in the PSW regitser selects which stack pointer is used. During execution of tasks, software timers or the embOS scheduler, the U-flag is set and the USP is used. When an interrupt occurs, the U-flag is cleared and the ISP is used. On interrupt exit, the stack pointer is switched to the previous stack pointer.

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

7.3 System stack

The system stack is the stack that is used by embOS for the scheduler and software timers. When `OS_Init()` is called, embOS switches to the user stack pointer and uses its stack as the system stack. The minimum system stack size required by embOS is about 128 bytes (stack check & profiling build). However, since the system stack is also used by the application before the start of multitasking, and because software timers also use the system stack, the actual stack requirements depend on the application.

The size of the system stack can be changed by modifying the linker file. We recommend a minimum stack size of 256 bytes for the system stack.

7.4 Interrupt stack

Additional software stack switching in interrupts as for other CPUs is not necessary for the RX. If an interrupt occurs, the RX clears the U-flag and switches automatically to the ISP. 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 size of the task stack does not have to be increased for interrupt routines.

7.5 Stack section

The user and interrupt stacks and their size are defined by the linker script file. In order for embOS to link properly, the linker file needs to provide specific symbols that mark the start and the end of the user and interrupt stacks.

Symbol	Description
`_ustack_start`	Contains the start address (lower address) of the user stack
`_ustack`	Contains the end address (higher address) of the user stack
`_istack_start`	Contains the start address (lower address) of the interrupt stack
`_istack`	Contains the end address (higher address) of the interrupt stack

An example implementation of the user and interrupt stacks in the linker script file could look like this:

```
...
.ustack :
{
  . = ALIGN(8);
  _ustack_start = .;
  . = . + 0x200;
  _ustack = .;
} > RAM
.istack :
{
  . = ALIGN(8);
  _istack_start = .;
  . = . + 0x200;
  _istack = .;
} > RAM
...
```

Chapter 8

Technical data

8.1 Resource Usage

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

Module	Memory type	Memory requirements
embOS kernel	ROM	~1700 bytes
embOS kernel	RAM	~110 bytes
Task control block	RAM	36 bytes
Software timer	RAM	20 bytes
Task event	RAM	0 bytes
Event object	RAM	16 bytes
Mutex	RAM	16 bytes
Semaphore	RAM	8 bytes
RWLocks	RAM	28 bytes
Mailbox	RAM	24 bytes
Queue	RAM	32 bytes
Watchdog	RAM	12 bytes
Fixed Block Size Memory Pool	RAM	32 bytes