embOS

Real-Time Operating System

CPU & Compiler specifics for RISC-V using Embedded Studio

Document: UM01069
Software Version: 4.38
Revision: 0
Date: December 5, 2017

A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Contact address

SEGGER Microcontroller GmbH & Co. KG

In den Weiden 11
D-40721 Hilden

Germany

Tel.	+49 2103-2878-0
Fax.	+49 2103-2878-28
E-mail:	support@segger.com
Internet:	www.segger.com

Manual versions

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

Print date: December 5, 2017

Software	Revision	Date	By	Description
4.38	0	171205	MC	Initial version.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler).
- The C programming language.
- The target processor.
- DOS command line.

If you feel that your knowledge of C is not sufficient, we recommend *The C Programming Language* by Kernighan and Richie (ISBN 0–13–1103628), which describes the standard in C programming and, in newer editions, also covers the ANSI C standard.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections.

Table of contents

Chapter 1

Using embOS

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

1.1 Installation

embOS is shipped as a zip-file in electronic form.

To install it, proceed as follows:

Extract the zip-file to any folder of your choice, preserving the directory structure of this file. Keep all files in their respective sub directories. Make sure the files are not read only after copying.

Assuming that you are using an IDE to develop your application, no further installation steps are required. You will find a lot of prepared sample start projects, which you should use and modify to write your application. So follow the instructions of section *First Steps* on page 10.

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

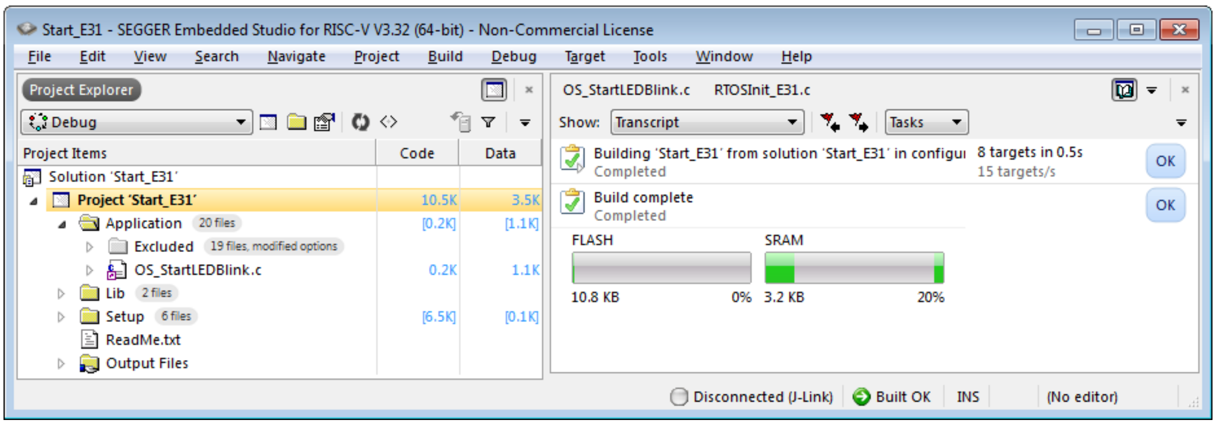
After installation of embOS you can create your first multitasking application. You have received several ready to go sample start workspaces and projects and every other files needed in the subfolder `Start`. It is a good idea to use one of them as a starting point for all of your applications. The 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\<DeviceManufactor>\<CPU>` with your IDE (for example, by double clicking it).
- Build the project. It should be built without any error or warning messages.

After generating the project of your choice, the screen should look like this:


Start_E31 - SEGGER Embedded Studio for RISC-V V3.32 (64-bit) - Non-Commercial License

Project Items	Code	Data
Solution 'Start_E31'		
Project 'Start_E31'	10.5K	3.5K
Application 20 files	[0.2K]	[1.1K]
Excluded 19 files, modified options		
OS_StartLEDBlink.c	0.2K	1.1K
Lib 2 files		
Setup 6 files	[6.5K]	[0.1K]
ReadMe.txt		
Output Files		

Building 'Start_E31' from solution 'Start_E31' in configu 8 targets in 0.5s
Completed 15 targets/s

Build complete
Completed

FLASH 10.8 KB 0%
SRAM 3.2 KB 20%

Disconnected (J-Link) Built OK INS (No editor)

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.

```c
/*********************************************************************
*                 SEGGER Microcontroller GmbH & Co. KG               *
*                         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_Delay(50);
  }
}

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

/*********************************************************************
*
*       main()
*/
int main(void) {
  OS_InitKern();   // Initialize embOS
  OS_InitHW();     // Initialize required hardware
  BSP_Init();      // Initialize LED ports
  OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
  OS_Start();      // Start embOS
  return 0;
}

/*************************** End of file ****************************/
```

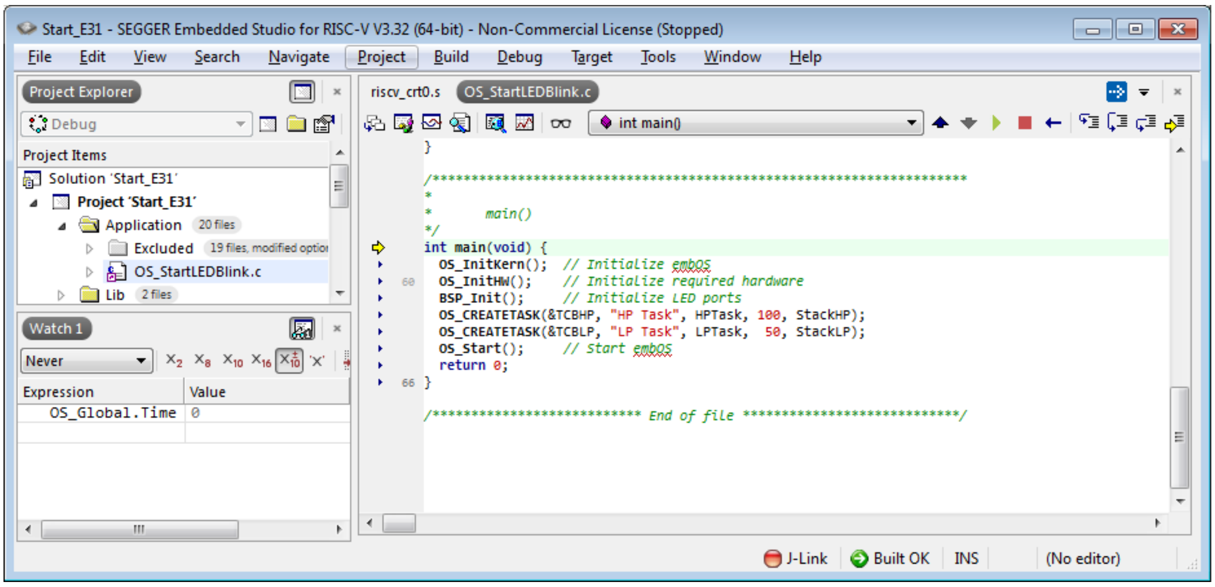
1.4 Stepping through the sample application

When starting the debugger, you will see the `main()` function (see example screen shot below). The `main()` function appears as long as project option `Run to main` is selected, which it is enabled by default. Now you can step through the program.

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

`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.

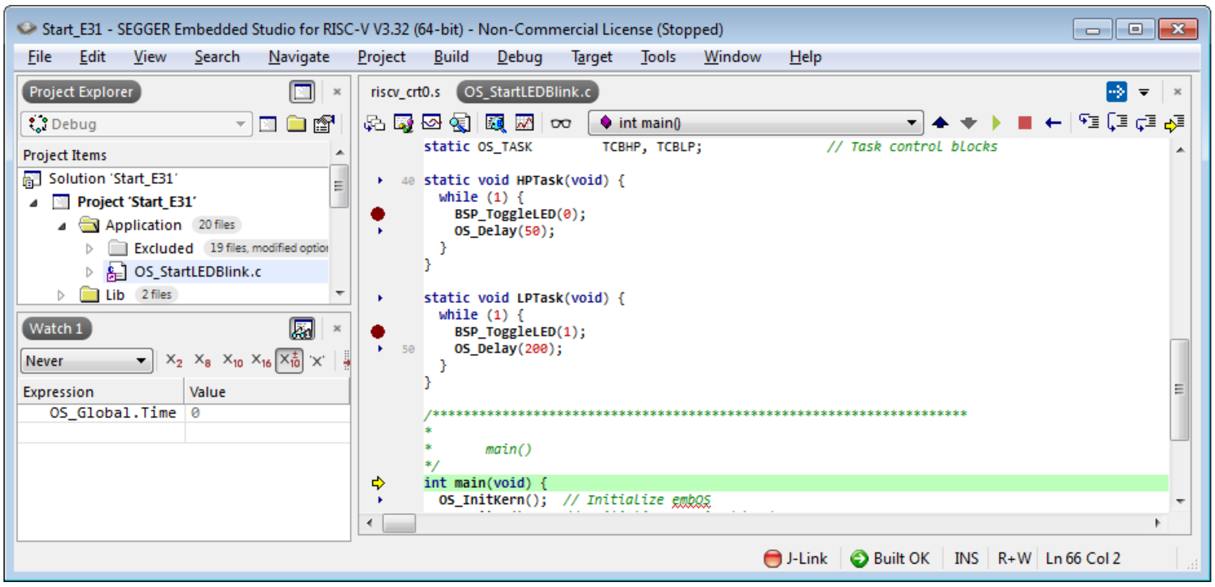
`OS_Start()` should be the last line in `main()`, because it starts multitasking and does not return.


Start_E31 - SEGGER Embedded Studio for RISC-V V3.32 (64-bit) - Non-Commercial License (Stopped)
File Edit View Search Navigate Project Build Debug Target Tools Window Help
Project Explorer
Debug
Project Items
Solution 'Start_E31'
Project 'Start_E31'
Application 20 files
Excluded 19 files, modified option
OS_StartLEDBlink.c
Lib 2 files
Watch 1
Never
riscv_crt0.s
OS_StartLEDBlink.c
int main()
J-Link
Built OK
INS
(No editor)

Expression	Value
OS_Global.Time	0

```
}

/*********************************************************************
*
*       main()
*/
int main(void) {
  OS_InitKern();  // Initialize embOS
  OS_InitHW();    // Initialize required hardware
  BSP_Init();     // Initialize LED ports
  OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
  OS_Start();     // Start embOS
  return 0;
}

/*************************** End of file ****************************/
```

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

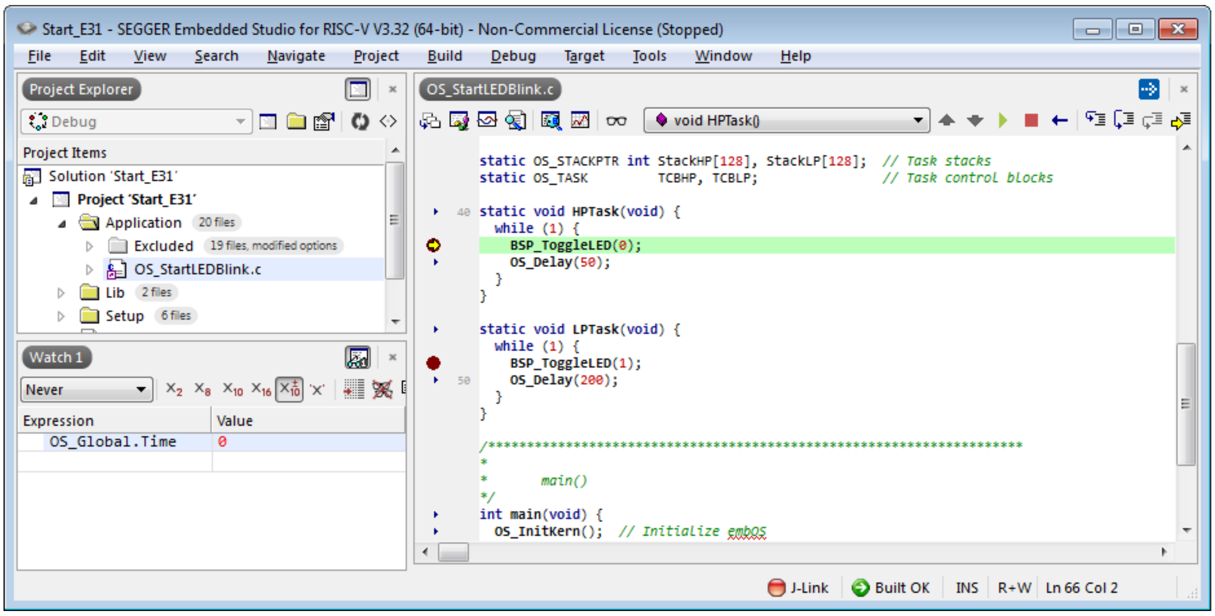


Expression	Value
OS_Global.Time	0

```
static OS_TASK         TCBHP, TCBLP;                // Task control blocks

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

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

/*********************************************************************
*
*       main()
*/
int main(void) {
  OS_InitKern();  // Initialize embOS
```

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

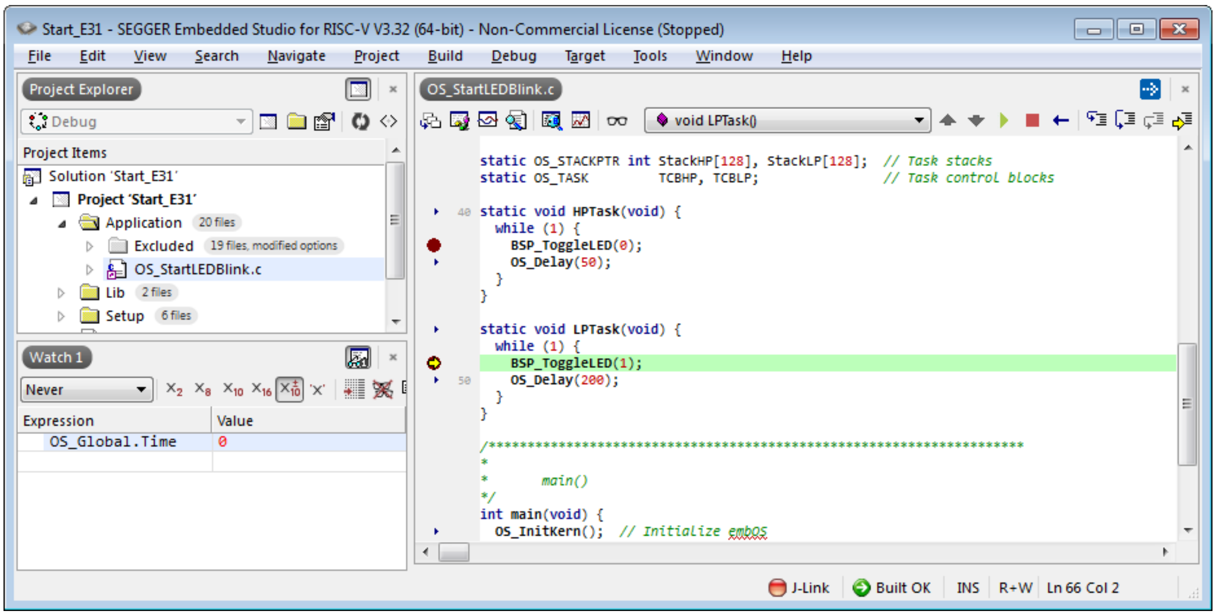
Click GO, step over `OS_Start()`, or step into `OS_Start()` in disassembly mode until you reach the highest priority task.



Expression	Value
OS_Global.Time	0

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

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

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

/*********************************************************************
*
*       main()
*/
int main(void) {
  OS_InitKern();  // Initialize embOS
```

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

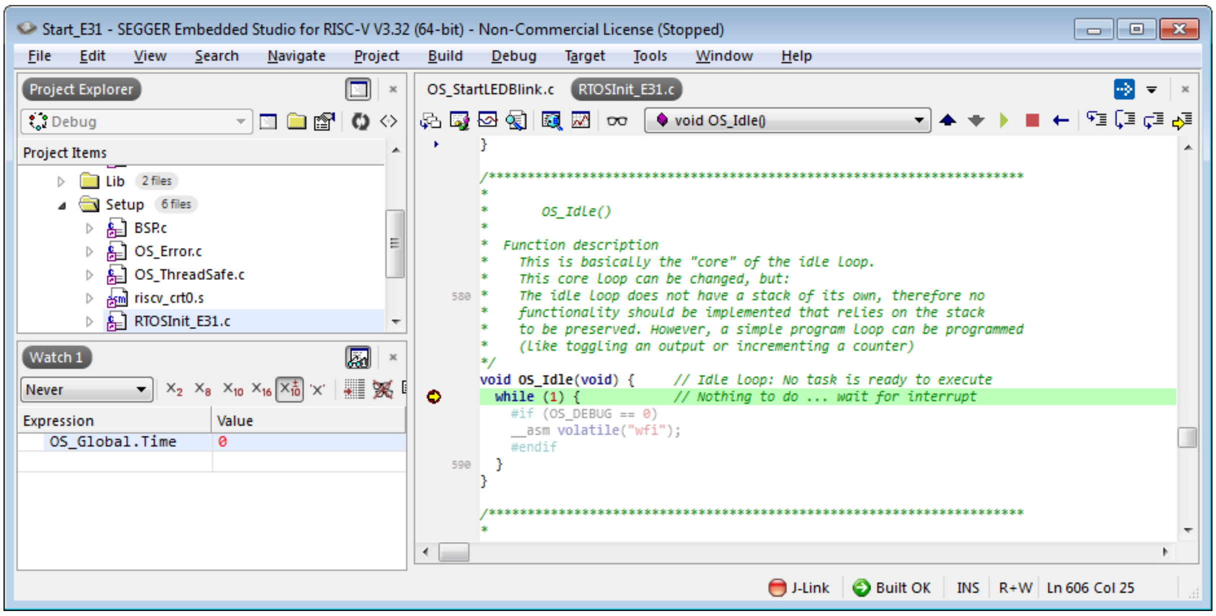


Expression	Value
OS_Global.Time	0

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

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

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

/*********************************************************************
*
*       main()
*/
int main(void) {
  OS_InitKern();  // Initialize embOS
```

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

You will arrive there when you step into the `OS_Delay()` function in disassembly mode. `OS_Idle()` is part of `RTOSInit.c`. You may also set a breakpoint there before stepping over the delay in `LPTask()`.

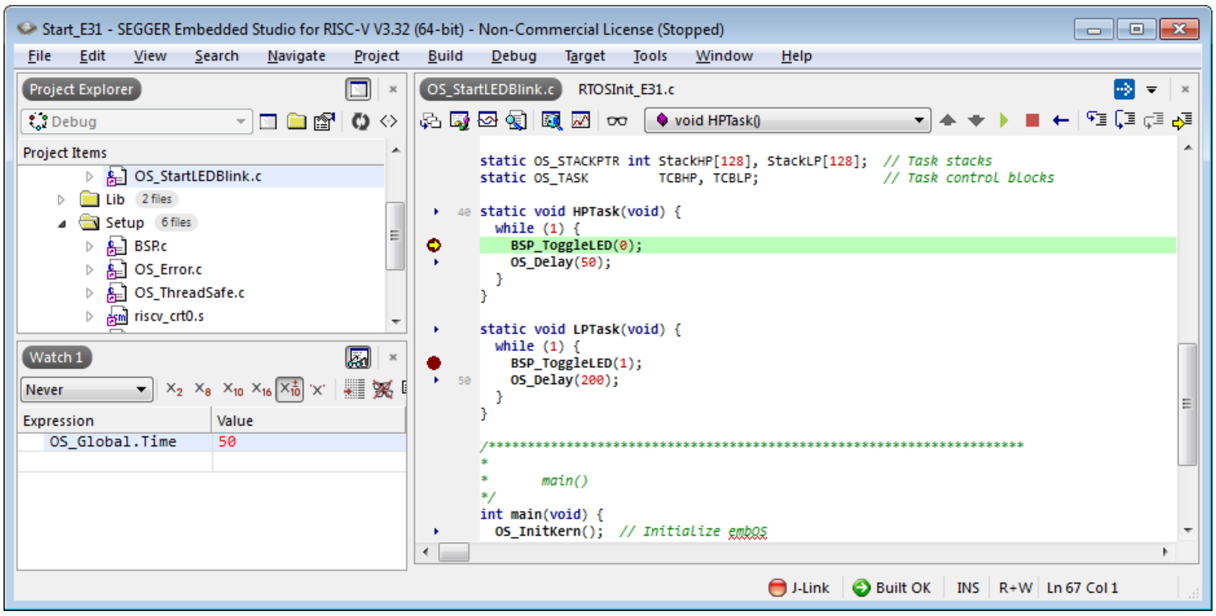


Expression	Value
OS_Global.Time	0

```
}

/*********************************************************************
*
*       OS_Idle()
*
*  Function description
*    This is basically the "core" of the idle loop.
*    This core loop can be changed, but:
*    The idle loop does not have a stack of its own, therefore no
*    functionality should be implemented that relies on the stack
*    to be preserved. However, a simple program loop can be programmed
*    (like toggling an output or incrementing a counter)
*/
void OS_Idle(void) {     // Idle loop: No task is ready to execute
  while (1) {            // Nothing to do ... wait for interrupt
    #if (OS_DEBUG == 0)
    __asm volatile("wfi");
    #endif
  }
}

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

If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay.

As can be seen by the value of embOS timer variable `OS_Global.Time`, shown in the Watch window, `HPTask()` continues operation after expiration of the 50 system tick delay.



Expression	Value
OS_Global.Time	50

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

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

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

/*********************************************************************
*
*       main()
*/
int main(void) {
  OS_InitKern();  // Initialize embOS
```

Chapter 2

Build your own application

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

2.1 Introduction

To build your own application, you should always start with one of the supplied sample workspaces and projects. Therefore, select an embOS workspace as described in chapter *First Steps* on page 10 and modify the project to fit your needs. Using an embOS start project as starting point has the advantage that all necessary files are included and all settings for the project are already done.

2.2 Required files for an embOS

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

- `RTOS.h` from subfolder `Inc\`. This header file declares all embOS API functions and data types and has to be included in any source file using embOS functions.
- `RTOSInit.c` from one target specific `BoardSupport\<Manufacturer>\<MCU>` subfolder. It contains hardware-dependent initialization code for embOS. It initializes the system timer interrupt and optional communication for embOSView via UART or JTAG.
- `OS_Error.c` from one target specific subfolder `BoardSupport\<Manufacturer>\<MCU>`. The error handler is used if any debug library is used in your project.
- One embOS library from the subfolder `Lib\`.
- Additional CPU and compiler specific files may be required according to CPU.

When you decide to write your own startup code or use a low level `init()` function, ensure that non-initialized variables are initialized with zero, according to C standard. This is required for some embOS internal variables. Your `main()` function has to initialize embOS by a call of `OS_InitKern()` and `OS_InitHW()` prior any other embOS functions are called. You should then modify or replace the `OS_StartLEDBlink.c` source file in the subfolder `Application\`.

2.3 Change library mode

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

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

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

2.4 Select another CPU

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

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

Chapter 3

Libraries

This chapter includes CPU-specific information such as CPU-modes and available libraries.

3.1 Naming conventions for prebuilt libraries

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

The libraries are named as follows: `libos_rv<Arch>_<LibMode>.a`

Parameter	Meaning	Values
`Arch`	Specifies the RISC-V ISA	32imac: RV32I with ’M’, ’A’ and ’C’ extensions
`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

`libos_rv32imac_dp.a` is the library for a project using an RV32IMAC core with debug and profiling support.

Chapter 4

CPU and compiler specifics

4.1 Standard system libraries

embOS for Cortex-M and Segger Embedded Studio may be used with standard system libraries. embOS delivers the file `OS_ThreadSafe.c` which includes hook functions to make e.g. the heap management functions thread safe.

Chapter 5

Interrupts

5.1 What happens when an interrupt occurs?

- A hart receives an interrupt request.
- As soon as interrupts are globally enabled in *mstatus.MIE* and the specific interrupt source is enabled in *mie.MxIE*, the interrupt is accepted and executed.
- The value of *mstatus.MIE* is copied into *mstatus.MPIE*, then *mstatus.MIE* is cleared, effectively disabling interrupts.
- The current pc is copied into the *mepc* register, and then *pc* is set to the value of *mtvec*. In case vectored interrupts are enabled, *pc* is set to *mtvec.BASE + 4 * exception code*.
- The privilege mode prior to the interrupt is encoded in *mstatus.MPP*.
- At this point, control is handed over to software in the interrupt handler with interrupts disabled.
  Interrupts can be re-enabled by explicitly setting *mstatus.MIE* (or by executing an *MRET* instruction to exit the handler).
- The low-level interrupt handler, `trap_entry()` in direct mode or the appropriate vector in vectored mode, saves the caller-save register on stack.
- The high-level interrupt handler, implemented in ’C’, is called and serves the interrupt before returning to the low-level interrupt handler.
- The low-interrupt handler restores the caller-save register from stack.
- The low-interrupt handler ends by executing an *MRET* instruction.
- The privilege mode is set to the value encoded in *mstatus.MPP*.
- The value of *mstatus.MPIE* is copied into *mstatus.MIE*.
- The *pc* is set to the value of *mepc*, thus continuing the interrupted function.

5.2 RISC-V interrupt sources

RISC-V harts can have both local and external interrupt sources:

Local interrupt sources are those that do not pass through the *Platform-Level Interrupt Controller (PLIC)*. These include the standard software and timer interrupts for each privilege level, and an optional number of further machine local interrupts.

External interrupt sources, on the other hand, are prioritized and distributed by the platform-specific PLIC implementation.

Local ISR handling may be performed in direct mode, in which all traps are distributed through `OS_IRQ_Handler()`. Alternatively, local ISR handling may also be performed in vectored mode. In this case, a ROM vector table is used to distribute traps.

5.3 Defining interrupt handlers in C

Interrupt handlers for RISC-V cores are written as normal C-functions which do not take parameters and do not return any value. Interrupt handler which call an embOS function need a prolog and epilog function as described in the generic manual and in the examples below.

Example

Simple interrupt routine:

```
static void _Systick(void) {
  OS_EnterNestableInterrupt();  // Inform embOS that interrupt code is running
  OS_HandleTick();              // May be interrupted
  OS_LeaveNestableInterrupt();  // Inform embOS that interrupt handler is left
}
```

5.4 Interrupt-stack switching

Interrupt-stack switching is currently not supported with embOS. Interrupts will execute on the stack of the interrupted task, if any, and will otherwise use the system stack.

5.5 Zero latency interrupts

Zero latency interrupts are currently not supported.

5.6 Interrupt priorities

Interrupts are prioritized as follows, in decreasing order of priority:

Trap name
Machine external interrupts (with configurable external priority)
Machine software interrupts
Machine timer interrupts
Synchronous trap

Machine external interrupts are furthermore distributed by the PLIC according to an additional priority, with a platform-specific maximum number of supported priority levels: Generically, the priority value ’0’ is reserved, and further interrupt priorities increase with increasing values.

Each external interrupt source has an interrupt priority held in a platform-specific memory-mapped register. Each interrupt target has an associated priority threshold, held in a platform-specific memory-mapped register. Only active interrupts that have a priority strictly greater than the threshold will cause a interrupt notification to be sent to the target.

Further (optional) machine local interrupts run at a platform-dependant priority level. For example, with RISC-V Coreplex IP, machine local interrupts take precendence over any other interrupt source, and are themselves prioritized by their ID. A comprehensive priority table for local interrupts on one platform, in decreasing order of priority, could therefore read as follows:

Trap name
Machine local interrupt 15
Machine local interrupt 14
…
Machine local interrupt 1
Machine local interrupt 0
Machine external interrupts (with configurable external priority)
Machine software interrupts
Machine timer interrupts
Synchronous trap

5.7 Interrupt handling

5.7.1 Local Interrupt handling

To handle local interrupts, embOS offers the following functions:

Function	Description
`OS_RISCV_ISR_Disable()`	Disables the specified local interrupt source
`OS_RISCV_ISR_Enable()`	Enables the specified local interrupt source
`OS_RISCV_ISR_Init()`	Configures RAM vector table address (used in direct mode only)
`OS_RISCV_ISR_InstallHandler()`	Installs a local interrupt handler (used in direct mode only)

Local interrupt sources should be specified using the following enumeration:

Local IRQ type	Numeric value
IRQ_M_SOFTWARE	3
IRQ_M_TIMER	7
IRQ_M_EXTERNAL	11
IRQ_LOCAL0	16
IRQ_LOCAL1	17
IRQ_LOCAL2	18
IRQ_LOCAL3	19
IRQ_LOCAL4	20
IRQ_LOCAL5	21
IRQ_LOCAL6	22
IRQ_LOCAL7	23
IRQ_LOCAL8	24
IRQ_LOCAL9	25
IRQ_LOCAL10	26
IRQ_LOCAL11	27
IRQ_LOCAL12	28
IRQ_LOCAL13	29
IRQ_LOCAL14	30
IRQ_LOCAL15	31

5.7.2 OS_RISCV_ISR_Disable()

Description

`OS_RISCV_ISR_Disable()` is used to disable the specified local interrupt source.

Prototype

```
void OS_RISCV_ISR_Disable (RISCV_IRQ ISRIndex);
```

Parameters

Parameter	Description
`ISRIndex`	Interrupt index

5.7.3 OS_RISCV_ISR_Enable()

Description

`OS_RISCV_ISR_Enable()` is used to enable the specified local interrupt source.

Prototype

```
void OS_RISCV_ISR_Enable (RISCV_IRQ ISRIndex);
```

Parameters

Parameter	Description
ISRIndex	Interrupt index

Additional information

`OS_RISCV_CLINT_DisableISR()` is not implemented as a function, but as a macro.

5.7.4 OS_RISCV_ISR_Init()

Description

`OS_RISCV_ISR_Init()` is used to configure the RAM vector table address for local interrupts. Since a RAM vector table is used in direct mode only, this function mustn't be called when using vectored mode.

Prototype

```
void OS_RISCV_ISR_Init (OS_U8           NumInterrupts,
                        OS_ISR_HANDLER* TableBaseAddr[]);
```

Parameters

Parameter	Description
`NumInterrupts`	Number of supported interrupt sources
`TableBaseAddr`	RAM vector table base address

5.7.5 OS_RISCV_ISR_InstallHandler()

Description

`OS_RISCV_ISR_InstallHandler()` is used to install the specified local interrupt handler in the RAM vector table. Since a RAM vector table is used in direct mode only, this function mustn't be called when using vectored mode.

Prototype

```
OS_ISR_HANDLER* OS_RISCV_ISR_InstallHandler (RISCV_IRQ       ISRIndex,
                                             OS_ISR_HANDLER* pISRHandler);
```

Parameters

Parameter	Description
`ISRIndex`	Interrupt index
`pISRHandler`	Address of interrupt handler

Return value

`OS_ISR_HANDLER*`: Address of the previously installed interrupt handler, or `NULL` if not applicable.

5.7.6 External Interrupt handling

To handle external interrupts (on the Coreplex IP implementation of the Platform-Level Interrupt Controller), embOS offers the following functions:

Function	Description
OS_RISCV_COREPLEX_ISR_Claim()	Retrieves the ID of highest-priority pending external interrupt and clears pending condition
OS_RISCV_COREPLEX_ISR_Complete()	Notifies PLIC of ISR completion
OS_RISCV_COREPLEX_ISR_Disable()	Disables the specified external interrupt source
OS_RISCV_COREPLEX_ISR_Enable()	Enables the specified external interrupt source
OS_RISCV_COREPLEX_ISR_GetPriority()	Returns the current interrupt priority for the specified interrupt source
OS_RISCV_COREPLEX_ISR_GetThreshold()	Returns the current interrupt priority threshold
OS_RISCV_COREPLEX_ISR_Init()	Configures PLIC base address and RAM vector table address
OS_RISCV_COREPLEX_ISR_InstallHandler()	Installs an external interrupt handler
OS_RISCV_COREPLEX_ISR_SetPriority()	Sets the priority of the specified external interrupt
OS_RISCV_COREPLEX_ISR_SetThreshold()	Configures the IRQ threshold, masking lower-priority external interrupts

5.7.7 OS_RISCV_COREPLEX_ISR_Claim()

Description

`OS_RISCV_COREPLEX_ISR_Claim()` is used to retrieve the ID of the highest-priority pending external interrupt. Clears the corresponding source's pending bit.

Prototype

```
OS_U32 OS_RISCV_COREPLEX_ISR_Claim (void);
```

Return value

`OS_U32`: Interrupt index

5.7.8 OS_RISCV_COREPLEX_ISR_Complete()

Description

`OS_RISCV_COREPLEX_ISR_Complete()` is used to signal ISR completion to the PLIC.

Prototype

```
void OS_RISCV_COREPLEX_ISR_Complete (OS_U32 ISRIndex);
```

Parameters

Parameter	Description
ISRIndex	Interrupt index

5.7.9 OS_RISCV_COREPLEX_ISR_Disable()

Description

`OS_RISCV_COREPLEX_ISR_Disable()` is used to disable the specified external interrupt.

Prototype

```
void OS_RISCV_COREPLEX_ISR_Disable (OS_U32 ISRIndex);
```

Parameters

Parameter	Description
`ISRIndex`	Interrupt index

5.7.10 OS_RISCV_COREPLEX_ISR_Enable()

Description

`OS_RISCV_COREPLEX_ISR_Enable()` is used to enable the specified external interrupt.

Prototype

```
void OS_RISCV_COREPLEX_ISR_Enable (OS_U32 ISRIndex);
```

Parameters

Parameter	Description
`ISRIndex`	Interrupt index

5.7.11 OS_RISCV_COREPLEX_ISR_GetPriority()

Description

`OS_RISCV_COREPLEX_ISR_GetPriority()` retrieves the current interrupt priority for the specified interrupt source.

Prototype

```
OS_U32 OS_RISCV_COREPLEX_ISR_GetPriority (OS_U32 ISRIndex);
```

Parameters

Parameter	Description
`ISRIndex`	Interrupt index

Return value

`OS_U32`: Current interrupt priority of the specified interrupt source

5.7.12 OS_RISCV_COREPLEX_ISR_GetThreshold()

Description

`OS_RISCV_COREPLEX_ISR_GetThreshold()` retrieves the current interrupt priority threshold.

Prototype

```
OS_U32 OS_RISCV_COREPLEX_ISR_GetThreshold (void);
```

Return value

`OS_U32`: Current interrupt priority threshold

5.7.13 OS_RISCV_COREPLEX_ISR_Init()

Description

`OS_RISCV_COREPLEX_ISR_Init()` is used to configure the RAM vector table base address for external interrupts.

Prototype

```
void OS_RISCV_COREPLEX_ISR_Init(OS_U32          BaseAddr,
                                OS_U16          NumInterrupts,
                                OS_U32          NumPriorities,
                                OS_ISR_HANDLER* TableBaseAddr[]);
```

Parameters

Parameter	Description
BaseAddr	Coreplex PLIC base address
NumInterrupts	Number of supported external interrupt sources
NumPriorities	Number of supported external interrupt priorities
TableBaseAddr	RAM vector table base address

5.7.14 OS_RISCV_COREPLEX_ISR_InstallHandler()

Description

OS_RISCV_COREPLEX_ISR_InstallHandler() is used to install the specified external interrupt handler in the RAM vector table.

Prototype

```
OS_ISR_HANDLER* OS_RISCV_COREPLEX_ISR_InstallHandler(OS_U32          ISRIndex,
                                                     OS_ISR_HANDLER* pISRHandler);
```

Parameters

Parameter	Description
ISRIndex	Interrupt index
pISRHandler	Address of interrupt handler

Return value

OS_ISR_HANDLER*: Address of the previously installed interrupt handler, or NULL if not applicable.

5.7.15 OS_RISCV_COREPLEX_ISR_SetPriority()

Description

`OS_RISCV_COREPLEX_ISR_SetPriority()` is used to configure the interrupt priority for the specified external interrupt.

Prototype

```
OS_U32 OS_RISCV_COREPLEX_ISR_SetPriority (OS_U32 ISRIndex,
                                          OS_U32 Prio);
```

Parameters

Parameter	Description
`ISRIndex`	Interrupt index
`Prio`	Interrupt priority

Return value

`OS_U32`: Previous priority which was assigned before

5.7.16 OS_RISCV_COREPLEX_ISR_SetThreshold()

Description

OS_RISCV_COREPLEX_ISR_SetThreshold() is used to configure the interrupt priority threshold. All priorities less than or equal to Threshold will be masked.

Prototype

```
void OS_RISCV_COREPLEX_ISR_SetThreshold (OS_U32 Threshold);
```

Parameters

Parameter	Description
Threshold	Desired interrupt priority threshold

Chapter 6

Stacks

6.1 Task stack for RISC-V

Each task uses its individual stack. The stack pointer is initialized and set every time a task is activated by the scheduler. The stack-size required for a task is the sum of the stack-size of all routines, plus a basic stack size, plus size used by exceptions.

The basic stack size is the size of memory required to store the registers of the CPU plus the stack size required by calling embOS-routines.

For the RISC-V CPUs, this minimum basic task stack size is about 170 bytes. Because any function call uses some amount of stack and every exception also pushes at least 32 bytes onto the current stack, the task stack size has to be large enough to handle one exception too. We recommend at least 256 bytes stack as a start.

6.2 System stack for RISC-V

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 (the call to `OS_Start()`), and because software timers and C-level interrupt handlers may 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 project settings. We recommend a minimum stack size of 768 bytes for the system stack.

Chapter 7

embOS Thread Script

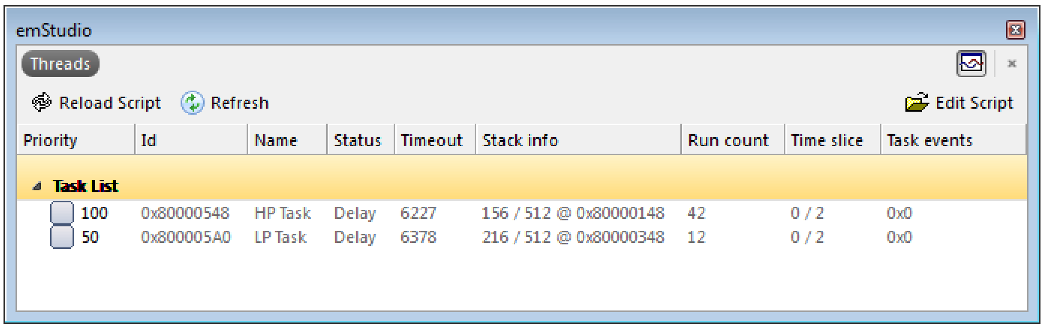
This chapter describes how to use the thread script shipped with embOS Cortex-M SES.

7.1 Introduction

A thread script is included with any start project shipped with embOS Cortex-M SES. This script may be used to display various information about the tasks currently running on the target.

7.2 How to use it

To enable the threads window, click on View in the menu bar and choose the option Threads in the sub-menu Other Windows. Alternatively, the threads window may also be enabled by pressing `[Ctrl + Alt + H]`. Now, the thread window gets updated every time the application is stopped. It should closely resemble the screenshot below:



Priority	Id	Name	Status	Timeout	Stack info	Run count	Time slice	Task events
Task List								
100	0x80000548	HP Task	Delay	6227	156 / 512 @ 0x80000148	42	0 / 2	0x0
50	0x800005A0	LP Task	Delay	6378	216 / 512 @ 0x80000348	12	0 / 2	0x0

The threads window diaplys various information about the running tasks:

- Priority - This is the priority of the task
- ID - The address of a tasks task control block
- Name - The name of the task
- State - The current state of the task
- Timeout - Time in ms till the task gets called again
- Stack - Shows the maximum usage (left) of the total stack for this task (right) in Bytes
- Run count - Shows how many times the task has been started since the last reset

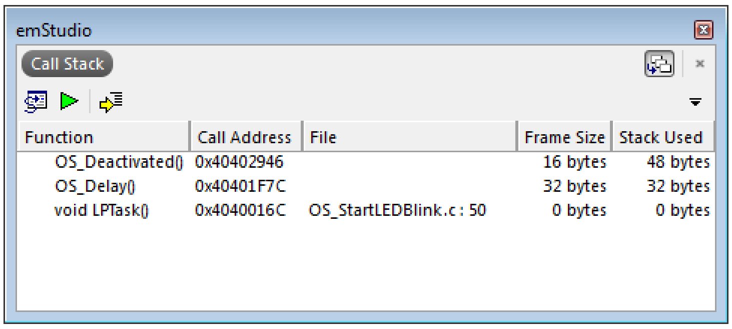
Some of this information is available in debug builds of embOS only. Using other builds, the respective entries will show “`n/a`” to indicate this.

Please note that by default the thread script is limited to display a total of 25 tasks only. This limit may be changed inside the respective project’s options (“`Debugging`” `->` “`Debugger`” `->` “`Thread Maximum`”).

7.2.1 Task sensitivity

In addition to the information displayed in the threads window, the threads script furthermore allows for the investigation of the register contents and the call stack of inactive tasks. To display this information, double click the entry of the respective task in the threads window. The register window and the call stack window will subsequently be updated to display information about the chosen task's state. To view this information, the call stack and the register window have to be enabled.

After double clicking the inactive task, the call stack window shows the last function (`_DelayUntil()`) that has been called by this task:



Function	Call Address	File	Frame Size	Stack Used
OS_Deactivated()	0x40402946		16 bytes	48 bytes
OS_Delay()	0x40401F7C		32 bytes	32 bytes
void LPTask()	0x4040016C	OS_StartLEDBlink.c : 50	0 bytes	0 bytes

Chapter 8

Technical data

This chapter lists technical data of embOS used with RISC-V CPUs.

8.1 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 2.000 bytes.

In the table below, which is for X-Release build, you can find minimum RAM size requirements for embOS resources. Note that the sizes depend on selected embOS library mode.

embOS resource	RAM [bytes]
Task control block	36
Software timer	20
Resource semaphore	16
Counting semaphore	8
Mailbox	24
Queue	36
Task event	0
Event object	12