embOS

Real-Time Operating System

CPU & Compiler specifics for RISC-V using Embedded Studio

Document: UM01069
Software Version: 5.12.0.0
Revision: 0
Date: February 8, 2021

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

Specifications written in this document are believed to be accurate, but are not guaranteed to be entirely free of error. The information in this manual is subject to change for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions. SEGGER makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. SEGGER specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH

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: February 8, 2021

Software	Revision	Date	By	Description
5.12.0.0	0	210208	MM	Added information about thread-local storage.
5.8.2.0	0	200311	MM	Added information about the ECLIC interrupt controller. Added information to the stack chapter.
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

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

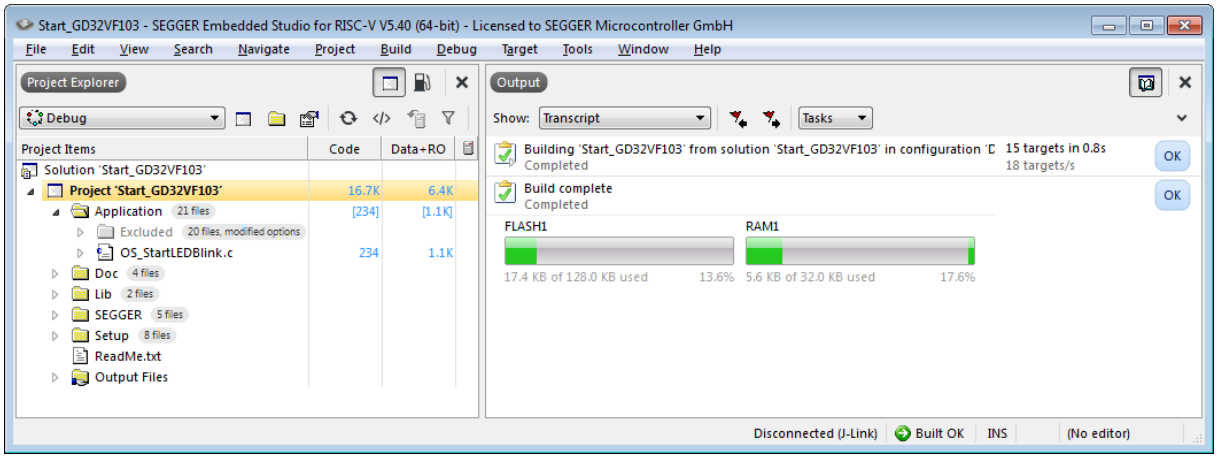
After installation of embOS you can create your first multitasking application. You have received several ready to go sample start workspaces and projects and every other files needed in the subfolder `Start`. It is a good idea to use one of them as a starting point for all of your applications. The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



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

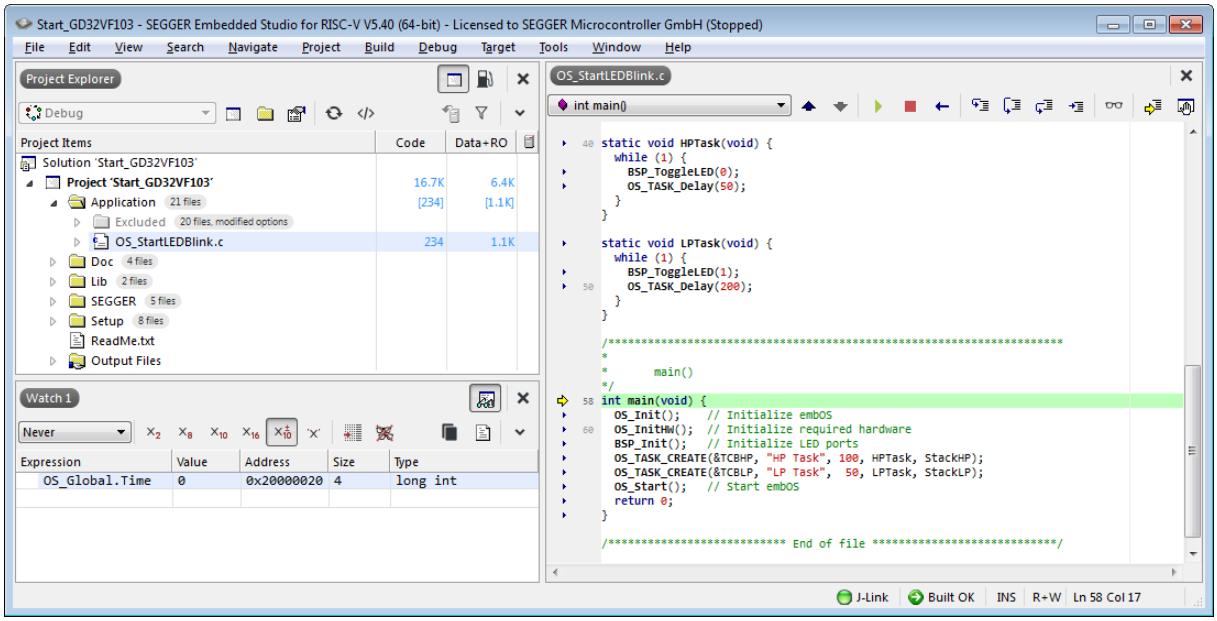

1.4 Stepping through the sample application

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.

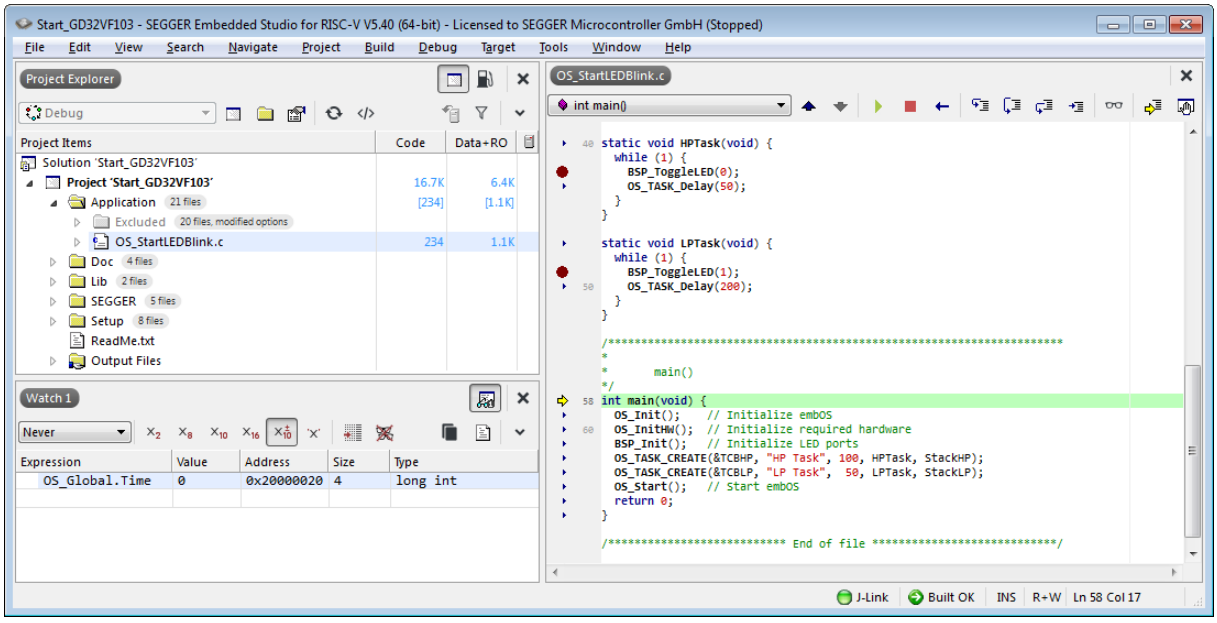
`OS_Init()` is part of the embOS library and written in assembler; you can therefore only step into it in disassembly mode. It initializes the relevant OS variables.

`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.

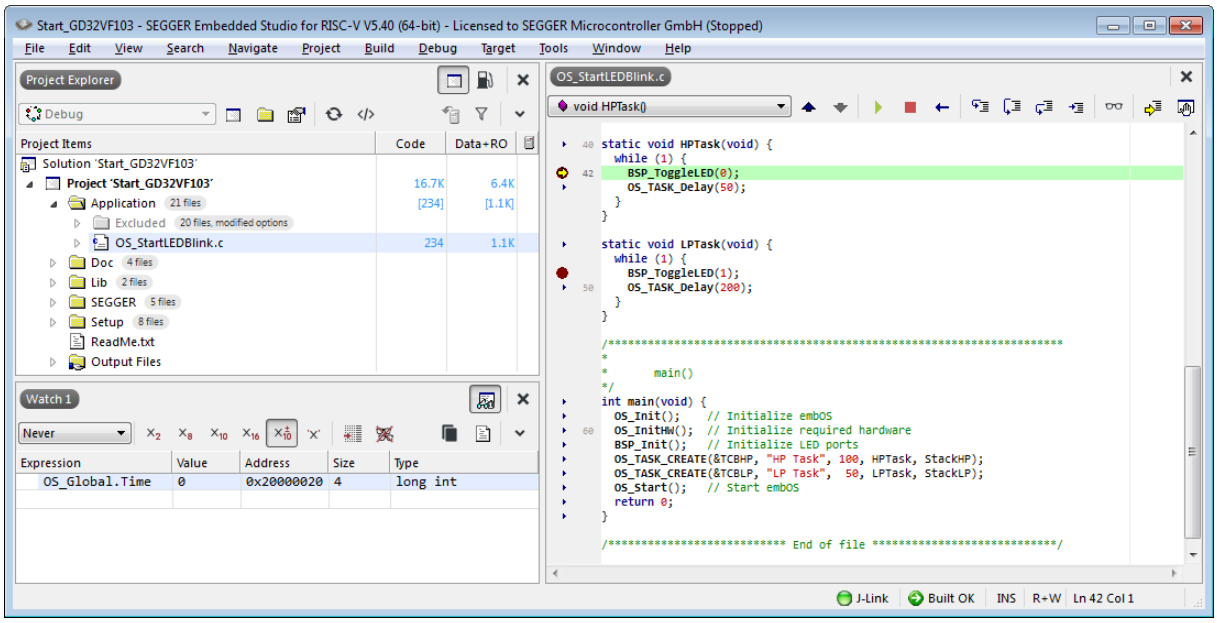


Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.

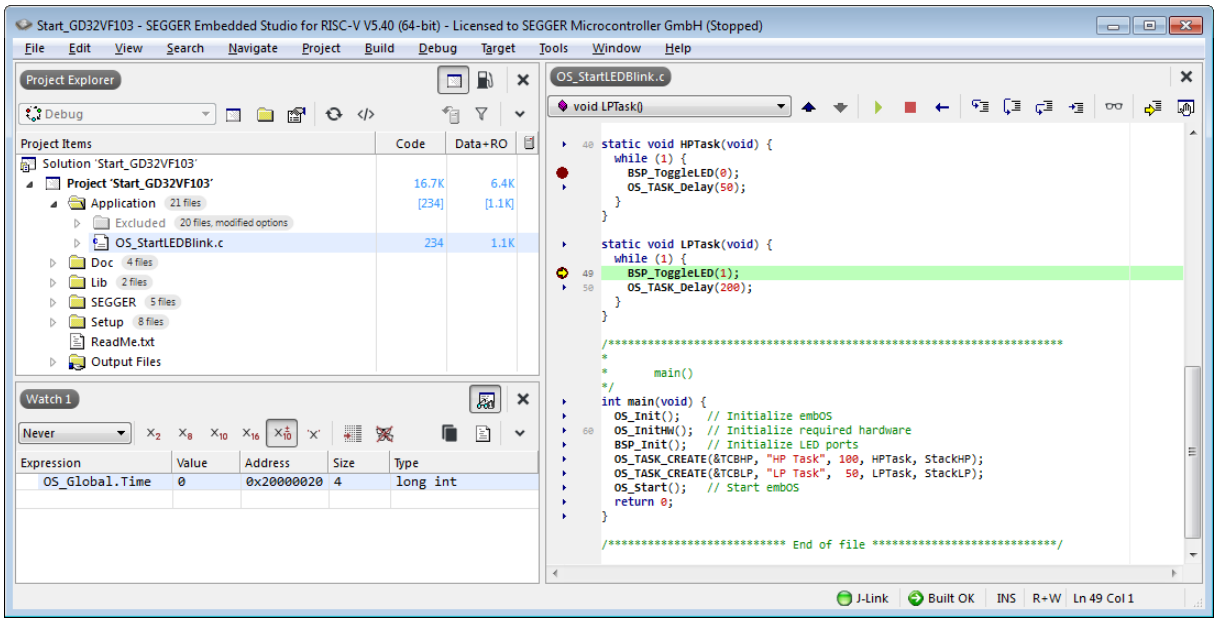


As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only.

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

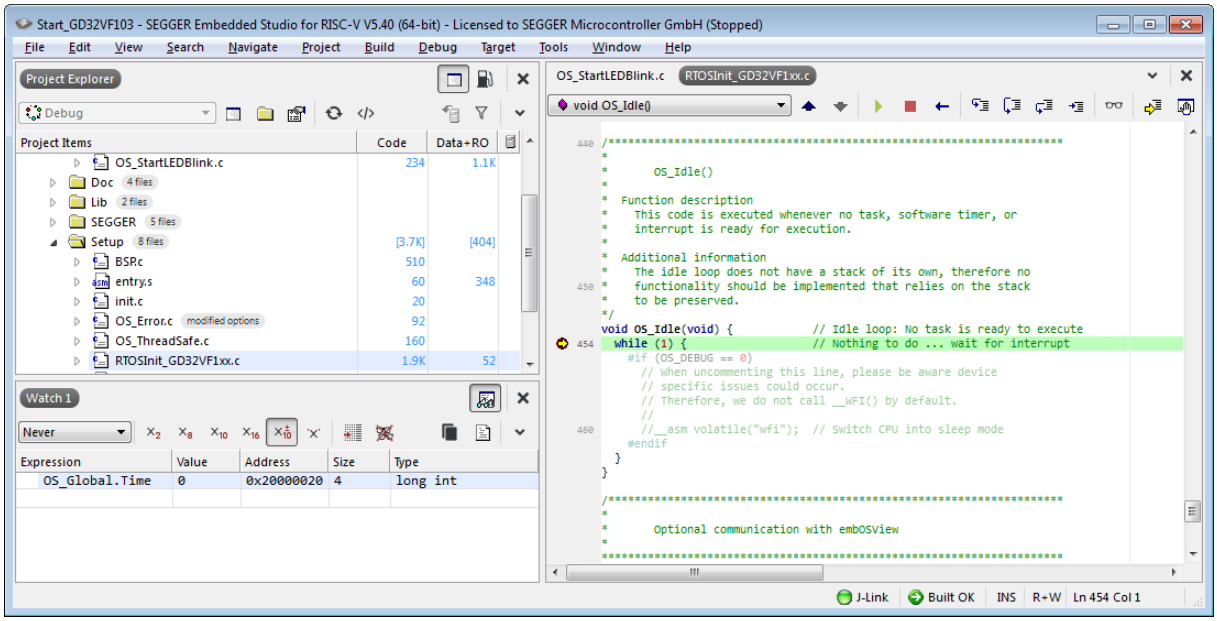


If you continue stepping, you will arrive at the task that has lower priority:



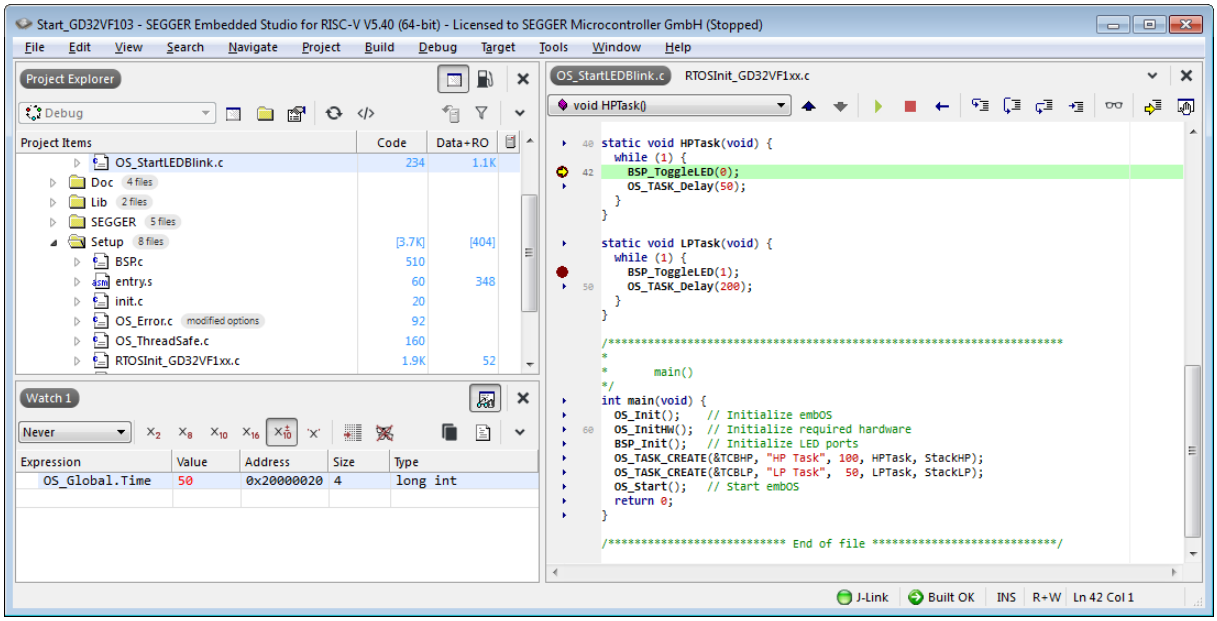
Continue to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

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



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

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



Chapter 2

Build your own application

2.1 Introduction

This chapter provides all information to set up your own embOS project. To build your own application, you should always start with one of the supplied sample workspaces and projects. Therefore, select an embOS workspace as described in chapter *First Steps* on page 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

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 RISC-V and Embedded Studio may be used with Embedded Studio's standard libraries.

embOS delivers the file `OS_ThreadSafe.c` which includes hook functions to make standard library calls (e.g. the heap management functions) thread safe.

4.2 Thread-Local Storage TLS

The Embedded Studio standard library supports the usage of thread-local storage. Several library objects and functions need local variables which have to be unique to a thread. Thread-local storage will be required when these functions are called from multiple threads.

embOS for Embedded Studio is prepared to support the tread-local storage, but does not use it per default. This has the advantage of no additional overhead as long as thread-local storage is not needed by the application. The embOS implementation of thread-local storage allows activation of TLS separately for every task. Only tasks that call functions using TLS need to activate the TLS by calling an initialization function when the task is started.

Library objects that need thread-local storage when used in multiple tasks are e.g.:

- error functions - errno, strerror.
- locale functions - localeconv, setlocale.
- time functions - asctime, localtime, gmtime, mktime.
- multibyte functions - mbrlen, mbrtowc, mbsrtowc, mbtowc, wcrtomb, wcsrtomb, wctomb.
- rand functions - rand, srand.
- etc functions - atexit, strtok.
- C++ exception engine.

> **Note**
>
> The usage of thread-local storage will prevent the SEGGER Linker from applying additional optimizations.

4.2.1 OS_TLS_SetTaskContextExtension()

Description

`OS_TLS_SetTaskContextExtension()` may be called from a task to initialize and use thread-local storage.

Prototype

```
void OS_TLS_SetTaskContextExtension(void);
```

Additional information

`OS_TLS_SetTaskContextExtension()` shall be the first function called from a task when TLS should be used in the specific task. The function must not be called multiple times from one task. The thread-local storage is allocated on the heap. To ensure thread-safe heap management when using TLS, make sure to include the `OS_ThreadSafe.c`.

Example

The following printout demonstrates the usage of task specific TLS in an application.

```
#include "RTOS.h"

static OS_STACKPTR int StackHP[128], StackLP[128];  // Task stacks
```



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

static void HPTask(void) {
  OS_TLS_SetTaskContextExtension();
  while (1) {
    errno = 42;  // errno specific to HPTask
    OS_TASK_Delay(50);
  }
}

static void LPTask(void) {
  OS_TLS_SetTaskContextExtension();
  while (1) {
    errno = 1;  // errno specific to LPTask
    OS_TASK_Delay(200);
  }
}

int main(void) {
  errno = 0;      // errno not specific to any task
  OS_Init();      // Initialize embOS
  OS_InitHW();    // Initialize required hardware
  OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
  OS_TASK_CREATE(&TCBLP, "LP Task",  50, LPTask, StackLP);
  OS_Start();     // Start embOS
  return 0;
}
```


Chapter 5

Stacks

5.1 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 RISC-V CPUs, this minimum basic task stack size is about 160 bytes. Because any function call uses some amount of stack and every exception also pushes at least 80 bytes onto the current stack, the task stack size has to be large enough to handle one exception, too. We recommend at least 512 bytes stack as a start.

> **Note**
>
> Stacks for RV32I devices need to be 16-byte aligned. embOS ensures that task stacks are properly aligned. However, since this can result in unused bytes, the application should ensure that task stacks are properly aligned. This can be achieved by defining an array using the compilers "`__attribute__`" keyword with the "aligned(16)" attribute.

5.2 System stack for RISC-V

The minimum system stack size required by embOS is about 192 bytes (stack check & profiling build). However, since the system stack is also used by the application before the start of multitasking (the call to `OS_Start()`), and because software timers and C-level interrupt handlers also use the system stack, the actual stack requirements depend on the application.

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.

5.3 Interrupt stack

RISC-V does not support a hardware interrupt stack. All interrupts primarily run on the current stack. This means that an interrupt might use any task stack or the system stack. Therefore, each task stack needs to be big enough to handle interrupts. Because giving each task additional memory for handling interrupts consumes much memory, it is possible to switch to the system stack instead.

Switching to the system stack can be done by calling `OS_INT_EnterIntStack()` and `OS_INT_LeaveIntStack()`. Fore more information, please refer to their function description in the generic embOS manual.

Chapter 6

Interrupts

6.1 CLINT and PLIC

This part describes interrupts and interrupt handling for RISC-V devices using a CLINT and PLIC as interrupt controller.

6.1.1 What happens when an interrupt occurs?

- A hart (hardware thread) 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, `OS_TrapHandler()` which is 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 registers 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*, continuing the interrupted function.

6.1.2 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_TrapHandler()`. Alternatively, local ISR handling may also be performed in vectored mode. In this case, a ROM vector table is used to distribute traps.

6.1.3 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 prologue and epilogue function as described in the generic manual and in the examples below.

Example

Simple interrupt routine:

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


6.1.4 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

6.1.5 Interrupt handling

6.1.5.1 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

6.1.5.1.1 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

6.1.5.1.2 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.

6.1.5.1.3 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

6.1.5.1.4 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.

6.1.5.2 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

6.1.5.2.1 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

6.1.5.2.2 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

6.1.5.2.3 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

6.1.5.2.4 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

6.1.5.2.5 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

6.1.5.2.6 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

6.1.5.2.7 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

6.1.5.2.8 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.

6.1.5.2.9 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

6.1.5.2.10 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

6.2 Enhanced CLIC (ECLIC)

This part describes interrupts and interrupt handling for RISC-V devices using an Enhanced CLIC interrupt controller. The ECLIC is an improved version of the CLIC and adds additional functionality.

6.2.1 What happens when an interrupt occurs?

- As soon as interrupts are globally enabled in *mstatus.MIE* and the interrupt source is enabled, the hart's (hardware thread) privilege mode is set to machine mode and following CSRs and registers are updated automatically by the hardware within one cycle:
  - *mepc* is set to the current pc.
  - *mstatus.MPIE* is set to *mstatus.MIE*.
  - *mstatus.MIE* is set to zero, disabling interrupts.
  - *mstatus.MPP* is updated with the previous privilege mode.
  - *msubm.PTYP* is stores the interrupt handling mode contained in *msubm.TYP*.
  - *msubm.TYP* is set to the new interrupt handling mode.
  - *mcause.EXCODE* is updated with the interrupt Id.
  - *mcause.MPIL* is updated with the interrupt level stored in *mintstatus.MIL*.
  - The pc is set to *mtvt2*.
- The processor continues execution at the updated *pc* address which points to the low-level interrupt handler.
- The low-level interrupt handler saves the caller-save registers as well as *mepc*, *msubm* and *mcause* on the stack, as those are overwritten and would be lost if a nested interrupt occurs.
- The low-level interrupt handler fetches the vector address of the next pending non-vector interrupt with highest interrupt level and highest priority in the same privilege mode, loads the handler's address from fetched vector table address and jumps into the interrupt specific high-level handler.
- After the high-level interrupt handler returns, the low-interrupt handler restores the caller-save registers and CSRs from the stack.
- The low-level interrupt handler ends by executing an *mret* instruction, causing following CSRs to be updated:
  - *mintstatus.MIL* is updated with the interrupt level stored in *mcause.MPIL*.
  - *msubm.TYP* is set to the mode encoded in *msubm.PTYP*.
  - The privilege mode is set to the value encoded in *mstatus.MPP*.
  - *mstatus.MIE* is set to the value of *mstatus.MPIE*.
  - The *pc* is set to the value of *mepc*, continuing the interrupted function.

6.2.2 RISC-V interrupt sources

RISC-V harts can have local and external interrupt sources. However, both interrupt types are handled by the ECLIC and behave the same. That is, all interrupts are controlled via the memory mapped interrupt configuration SFRs and the *mie* and *mip* CSRs are disabled in ECLIC mode.

6.2.3 Interrupt level and priority

For CLIC interrupt controllers, each interrupt has an 8-bit control register which is used to specify the interrupt level and priority. Depending on how many of the control bits are implemented on the device, there can be a maximum of 256 different combinations of interrupt level and priority for an interrupt. The level is stored on the MSB side of the control register, while the remaining bits are used for the priority. How many of the available control bits are used for the interrupt level can be specified. By default, all control bits are used for the interrupt level. That is, the number of level bits is set to 8.

Interrupt level

Interrupts with higher interrupt level can interrupt interrupts with lower interrupt level, leading to interrupt nesting. Furthermore, interrupts can be nested by synchronous excep-

tions. The synchronous exception is always taken with the current interrupt level. That means that interrupts and exceptions with greater interrupt level are able to interrupt an exception with lower interrupt level.

Interrupt priority

Interrupts with higher priority won't interrupt interrupts with same interrupt level even if the current active interrupt has a lower priority. The interrupt priority is used only for interrupt arbitration if there are two interrupts with the same interrupt level pending.

6.2.4 Interrupt handling

To handle ECLIC interrupts, embOS offers the following functions:

Function	Description
OS_RISCV_ECLIC_ISR_Disable()	Disables the specified interrupt source.
OS_RISCV_ECLIC_ISR_Enable()	Enables the specified interrupt source.
OS_RISCV_ECLIC_ISR_GetNumLevelBits()	Returns how many bits of the interrupt control register are used for the interrupt level.
OS_RISCV_ECLIC_ISR_GetPriority()	Returns the interrupt control value of the specified interrupt.
OS_RISCV_ECLIC_ISR_GetThreshold()	Returns the current interrupt level threshold.
OS_RISCV_ECLIC_ISR_Init()	Initializes the ECLIC interrupt controller.
OS_RISCV_ECLIC_ISR_SetNumLevelBits()	Specifies how many bits of the interrupt control register shall be used for the interrupt level.
OS_RISCV_ECLIC_ISR_SetPriority()	Sets the interrupt control value of the specified interrupt.
OS_RISCV_ECLIC_ISR_SetThreshold()	Configures the IRQ threshold, masking lower-level interrupts.

6.2.4.1 OS_RISCV_ECLIC_ISR_Disable()

Description

`OS_RISCV_ECLIC_ISR_Disable()` disables the specified interrupt.

Prototype

```
void OS_RISCV_ECLIC_ISR_Disable(OS_UINT ISRIndex);
```

Parameters

Parameter	Description
ISRIndex	Interrupt index.

6.2.4.2 OS_RISCV_ECLIC_ISR_Enable()

Description

`OS_RISCV_ECLIC_ISR_Enable()` enables the specified interrupt.

Prototype

```
void OS_RISCV_ECLIC_ISR_Enable(OS_UINT ISRIndex);
```

Parameters

Parameter	Description
`ISRIndex`	Interrupt index.

6.2.4.3 OS_RISCV_ECLIC_ISR_GetNumLevelBits()

Description

`OS_RISCV_ECLIC_ISR_GetNumLevelBits()` returns how many bits of the interrupt control register are used for the interrupt level.

Prototype

```
OS_U8 OS_RISCV_ECLIC_ISR_GetNumLevelBits(void);
```

Return value

The number of level bits.

6.2.4.4 OS_RISCV_ECLIC_ISR_GetPriority()

Description

`OS_RISCV_ECLIC_ISR_GetPriority()` returns the interrupt control value of the specified interrupt.

Prototype

```
OS_U8 OS_RISCV_ECLIC_ISR_GetPriority(OS_UINT ISRIndex);
```

Parameters

Parameter	Description
`ISRIndex`	Interrupt index.

Return value

The interrupt control value containing the interrupt level and priority.

6.2.4.5 OS_RISCV_ECLIC_ISR_GetThreshold()

Description

`OS_RISCV_ECLIC_ISR_GetThreshold()` returns the current interrupt level threshold.

Prototype

```
OS_U8 OS_RISCV_ECLIC_ISR_GetThreshold(void);
```

Return value

The current interrupt level threshold.

6.2.4.6 OS_RISCV_ECLIC_ISR_Init()

Description

`OS_RISCV_ECLIC_ISR_Init()` initializes the ECLIC interrupt controller.

Prototype

```
void OS_RISCV_ECLIC_ISR_Init(void* pBaseAddr,
                             void* pVectorTable,
                             void* pTrapHandler);
```

Parameters

Parameter	Description
`pBaseAddr`	Base address of the memory mapped ECLIC SFRs.
`pVectorTableAddress`	Address of the vector table containing the ISR handler addresses. Needs to be at least 64-bit aligned. Alignment increases with size of the vector table (See additional information).
`pTrapHandlerAddress`	Address of the synchronous trap handler. Needs to be 64-bit aligned.

Additional information

The vector table address is constrained to be at least 64-byte aligned. This alignment should be considered when linking the application.

```
   0 to   16 max. interrupts =>    64-byte aligned
  17 to   32 max. interrupts =>   128-byte aligned
  33 to   64 max. interrupts =>   256-byte aligned
  65 to  128 max. interrupts =>   512-byte aligned
 129 to  256 max. interrupts =>  1024-byte aligned
 257 to  512 max. interrupts =>  2048-byte aligned
 513 to 1024 max. interrupts =>  4096-byte aligned
1025 to 2048 max. interrupts =>  8192-byte aligned
2045 to 4096 max. interrupts => 16384-byte aligned
```

6.2.4.7 OS_RISCV_ECLIC_ISR_SetNumLevelBits()

Description

`OS_RISCV_ECLIC_ISR_SetNumLevelBits()` Specifies how many bits of the interrupt control register shall be used for the interrupt level.

Prototype

```
void OS_RISCV_ECLIC_ISR_SetNumLevelBits(OS_U8 NumLevelBits);
```

Parameters

Parameter	Description
`NumLevelBits`	Number of level bits that shall be used. Valid value are 0-8.

6.2.4.8 OS_RISCV_ECLIC_ISR_SetPriority()

Description

`OS_RISCV_ECLIC_ISR_SetPriority()` sets the interrupt control bits of the specified interrupt. The interrupt control register consists of two parts: the interrupt level and the interrupt priority, depending on the number of level bits used. The interrupt level bits are on the MSB side, while priority bits are on the LSB side. The number of level bits used is by default set to 8, but can be changed by a call to `OS_RISCV_ECLIC_ISR_SetNumLevelBits()`.

Prototype

```
void OS_RISCV_ECLIC_ISR_SetPriority(OS_UINT ISRIndex,
                                    OS_U8   InterruptPriority);
```

Parameters

Parameter	Description
`ISRIndex`	Interrupt index.
`InterruptPriority`	Interrupt level and priority.

6.2.4.9 OS_RISCV_ECLIC_ISR_SetThreshold()

Description

`OS_RISCV_ECLIC_ISR_SetThreshold()` configures the IRQ threshold, masking lower-level interrupts.

Prototype

```
void OS_RISCV_ECLIC_ISR_SetThreshold(OS_U8 Threshold);
```

Parameters

Parameter	Description
`Threshold`	Desired interrupt priority threshold.

Example

For a device with 5 implemented control bits it is possible to use 2^5=32 different values for interrupt priority arbitration. If the number of level bits is set to 3, 8 levels and 4 priorities can be used. In order to set an interrupt to level 7 and priority 2, the value ((7 << (5 - 3)) | 2) = 30 has to be passed as interrupt priority.

6.3 Interrupt-stack switching

RISC-V does not support a hardware Interrupt stack. All interrupts primarily run on the current stack. This means, that an interrupt might use any task stack or the system stack, and thus each task stack needs to be big enough to handle interrupts. As this would consume much memory, it is possible to switch to the system stack on interrupt entry if desired, so that it has to be ensured that the system-stack is big enough for handling interrupt.

Switching to the system stack can be done by calling `OS_INT_EnterIntStack()` and `OS_INT_LeaveIntStack()`. Fore more information, please refer to their function description in the generic embOS manual.

6.4 Zero latency interrupts

Zero latency interrupts are currently not supported.

Chapter 7

RTT and SystemView

7.1 SEGGER Real Time Transfer

SEGGER's Real Time Transfer (RTT) is the new technology for interactive user I/O in embedded applications. 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.

7.1.1 Shipped files related to SEGGER RTT

All files related to SEGGER RTT are shipped inside the respective start project's Setup folder:

File	Description
`SEGGER_RTT.c`	Generic implementation of SEGGER RTT.
`SEGGER_RTT.html`	Generic implementation header file.
`SEGGER_RTT_Conf.h`	Generic RTT configuration file.
`SEGGER_RTT_printf.c`	Generic printf() replacement to write formatted data via RTT.
`SEGGER_RTT_Syscalls_*.c`	Compiler-specific low-level functions for using printf() via RTT. If this file is included in a project, RTT is used for debug output. To use the standard out of your IDE, exclude this file from build.

7.2 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, SystemViewer, is not shipped with embOS. Instead, the most recent version of that application is available for download from our website.

For more information on SEGGER SystemView, including the SystemViewer download, refer to segger.com/systemview.

7.2.1 Shipped files related to SEGGER SystemView

All files related to SEGGER SystemView are shipped inside the respective start project's `Setup` folder:

File	Description
`Global.h`	Global type definitios required by SEGGER SystemView.
`SEGGER.h`	Generic types and utility function header.
`SEGGER_SYSVIEW.c`	Generic implementation of SEGGER RTT.
`SEGGER_SYSVIEW.h`	Generic implementation include file.
`SEGGER_SYSVIEW_Conf.h`	Generic configuration file.
`SEGGER_SYSVIEW_ConfDefaults.h`	Generic default configuration file.
`SEGGER_SYSVIEW_Config_embOS.c`	Target-specific configuration of SystemView with embOS.
`SEGGER_SYSVIEW_embOS.c`	Generic interface implementation for SystemView with embOS.
`SEGGER_SYSVIEW_embOS.h`	Generic interface implementation header file for SystemView with embOS.
`SEGGER_SYSVIEW_Int.h`	Generic internal header file.

Chapter 8

embOS Thread Script

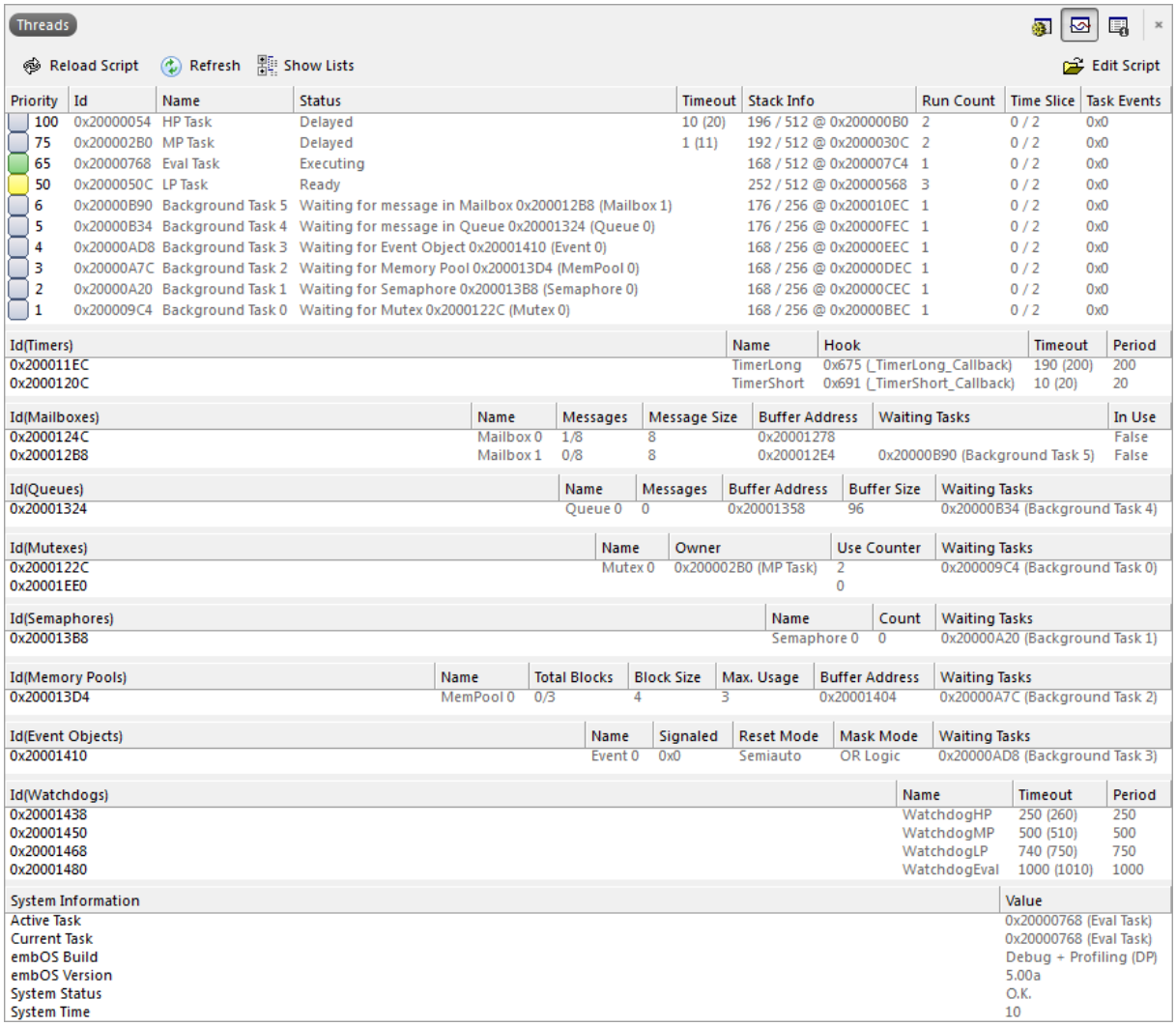
8.1 Introduction

A thread script is included with every board support package shipped with embOS. This script may be used to display various information about the system, the tasks and created embOS objects like timers, mailboxes, queues, semaphores, memory pools, events and watchdogs.

When creating a custom project, the thread script may be added to the respective project's options (`"Debug" -> "Debugger" -> "Threads Script File"`).

8.2 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 `More Debug Windows`. Alternatively, the threads window may also be enabled by pressing `[Ctrl + Alt + H]`. The object lists and system information within the threads window can be enabled or disabled via the `Show Lists` dropdown menu. The threads window gets updated every time the application is halted. It should closely resemble the screenshot below:



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.

8.2.1 Task List

Priority	Id	Name	Status	Timeout	Stack Info	Run Count	Time Slice	Task Events
100	0x20000054	HP Task	Delayed	10 (20)	196 / 512 @ 0x200000B0	2	0 / 2	0x0
75	0x200002B0	MP Task	Delayed	1 (11)	192 / 512 @ 0x2000030C	2	0 / 2	0x0
65	0x20000768	Eval Task	Executing		168 / 512 @ 0x200007C4	1	0 / 2	0x0
50	0x2000050C	LP Task	Ready		252 / 512 @ 0x20000568	3	0 / 2	0x0
6	0x20000B90	Background Task 5	Waiting for message in Mailbox 0x200012B8 (Mailbox 1)		176 / 256 @ 0x200010EC	1	0 / 2	0x0
5	0x20000B34	Background Task 4	Waiting for message in Queue 0x20001324 (Queue 0)		176 / 256 @ 0x20000FEC	1	0 / 2	0x0
4	0x20000AD8	Background Task 3	Waiting for Event Object 0x20001410 (Event 0)		168 / 256 @ 0x20000EEC	1	0 / 2	0x0
3	0x20000A7C	Background Task 2	Waiting for Memory Pool 0x200013D4 (MemPool 0)		168 / 256 @ 0x20000DEC	1	0 / 2	0x0
2	0x20000A20	Background Task 1	Waiting for Semaphore 0x200013B8 (Semaphore 0)		168 / 256 @ 0x20000CEC	1	0 / 2	0x0
1	0x200009C4	Background Task 0	Waiting for Mutex 0x2000122C (Mutex 0)		168 / 256 @ 0x20000BEC	1	0 / 2	0x0

The task list displays various information about the running tasks:

Column	Description
Priority	This is the priority of the task
Id	The address of a tasks task control block
Name	The name of the task
Status	The current status of the task
Timeout	Time in ms till the task gets called again
Stack Info	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
Time Slice	Show the number of remaining and maximum time slices if round robin scheduling is available
Task Events	Show the event mask of a task

> **Note**
>
> 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 ("Debug" -> "Debugger" -> "Thread Maximum").

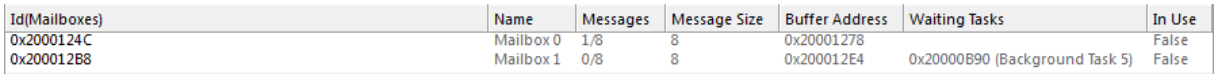
8.2.2 Timers

Id(Timers)	Name	Hook	Timeout	Period
0x200011EC	TimerLong	0x675 (_TimerLong_Callback)	190 (200)	200
0x2000120C	TimerShort	0x691 (_TimerShort_Callback)	10 (20)	20

The timers list displays various information about active timers:

Column	Description
Id(Timers)	The timer's address
Name	If available, the respective object identifier is shown here
Hook	The function address that is called after the timeout
Timeout	The time delay and the point in time, when the timer finishes waiting
Period	The time period the timer runs

8.2.3 Mailboxes

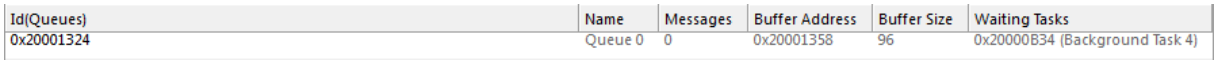


Id(Mailboxes)	Name	Messages	Message Size	Buffer Address	Waiting Tasks	In Use
0x2000124C	Mailbox 0	1/8	8	0x20001278		False
0x200012B8	Mailbox 1	0/8	8	0x200012E4	0x20000B90 (Background Task 5)	False

The mailboxes list displays various information about used mailboxes:

Column	Description
Id(Mailboxes)	The mailbox's address
Name	If available, the respective object identifier is shown here
Messages	The number of messages in a mailbox and the maximum number of messages the mailbox can hold
Message Size	The size of an individual message in bytes
Buffer Address	The message buffer address
Waiting Tasks	The list of tasks that are waiting for the mailbox (address and, if available, name)

8.2.4 Queues

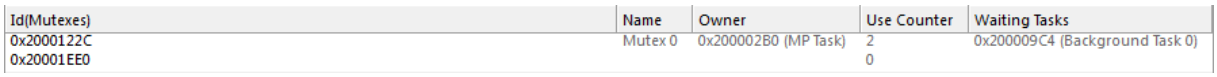


Id(Queues)	Name	Messages	Buffer Address	Buffer Size	Waiting Tasks
0x20001324	Queue 0	0	0x20001358	96	0x20000B34 (Background Task 4)

The queues list displays various information about used queues:

Column	Description
Id(Queues)	The queue's address
Name	If available, the respective object identifier is shown here
Messages	The number of messages in a queue
Buffer Address	The message buffer address
Buffer Size	The size of the message buffer in bytes
Waiting Tasks	The list of tasks that are waiting for the queue (address and, if available, name)

8.2.5 Mutexes



Id(Mutexes)	Name	Owner	Use Counter	Waiting Tasks
0x2000122C	Mutex 0	0x200002B0 (MP Task)	2	0x200009C4 (Background Task 0)
0x20001EE0			0	

The mutexes list displays various information about used mutexes:

Column	Description
Id(Mutexes)	The mutexes' address
Name	If available, the respective object identifier is shown here
Owner	The address and name of the owner task
Use Counter	Counts the number of times the mutex was claimed
Waiting Tasks	The list of tasks that are waiting for the mutex (address and, if available, name)

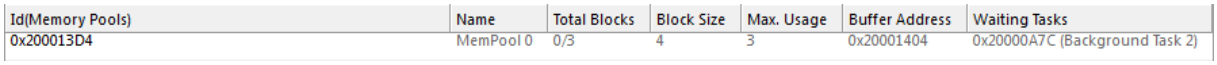
8.2.6 Semaphores

Id(Semaphores)	Name	Count	Waiting Tasks
0x200013B8	Semaphore 0	0	0x20000A20 (Background Task 1)

The semaphores list displays various information about used semaphores:

Column	Description
Id(Semaphores)	The semaphores' address
Name	If available, the respective object identifier is shown here
Count	Counts how often this semaphore can be claimed
Waiting Tasks	The list of tasks that are waiting for the semaphore (address and, if available, name)

8.2.7 Memory Pools

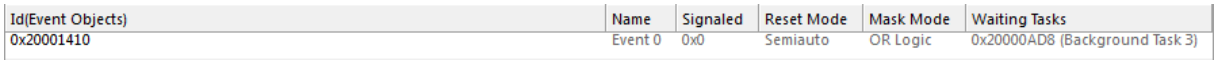


Id(Memory Pools)	Name	Total Blocks	Block Size	Max. Usage	Buffer Address	Waiting Tasks
0x200013D4	MemPool 0	0/3	4	3	0x20001404	0x20000A7C (Background Task 2)

The memory pools list displays various information about used memory pools:

Column	Description
Id(Memory Pools)	The memory pool's address
Name	If available, the respective object identifier is shown here
Total Blocks	Shows the available blocks and the maximal number of blocks
Block Size	Shows the size of a single memory block
Max. Usage	Shows the maximal count of blocks which were simultaneously allocated
Buffer Address	The address of the memory pool buffer
Waiting Tasks	The list of tasks that are waiting for free memory blocks (address and, if available, name)

8.2.8 Event Objects



Id(Event Objects)	Name	Signaled	Reset Mode	Mask Mode	Waiting Tasks
0x20001410	Event 0	0x0	Semiauto	OR Logic	0x20000AD8 (Background Task 3)

The event objects list displays various information about used event objects:

Column	Description
Id(Event Objects)	The event object's address
Name	If available, the respective object identifier is shown here
Signaled	The hexadecimal value of the bit mask containing the signaled event bits
Reset Mode	The event object's reset mode
Mask Mode	The current mask mode indicating whether OR or AND logic is used to check if a task shall resume
Waiting Tasks	The list of tasks that are waiting for the event object (address and, if available, name)

8.2.9 Watchdogs

Id(Watchdogs)	Name	Timeout	Period
0x20001438	WatchdogHP	250 (260)	250
0x20001450	WatchdogMP	500 (510)	500
0x20001468	WatchdogLP	740 (750)	750
0x20001480	WatchdogEval	1000 (1010)	1000

The watchdogs list displays various information about used watchdogs:

Column	Description
Id(Watchdogs)	The watchdog’s address
Name	If available, the respective object identifier is shown here
Timeout	The remaining time (and the system time in parentheses) until the watchdog has to be fed
Period	The period in which the watchdog has to be fed

8.2.10 System Information

The system information list displays various information about embOS.

System Information	Value
Active Task	0x20000768 (Eval Task)
Current Task	0x20000768 (Eval Task)
embOS Build	Debug + Profiling (DP)
embOS Version	5.00a
System Status	O.K.
System Time	10

Chapter 9

Technical data

9.1 Memory requirements

This chapter lists technical data of embOS used with RISC-V CPUs. 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
Mutex	16
Semaphore	8
Mailbox	24
Queue	32
Task event	0
Event object	12

