embOS

Real-Time Operating System

CPU & Compiler specifics for RISC-V using GCC

Document: UM01083
Software Version: 5.18.0.0
Revision: 0
Date: November 18, 2022

SEGGER

A product of SEGGER Microcontroller GmbH

www.segger.com

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: November 18, 2022

Software	Revision	Date	By	Description
5.18.0.0	0	221118	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.

Note

The BSP projects at `/Start/BoardSupport/<DeviceManufacturer>/<Device>` assume that the `/Start/Lib` and `/Start/Inc` folders are located relative to the BSP folder. If you copy a BSP folder to another location, you will need to adjust these paths in the project.

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

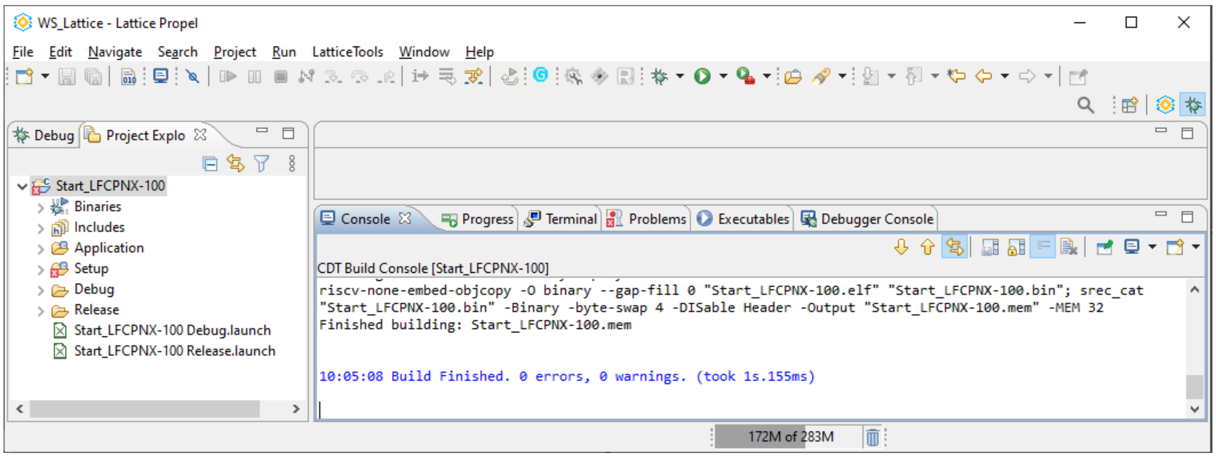
After installation of embOS you can create your first multitasking application. You have received several ready to go sample start workspaces and projects and every other files needed in the subfolder `Start`. It is a good idea to use one of them as a starting point for all of your applications. The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


WS_Lattice - Lattice Propel
File Edit Navigate Search Project Run LatticeTools Window Help
Debug
Project Explo
Start_LFCPNX-100
Binaries
Includes
Application
Setup
Debug
Release
Start_LFCPNX-100 Debug.launch
Start_LFCPNX-100 Release.launch
Console
Progress
Terminal
Problems
Executables
Debugger Console
CDT Build Console [Start_LFCPNX-100]
172M of 283M

```
riscv-none-embed-objcopy -O binary --gap-fill 0 "Start_LFCPNX-100.elf" "Start_LFCPNX-100.bin"; srec_cat
"Start_LFCPNX-100.bin" -Binary -byte-swap 4 -DISable Header -Output "Start_LFCPNX-100.mem" -MEM 32
Finished building: Start_LFCPNX-100.mem


10:05:08 Build Finished. 0 errors, 0 warnings. (took 1s.155ms)
```

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

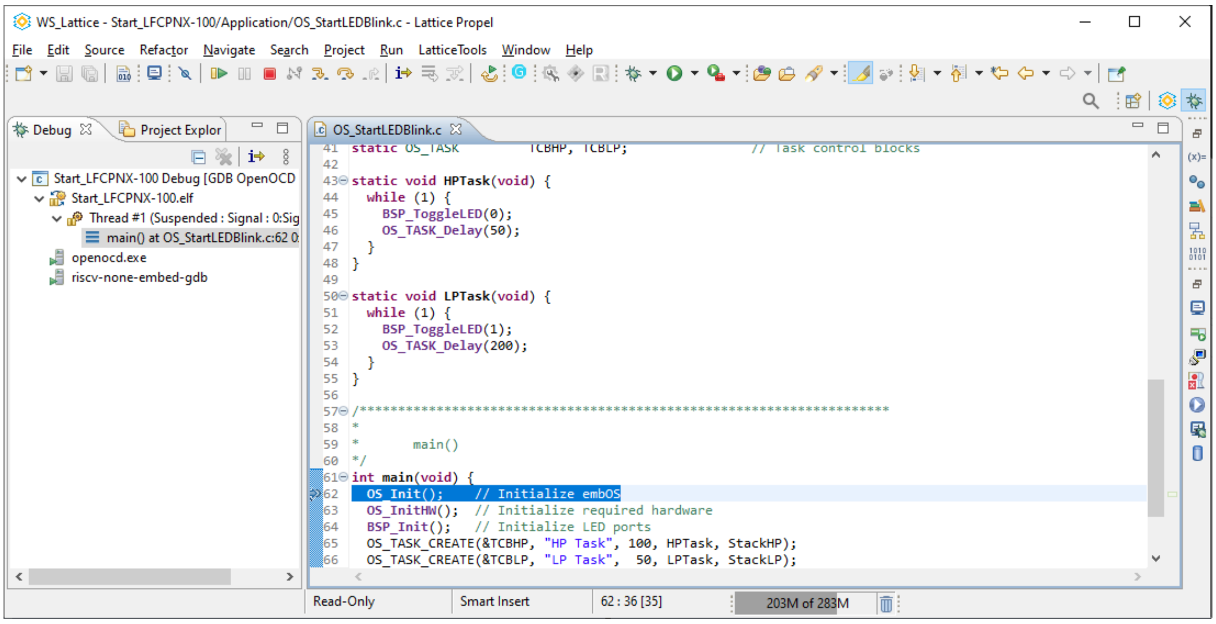
1.4 Stepping through the sample application

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.

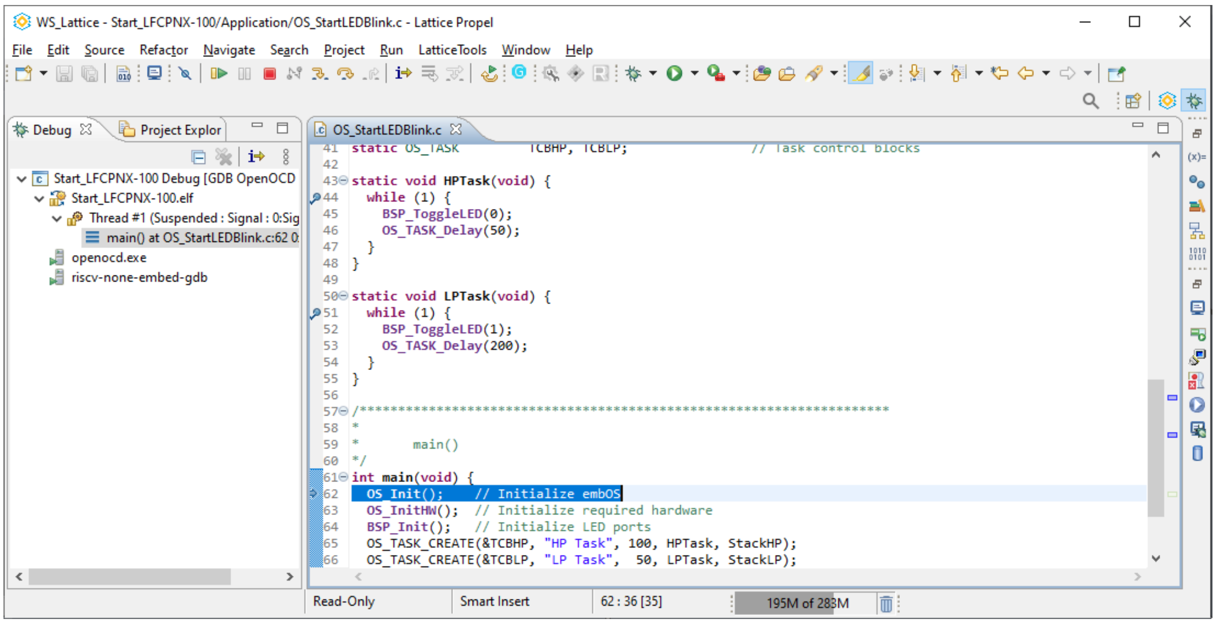
`OS_InitHW()` is part of `RTOSInit.c` and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the system tick interrupt for embOS. Step through it to see what is done.

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


WS_Lattice - Start_LFCPNX-100/Application/OS_StartLEDBlink.c - Lattice Propel
File Edit Source Refactor Navigate Search Project Run LatticeTools Window Help
Debug
Project Explor
Start_LFCPNX-100 Debug [GDB OpenOCD
Start_LFCPNX-100.elf
Thread #1 (Suspended : Signal : 0:Sig
main() at OS_StartLEDBlink.c:62 0
openocd.exe
riscv-none-embed-gdb
OS_StartLEDBlink.c
Read-Only
Smart Insert
62 : 36 [35]
203M of 283M

```
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);
```

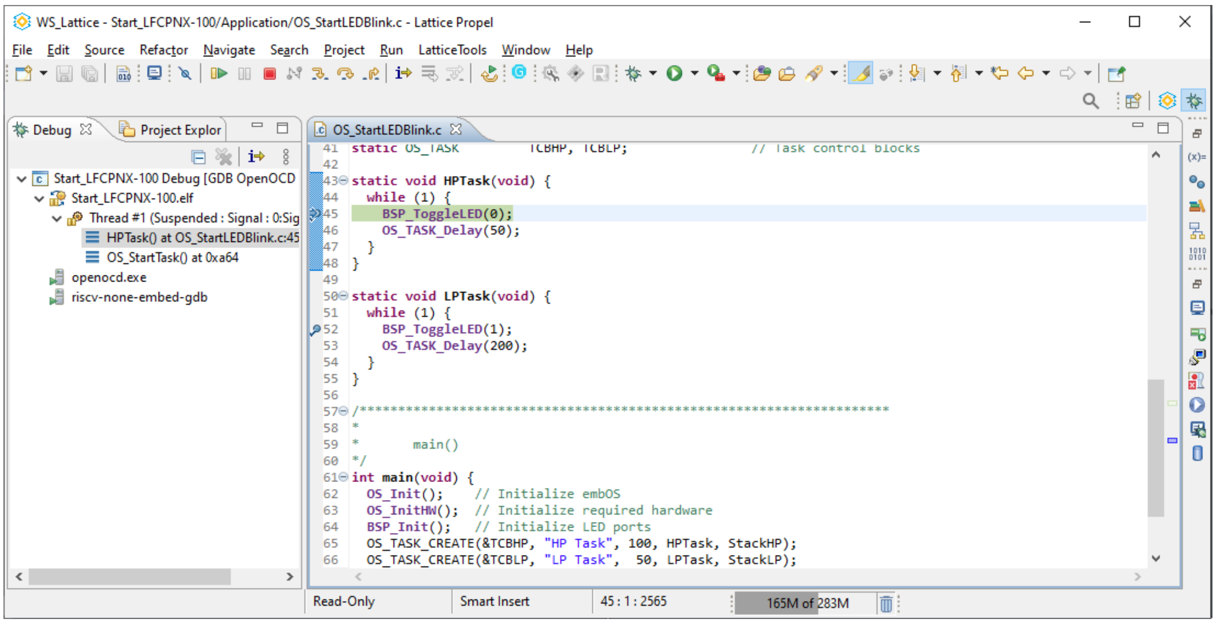
Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.


WS_Lattice - Start_LFCPNX-100/Application/OS_StartLEDBlink.c - Lattice Propel
File Edit Source Refactor Navigate Search Project Run LatticeTools Window Help
Debug
Project Explor
Start_LFCPNX-100 Debug [GDB OpenOCD
Start_LFCPNX-100.elf
Thread #1 (Suspended : Signal : 0:Sig
main() at OS_StartLEDBlink.c:62 0
openocd.exe
riscv-none-embed-gdb
OS_StartLEDBlink.c
Read-Only
Smart Insert
62 : 36 [35]
195M of 283M

```
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);
```

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.

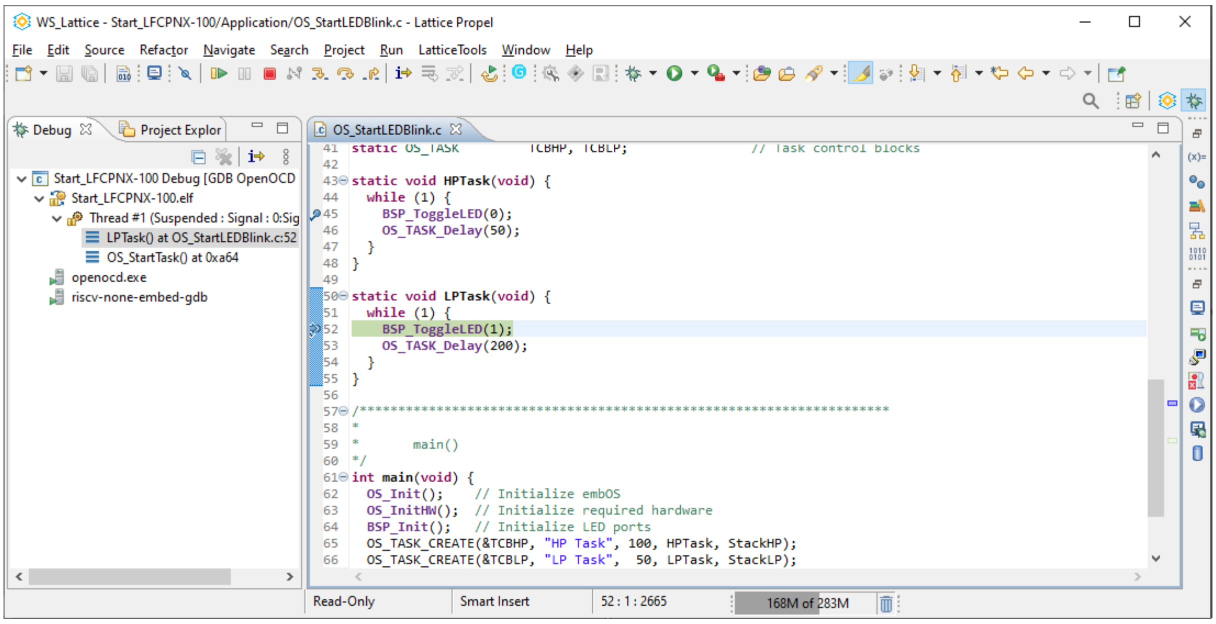

WS_Lattice - Start_LFCPNX-100/Application/OS_StartLEDBlink.c - Lattice Propel
File Edit Source Refactor Navigate Search Project Run LatticeTools Window Help
Debug
Project Explor
OS_StartLEDBlink.c

```
Start_LFCPNX-100 Debug [GDB OpenOCD
  Start_LFCPNX-100.elf
    Thread #1 (Suspended : Signal : 0:Sig
      HPTask() at OS_StartLEDBlink.c:45
      OS_StartTask() at 0xa64
  openocd.exe
  riscv-none-embed-gdb
```

```
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);
```

Read-Only | Smart Insert | 45 : 1 : 2665 | 165M of 283M

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


WS_Lattice - Start_LFCPNX-100/Application/OS_StartLEDBlink.c - Lattice Propel
File Edit Source Refactor Navigate Search Project Run LatticeTools Window Help
Debug
Project Explor
OS_StartLEDBlink.c

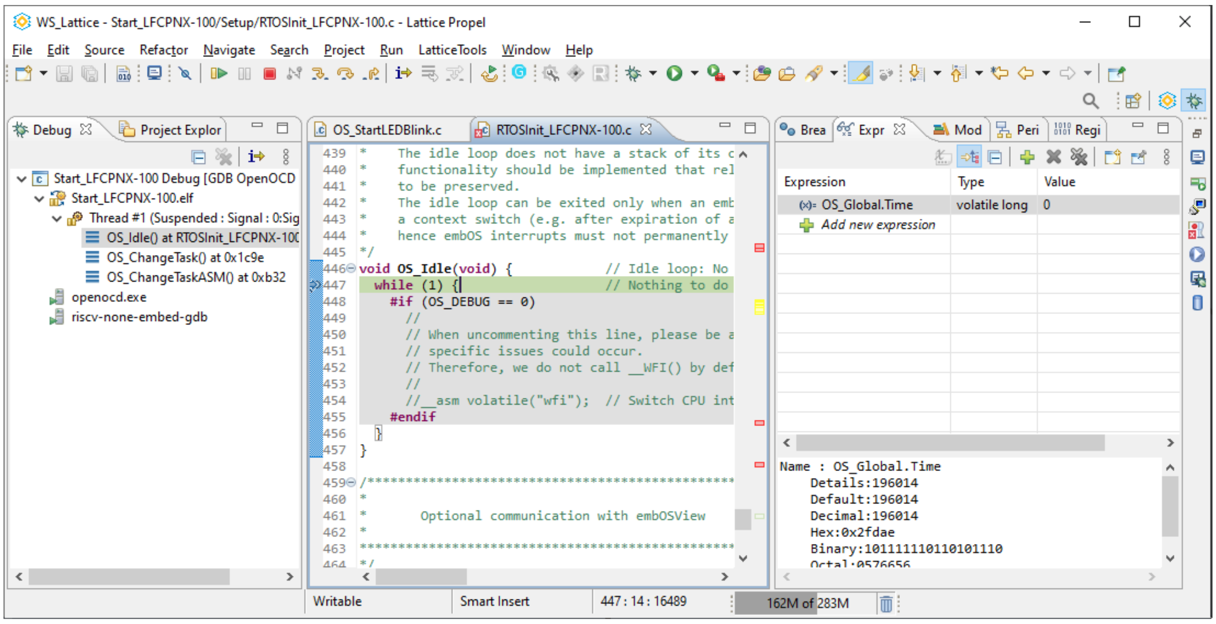
```
Start_LFCPNX-100 Debug [GDB OpenOCD
  Start_LFCPNX-100.elf
    Thread #1 (Suspended : Signal : 0:Sig
      LPTask() at OS_StartLEDBlink.c:52
      OS_StartTask() at 0xa64
  openocd.exe
  riscv-none-embed-gdb
```

```
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);
```

Read-Only | Smart Insert | 52 : 1 : 2665 | 168M of 283M

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()`.


WS_Lattice - Start_LFCPNX-100/Setup/RTOSInit_LFCPNX-100.c - Lattice Propel
File Edit Source Refactor Navigate Search Project Run LatticeTools Window Help
Debug
Project Explor
OS_StartLEDBlink.c
RTOSInit_LFCPNX-100.c
Brea
Expr
Mod
Peri
Regi

```
Start_LFCPNX-100 Debug [GDB OpenOCD
  Start_LFCPNX-100.elf
    Thread #1 (Suspended : Signal : 0:Sig
      OS_Idle() at RTOSInit_LFCPNX-10(
      OS_ChangeTask() at 0x1c9e
      OS_ChangeTaskASM() at 0xb32
  openocd.exe
  riscv-none-embed-gdb
```

```
*     The idle loop does not have a stack of its c
*     functionality should be implemented that rel
*     to be preserved.
*     The idle loop can be exited only when an emb
*     a context switch (e.g. after expiration of a
*     hence embOS interrupts must not permanently
*/
void OS_Idle(void) {              // Idle loop: No
  while (1) {                     // Nothing to do
    #if (OS_DEBUG == 0)
      //
      // When uncommenting this line, please be a
      // specific issues could occur.
      // Therefore, we do not call __WFI() by def
      //
      //__asm volatile("wfi");  // Switch CPU int
    #endif
  }
}

/**********************************************
*
*        Optional communication with embOSView
*
***********************************************
*/
```

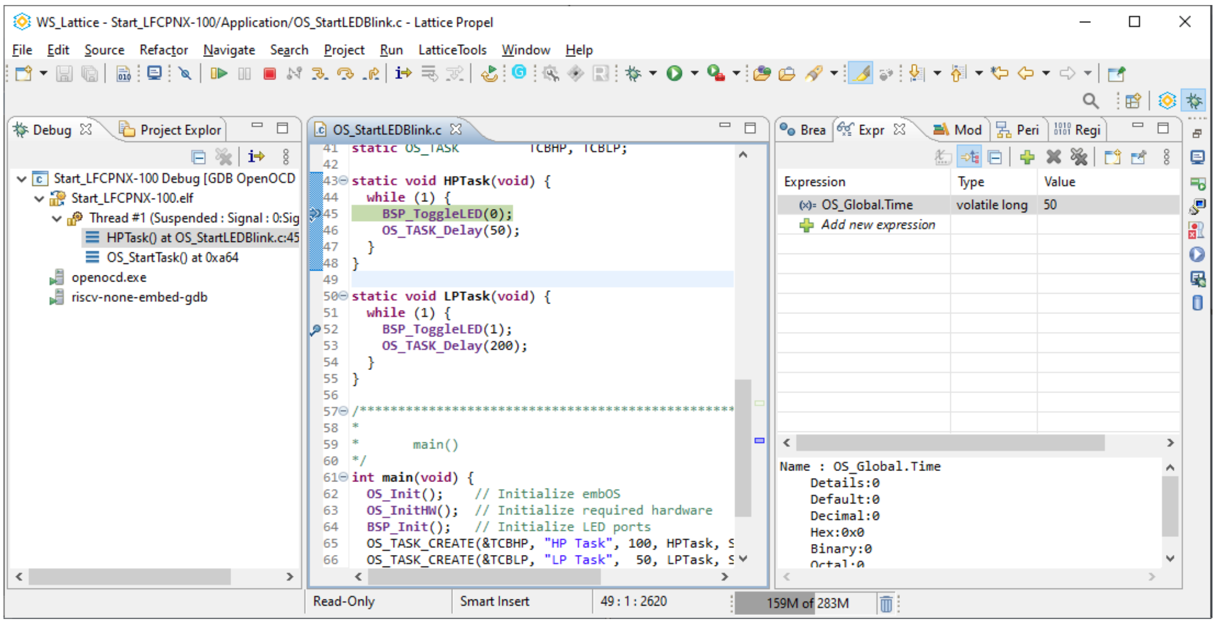
Expression	Type	Value
OS_Global.Time	volatile long	0
Add new expression		

```
Name : OS_Global.Time
    Details:196014
    Default:196014
    Decimal:196014
    Hex:0x2fdae
    Binary:101111110110101110
    Octal:0576656
```

```
Writable    Smart Insert    447 : 14 : 16489    162M of 283M
```

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

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


WS_Lattice - Start_LFCPNX-100/Application/OS_StartLEDBlink.c - Lattice Propel
File Edit Source Refactor Navigate Search Project Run LatticeTools Window Help
Debug
Project Explor
OS_StartLEDBlink.c
Brea
Expr
Mod
Peri
Regi

```
Start_LFCPNX-100 Debug [GDB OpenOCD
  Start_LFCPNX-100.elf
    Thread #1 (Suspended : Signal : 0:Sig
      HPTask() at OS_StartLEDBlink.c:45
      OS_StartTask() at 0xa64
  openocd.exe
  riscv-none-embed-gdb
```

```
static OS_TASK         TCBHP, TCBLP;

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, S
  OS_TASK_CREATE(&TCBLP, "LP Task",  50, LPTask, S
```

Expression	Type	Value
OS_Global.Time	volatile long	50
Add new expression		

```
Name : OS_Global.Time
    Details:0
    Default:0
    Decimal:0
    Hex:0x0
    Binary:0
    Octal:0
```

```
Read-Only    Smart Insert    49 : 1 : 2620    159M of 283M
```

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 32imc: RV32I with ’M’ 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 GCC may be used with the Red Hat newlib C libraries for most of all projects without any modification.

Since heap management with newlib depends on a working implementation of an `__sbrk()` function, that implementation is provided with embOS inside the source module `OS_Syscalls.c`, which itself is included in the “Setup” subdirectory of every embOS start project. Using that source file requires the symbols `__heap_start__` and `__heap_end__` to be appropriately defined in the respective project’s linker file.

Alternatively, if modification of the linker file is not feasible (e.g. when it is auto-generated by a project generator), the heap symbols may also be defined as additional linker flags. For example, if the linker file defines the symbols *`LOWEST_HEAP_ADDRESS`* and *`HIGHEST_HEAP_ADDRESS`*, the linker flags would need to be set as shown below:

```
--defsym=__heap_start__=LOWEST_HEAP_ADDRESS
--defsym=__heap_end__=HIGHEST_HEAP_ADDRESS
```

Heap management and file operation functions of standard system libraries are not reentrant and require a special initialization or additional modules when used with embOS when those non-thread-safe functions are used from different tasks (refer to *Interrupt and thread safety* on page 21).

4.2 Interrupt and thread safety

Using embOS with specific calls to standard library functions (e.g. heap management functions) may require thread-safe system libraries if these functions are called from several tasks or interrupts. Newlib provides functions, which can be overwritten to implement a locking mechanism making the system library functions thread-safe.

The Setup directory in each embOS BSP contains the file `OS_ThreadSafe.c` which overwrites these functions. By default they disable and restore embOS interrupts to ensure thread safety in tasks, embOS interrupts, `OS_Idle()` and software timers. Zero latency interrupts are not disabled and therefore unprotected. If you need to call e.g. malloc() also from within a zero latency interrupt additional handling needs to be added. If you don’t call such functions from within embOS interrupts, `OS_Idle()` or software timers, you can instead use thread safety for tasks only. This reduces the interrupt latency because a mutex is used instead of disabling embOS interrupts.

You can choose the safety variant with the macro `OS_INTERRUPT_SAFE`.

- When defined to 1 thread safety is guaranteed in tasks, embOS interrupts, `OS_Idle()` and software timers.
- When defined to 0 thread safety is guaranteed only in tasks. In this case you must not call e.g. heap functions from within an ISR, `OS_Idle()` or embOS software timers.

Alternatively, embOS delivers its own thread-safe functions for heap management. These are described in the embOS generic manual.

4.2.1 __malloc_lock(), lock the heap against mutual access

`__malloc_lock()` is the locking function which is called by the system library whenever the heap management has to be locked against mutual access. The implementation delivered with embOS claims a mutex or disables interrupts to achieve this.

4.2.2 __malloc_unlock()

`__malloc_unlock()` is the counterpart to `__malloc_lock()`. It is called by the system library whenever the heap management locking can be released. The implementation delivered with embOS releases the mutex or restores the interrupt state.

None of these functions has to be called directly by the application. They are called from the system library functions when required. The functions are delivered in source form to allow replacement of the dummy functions in the system library.

4.3 Thread-Local Storage TLS

Newlib supports 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 GCC is prepared to support thread-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 each task.

Only tasks that are accessing TLS variables, for instance by calling functions from the system library, need to initialize their TLS by calling an initialization function when the task is started. For each task that uses TLS the memory for the thread-local storage is allocated on the heap. Therefore, thread-safe heap management should be used together with TLS. For information on thread-safety, please refer to *Interrupt and thread safety* on page 21.

Library objects that need thread-local storage when used in multiple tasks are for example:

- 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.

4.3.1 OS_TLS_Set()

Description

`OS_TLS_Set()` is used by a task to initialize Thread-local storage for the current task.

Prototype

```
void OS_TLS_Set(struct _reent* pReentStruct);
```

Parameters

Parameter	Description
`pReentStruct`	Pointer to the thread local storage. It is the address of the variable of type struct `_reent` which holds the thread local data.

Additional information

`OS_TLS_Set()` shall be the first function called from a task when TLS should be used in the specific task. This function has to be only used in combination with `OS_TASK_AddContextExtension()` or `OS_TASK_SetContextExtension()` and `OS_TLS_ContextExtension` as argument to these functions. When `OS_TLS_SetTaskContextExtension()` is used, `OS_TLS_Set()` will be called automatically.

Please ensure sufficient task stack if the `_reent` structure variable is placed on the task stack. For details on the `_reent` structure, `_impure_ptr`, and library functions which require precautions on reentrancy, refer to the GNU documentation.

Example

```
static void Task(void) {
  struct _reent TaskReentStruct;

  OS_TLS_Set(&TaskReentStruct);
  OS_TASK_SetContextExtension(&OS_TLS_ContextExtension);
  while (1) {
  }
}
```

4.3.2 OS_TLS_SetTaskContextExtension()

Description

`OS_TLS_SetTaskContextExtension()` may be called from a task to initialize thread-local storage for the current task and set the respective task context extension.

Prototype

```
void OS_TLS_SetTaskContextExtension(struct _reent* pReentStruct);
```

Parameters

Parameter	Description
`pReentStruct`	Pointer to the thread local storage. It is the address of the variable of type struct `_reent` which holds the thread local data.

Additional information

`OS_TLS_SetTaskContextExtension()` shall be the first function called from a task when TLS should be used in the specific task. If the task already contains a task context extension, `OS_TLS_SetTaskContextExtension()` cannot be used. Instead, `OS_TASK_AddContextExtension()` needs to be called with `OS_TLS_ContextExtension` as argument. Furthermore, `OS_TLS_Set()` needs to be called to initialize TLS for this task.

Please ensure sufficient task stack if the `_reent` structure variable is placed on the task stack. For details on the `_reent` structure, `_impure_ptr`, and library functions which require precautions on reentrancy, refer to the GNU documentation.

Example

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

```
#include "RTOS.h"

static OS_STACKPTR int StackHP[512], StackLP[512];  // Task stacks
static OS_TASK         TCBHP, TCBLP;                // Task control blocks

static void HPTask(void) {
  struct _reent TaskReentStruct;

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

static void LPTask(void) {
  struct _reent TaskReentStruct;

  OS_TLS_SetTaskContextExtension(&TaskReentStruct);
  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;
}
```

4.4 RISC-V privilege levels

The *RISC-V Privileged Architecture Version 1.10* defines three distinct privilege levels:

- Machine mode
- Supervisor mode
- User mode

Only *machine mode* is mandatory when implementing the architecture; it constitutes the highest privilege level. *User mode* and *supervisor mode* are intended for conventional application and *Unix*-like operating system usage, respectively. A fourth privilege level, *hypervisor mode*, existed in *Version 1.9.1 of the Privileged Architecture*, but was subsequently removed.

embOS for RISC-V currently supports *machine mode* only.

4.5 RISC-V harts

A RISC-V-compatible core might support multiple RISC-V-compatible hardware threads (often referred to as *"harts"*) through multi-threading.
Each hart is assigned an ID, which might not necessarily be numbered contiguously. However, at least one hart must have a hart ID of 0.

Applications executing on RISC-V platforms implementing multiple harts must be aware of the executing hart.
For example, a reset handler shall be executed by one hart only and therefore must have access to the current hart's ID to ensure this. Another example is accessing memory-mapped registers like *mtimecmp* (used to generate *machine timer interrupts* using the RISC-V real-time counter): Here, applications must use a memory offset specific to the hart that executes the application to ensure the interrupt request is generated on that same hart.
For this purpose, embOS for RISC-V offers a specific API function.

4.5.1 API functions for hart identification

Function	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
`OS_GetHartID()`	Returns the ID of the executing hart.	●	●	●	●	●

4.5.1.1 OS_GetHartID()

Description

`OS_GetHartID()` returns the ID of the executing hart.

Prototype

```
OS_REG_TYPE OS_GetHartID(void);
```

Return value

ID of the executing hart.

Example

```
//
// Set MTIMECMP register for this specific hart to 1000 cycles.
//
(*(unsigned long long*)(MTIMECMP_BASE_ADDR + (8u * OS_GetHartID()))) = 1000uL;
```

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.

In order to perform overflow checks on the system check and to provide stack usage information, embOS requires the symbols `__stack_start__` and `__stack_end__` to be appropriately defined in the respective project's linker file.

Alternatively, if modification of the linker file is not feasible (e.g. when it is auto-generated by a project generator), the stack symbols may also be defined as additional linker flags. For example, if the linker file defines the symbols *`LOWEST_STACK_ADDRESS`* and *`HIGHEST_STACK_ADDRESS`*, the linker flags would need to be set as shown below:

- --defsym=`__stack_start__`=LOWEST_STACK_ADDRESS
- --defsym=`__stack_end__`=HIGHEST_STACK_ADDRESS

5.3 Interrupt stack

RISC-V does not support a dedicated hardware interrupt stack. This means that any interrupt might use any task stack or the system stack depending on which context it is interrupting. Consequently, each task stack would need to be large enough to handle (multiple) interrupts. However, since assigning additional memory to each individual task stack would consume large amounts of RAM, embOS for RISC-V offers API functions to switch to the system stack on interrupt entry.

The respective functions `OS_INT_EnterIntStack()` and `OS_INT_LeaveIntStack()` are described in the generic embOS manual.

Chapter 6

Interrupts

6.1 RISC-V interrupt sources

The *RISC-V Privileged Architecture Version 1.10* defines 16 generic *core-local interrupt sources*.
Of these, 3 address *machine mode* and are mandatory when implementing the architecture, while further 6 are mandatory only when their respective privilege level is implemented (i.e. 3 sources with *user mode* and 3 sources with *supervisor mode*). The remaining 7 generic core-local interrupt sources must not be implemented, but are *reserved* for future use.

In addition to these generic core-local interrupt sources, further core-local interrupt sources may be implemented with any specific RISC-V platform (up to 16 with *RV32I*).

Consequently, any RISC-V-compliant platform includes one *software interrupt*, one *timer interrupt* and one *external interrupt* for each privilege level it implements, as well as a variable number of platform-specific core-local interrupts.

While the *timer interrupt* serves interrupt requests generated by any RISC-V platforms' mandatory real-time counter, the *software interrupt*, as its name suggests, serves interrupt requests generated by software.
The *external interrupt*, on the other hand, is used to serve a variable number of *global interrupts* which themselves are managed by a dedicated interrupt controller. Some implementors of the architecture include core-local interrupt management with the same interrupt controller, but most often core-local interrupt sources are managed locally.

By default, all interrupts (often referred to as *"traps"*) are served in *machine mode*.
Although *machine mode* interrupt service routines could technically redirect interrupts to the appropriate mode, this currently is not supported with embOS for RISC-V. Neither is the *"Machine Trap Delegation"* hardware feature.

6.2 RISC-V interrupt priorities

Multiple simultaneous interrupts at the same privilege level are handled in the following decreasing priority order: External interrupts, software interrupts, timer interrupts, then finally any synchronous traps.

External interrupts may further be prioritized by the dedicated interrupt controller depending on its implementation, while the priority of non-standard core-local interrupt sources relative to external, timer, and software interrupt sources is platform-specific.

For example, with *SiFive's "RISC-V Coreplex IP"*, the platform-specific core-local interrupt sources take precedence over any other interrupt source and are themselves prioritized by their index. Considering *machine mode* only, a comprehensive priority table for that platform (in decreasing order of priority) would therefore read as follows:

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

6.3 Zero-latency interrupts

Zero-latency interrupts are not supported with the current version of embOS for RISC-V.

6.4 RISC-V core-local interrupt modes

Typically, core-local interrupt handling is performed in *direct mode*:
In this mode, a RISC-V platform will route all core-local interrupts through a low-level interrupt service routine, which ultimately calls the appropriate high-level interrupt service routines for the respective core-local interrupt sources. For this purpose, the low-level interrupt service routine's address needs to be held in the *mtvec* register, which typically is set during start-up.

Alternatively, core-local interrupt handling may also be performed in *vectored mode*:
In this mode, a RISC-V platform will route all core-local interrupts through a properly aligned vector table containing jumps to the appropriate interrupt service routines for the respective core-local interrupt sources. For this purpose, the vector table's base address needs to be held in the *mtvec* register, which then typically needs to be set explicitly by the application. The application then also needs to tell the hardware to utilize vectored mode by setting the least-significant bit of that register.

Note

In vectored mode, both *synchronous exceptions* and *user mode software interrupts* are ambiguously vectored to the same exception handler.

6.5 Interrupt handling with embOS for RISC-V

Core-local interrupt handling

Addressing core-local interrupt handling, embOS for RISC-V offers API functions for:

- Generic RISC-V core-local interrupt handling (refer to *Core-local interrupt handling* on page 35)
- *NucleiSys' "Enhanced Core-Local Interrupt Controller"* (refer to *Core-local and global interrupt handling using ECLIC* on page 56)

When using embOS API functions on core-local interrupt sources, these may be specified using the following enumeration (where missing numerical values indicate *reserved* core-local interrupt sources):

Core-local interrupt source	Numerical value
`IRQ_U_SOFTWARE`	0
`IRQ_S_SOFTWARE`	1
`IRQ_M_SOFTWARE`	3
`IRQ_U_TIMER`	4
`IRQ_S_TIMER`	5
`IRQ_M_TIMER`	7
`IRQ_U_EXTERNAL`	8
`IRQ_S_EXTERNAL`	9
`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

Global interrupt handling

Addressing global interrupt handling, embOS for RISC-V offers API functions for:

- Generic *"RISC-V Platform-Level Interrupt Controller"* implementations (refer to *Global interrupt handling using PLIC* on page 41)
- *Lattice's "Programmable Interrupt Controller"* (refer to *Global interrupt handling using PIC* on page 47)
- *NucleiSys' "Enhanced Core-Local Interrupt Controller"* (refer to *Core-local and global interrupt handling using ECLIC* on page 56)

6.5.1 Core-local interrupt handling

6.5.1.1 Implementing core-local interrupt handlers in "C"

6.5.1.1.1 Low-level interrupt service routine

In direct mode, the individual interrupt service routines for distinct core-local interrupt sources need to be dispatched by a common low-level service routine. With embOS for RISC-V, this low-level routine is split into an assembler part called *trap_entry()*, which saves and restores the interrupted context, and a *"C"*-function called `OS_TrapHandler()`, which performs the actual dispatching of high-level service routines.
In vectored mode, embOS for RISC-V will not call *trap_entry()* at all, while `OS_TrapHandler()` is called exclusively to handle *synchronous traps* and *user mode software interrupts* (which ambiguously share the same vector in that mode).

embOS for RISC-V sample projects will typically implement `OS_TrapHandler()` in their respective *RTOSInit*.c* as shown in the example below, allowing for customization of that function. As this exemplary implementation of `OS_TrapHandler()` does not call any embOS API functions, it does not need to include a prologue and an epilogue as described in the generic embOS manual.

Example

```
#if (USE_VECTORED_INT_MODE == 0)
static OS_IRQ_HANDLER* _apfIRQHandler[NUM_LOCAL_INTERRUPTS];
#endif

OS_REG_TYPE OS_TrapHandler(OS_REG_TYPE mcause, OS_REG_TYPE mepc) {
  if (mcause & MCAUSE_INT) {
#if (USE_VECTORED_INT_MODE == 0)
    //
    // Caused by interrupt: call appropriate high-level handler.
    //
    _apfIRQHandler[mcause & MCAUSE_CAUSE]();
#else
    //
    // In vectored mode, user mode software interrupt ambiguously shares
    // a vector with synchronous exceptions. If user mode software interrupt
    // is to be used by the application, its handler could be called here.
    //
    _ISR_NotInstalled();
#endif
  } else {
    //
    // Caused by synchronous trap: call fault handler.
    //
    _ExceptionHandler(mcause, mepc);
  }
  return mepc;
}
```

Note

mepc contains the address of the instruction that was executed when the interrupt was taken. It must eventually be returned by `OS_TrapHandler()` to continue regular program execution at that address once the interrupt completes.
In case of interrupts, *mepc* must never be modified by `OS_TrapHandler()` before returning it. In case of a synchronous traps, however, *mepc* can be used to examine the cause for the trap and to react accordingly (e.g. redirecting program execution elsewhere).

6.5.1.1.2 High-level interrupt service routines

The individual interrupt service routines for distinct core-local interrupt sources shall include a prologue and an epilogue as described in the generic embOS manual (both in direct mode and in vectored mode). A high-level service routine for any core-local interrupt source may therefore be implemented as shown in the example below.

Example

```
void ISR_Local0(void) {
  OS_INT_Enter();
  //
  // Perform any functionality here.
  //
  OS_INT_Leave();
}
```

> **Note**
>
> A CLINT’s high-level interrupt service routine for *machine external interrupts* is, at the same time, the low-level interrupt service routine for the (external) interrupt controller (e.g. PLIC).

6.5.1.2 API functions for core-local interrupt handling

For core-local interrupt handling, embOS offers the following functions:

Function	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
`OS_CLINT_ClearIntPending()`	Clears pending state of the specified core-local interrupt source.	●	●		●	●
`OS_CLINT_DisableInt()`	Disables the specified core-local interrupt source.	●	●		●	●
`OS_CLINT_EnableInt()`	Enables the specified core-local interrupt source.	●	●		●	●
`OS_CLINT_GetIntPending()`	Returns the current pending status of the specified core-local interrupt source.	●	●	●	●	●
`OS_CLINT_Init()`	Initializes core-local interrupt handling.	●				
`OS_CLINT_InstallISR()`	Installs the specified interrupt service routine in a RAM vector table.	●	●			
`OS_CLINT_SetIntPending()`	Sets the specified core-local interrupt source to pending state.	●	●		●	●
`OS_CLINT_SetDirectMode()`	Configures core-local interrupt handling to direct mode.	●				
`OS_CLINT_SetVectoredMode()`	Configures core-local interrupt handling to vectored mode.	●				

6.5.1.2.1 OS_CLINT_ClearIntPending()

Description

`OS_CLINT_ClearIntPending()` clears pending state of the specified core-local interrupt source.
Machine external and machine timer interrupt pending bits are read-only. The primary use of this function therefore is to clear machine software interrupts.

Prototype

```
void OS_CLINT_ClearIntPending(CLINT_IRQn IRQIndex);
```

Parameters

Parameter	Description
`IRQIndex`	Specifies the core-local interrupt source by its index.

6.5.1.2.2 OS_CLINT_DisableInt()

Description

`OS_CLINT_DisableInt()` disables the specified core-local interrupt source.

Prototype

```
void OS_CLINT_DisableInt(CLINT_IRQn IRQIndex);
```

Parameters

Parameter	Description
`IRQIndex`	Specifies the core-local interrupt source by its index.

6.5.1.2.3 OS_CLINT_EnableInt()

Description

`OS_CLINT_EnableInt()` enables the specified core-local interrupt source.

Prototype

```
void OS_CLINT_EnableInt(CLINT_IRQn IRQIndex);
```

Parameters

Parameter	Description
`IRQIndex`	Specifies the core-local interrupt source by its index.

6.5.1.2.4 OS_CLINT_GetIntPending()

Description

`OS_CLINT_GetIntPending()` returns the current pending state of the specified core-local interrupt source.

Prototype

```
OS_BOOL OS_CLINT_GetIntPending(CLINT_IRQn IRQIndex);
```

Parameters

Parameter	Description
`IRQIndex`	Specifies the core-local interrupt source by its index.

Return value

= 0: Specified interrupt source is not pending.
= 1: Specified interrupt source is pending.

6.5.1.2.5 OS_CLINT_Init()

Description

`OS_CLINT_Init()` initializes core-local interrupt handling.
Must be called *prior* to `OS_Start()` and *before* calling any other `OS_CLINT_*()` function.

Prototype

```
void OS_CLINT_Init(OS_U8           NumInterrupts,
                   OS_IRQ_HANDLER* apfISR[]);
```

Parameters

Parameter	Description
`NumInterrupts`	Number of supported core-local interrupt sources. Requires a minimum of 16 and may not exceed 32 (with *RV32I*).
`apfISR`	Pointer to a RAM vector table base. When using vectored mode or a ROM vector table in direct mode, this parameter must be `NULL`.

6.5.1.2.6 OS_CLINT_InstallISR()

Description

`OS_CLINT_InstallISR()` installs the specified interrupt service routine for the specified core-local interrupt source in a RAM vector table that was configured via `OS_CLINT_Init()`. This function must not be called when using vectored mode or a ROM vector table in direct mode.

Prototype

```
OS_IRQ_HANDLER* OS_CLINT_InstallISR(CLINT_IRQn      IRQIndex,
                                    OS_IRQ_HANDLER* pfISR);
```

Parameters

Parameter	Description
`IRQIndex`	Specifies the core-local interrupt source by its index.
`pfISR`	Pointer to the interrupt service routine to be installed.

Return value

Pointer to the previously installed interrupt service routine.

6.5.1.2.7 OS_CLINT_SetIntPending()

Description

`OS_CLINT_SetIntPending()` sets the specified core-local interrupt source to pending state. Machine external and machine timer interrupt pending bits are read-only. The primary use of this function therefore is to trigger machine software interrupts.

Prototype

```
void OS_CLINT_SetIntPending(CLINT_IRQn IRQIndex);
```

Parameters

Parameter	Description
`IRQIndex`	Specifies the core-local interrupt source by its index.

6.5.1.2.8 OS_CLINT_SetDirectMode()

Description

Configures core-local interrupt handling to direct mode.
Must be called prior to `OS_Start()`.

Prototype

```
void OS_CLINT_SetDirectMode(void);
```

6.5.1.2.9 OS_CLINT_SetVectoredMode()

Description

Configures core-local interrupt handling to vectored mode.
Must be called prior to `OS_Start()` and expects the vector table to be called *`vtrap_entry`*.

Prototype

```
void OS_CLINT_SetVectoredMode(void);
```

6.5.2 Global interrupt handling using PLIC

The generic *"RISC-V Platform-Level Interrupt Controller" (PLIC)* is defined by the *RISC-V Privileged Architecture Version 1.10*.

6.5.2.1 Implementing global interrupt handlers in "C"

6.5.2.1.1 Low-level interrupt service routine

The individual interrupt service routines for global interrupt sources need to be dispatched by a common low-level service routine. Since this low-level service routine needs to call embOS API functions, it must include a prologue and an epilogue as described in the generic embOS manual. Typically, embOS for RISC-V sample projects using PIC will implement the low-level service routine in their respective *RTOSInit*.c* as shown in the example below.

```
static OS_IRQ_HANDLER* _apfIRQHandler[PLIC_NUM_INTERRUPTS];

void ISR_M_External(void) {
  OS_U32 IRQIndex;

  OS_INT_Enter();
  IRQIndex = OS_PLIC_ClaimInt();    // Claim highest-priority global IRQ.
  if (IRQIndex != 0u) {             // "0" indicates no IRQ was pending.
    _apfIRQHandler[IRQIndex]();     // Call appropriate handler.
    OS_PLIC_CompleteInt(IRQIndex);  // Signal interrupt completion to PLIC.
  }
  OS_INT_Leave();
}
```

Note

A low-level interrupt service routine for the PLIC is, at the same time, the CLINT high-level interrupt service routine for *machine external interrupts*.

6.5.2.1.2 High-level interrupt service routines

The individual interrupt service routines for distinct global interrupt sources do not need to include a prologue and an epilogue as described in the manual, since these were already included in the low-level service routine. A high-level service routine for any global interrupt source may therefore be implemented as shown in the example below.

```
void ISR_External_S0(void) {
  //
  // Perform any functionality here.
  //
}
```

6.5.2.2 API functions for using PLIC

Function	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
`OS_PLIC_ClaimInt()`	Retrieves the index of highest-priority pending global interrupt and clears pending condition				●	
`OS_PLIC_CompleteInt()`	Notifies PLIC of ISR completion				●	
`OS_PLIC_DisableInt()`	Disables the specified global interrupt source	●	●		●	●
`OS_PLIC_EnableInt()`	Enables the specified global interrupt source	●	●		●	●
`OS_PLIC_GetIntPriority()`	Returns the current interrupt priority for the specified interrupt source	●	●	●	●	●
`OS_PLIC_GetIntThreshold()`	Returns the current interrupt priority threshold	●	●	●	●	●
`OS_PLIC_Init()`	Configures PLIC base address and RAM vector table address	●				
`OS_PLIC_InstallISR()`	Installs an global interrupt handler	●	●			
`OS_PLIC_SetIntPriority()`	Sets the priority of the specified global interrupt	●	●		●	●
`OS_PLIC_SetIntThreshold()`	Configures the IRQ threshold, masking lower-priority global interrupts	●	●		●	●

6.5.2.2.1 OS_PLIC_ClaimInt()

Description

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

Prototype

```
OS_U32 OS_PLIC_ClaimInt(void);
```

Return value

`OS_U32`: Interrupt index

6.5.2.2.2 OS_PLIC_CompleteInt()

Description

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

Prototype

```
void OS_PLIC_CompleteInt(OS_U32 IRQIndex);
```

Parameters

Parameter	Description
`IRQIndex`	Interrupt index

6.5.2.2.3 OS_PLIC_DisableInt()

Description

`OS_PLIC_DisableInt()` is used to disable the specified global interrupt.

Prototype

```
void OS_PLIC_DisableInt(OS_U32 IRQIndex);
```

Parameters

Parameter	Description
`IRQIndex`	Interrupt index

6.5.2.2.4 OS_PLIC_EnableInt()

Description

`OS_PLIC_EnableInt()` is used to enable the specified global interrupt.

Prototype

```
void OS_PLIC_EnableInt(OS_U32 IRQIndex);
```

Parameters

Parameter	Description
`IRQIndex`	Interrupt index

6.5.2.2.5 OS_PLIC_GetIntPriority()

Description

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

Prototype

```
OS_U32 OS_PLIC_GetIntPriority(OS_U32 IRQIndex);
```

Parameters

Parameter	Description
`IRQIndex`	Interrupt index

Return value

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

6.5.2.2.6 OS_PLIC_GetIntThreshold()

Description

`OS_PLIC_GetIntThreshold()` retrieves the current global interrupt priority threshold.

Prototype

```
OS_U32 OS_PLIC_GetIntThreshold(void);
```

Return value

`OS_U32`: Current interrupt priority threshold

6.5.2.2.7 OS_PLIC_Init()

Description

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

Prototype

```
void OS_PLIC_Init(OS_U32          BaseAddr,
                  OS_U16          NumInterrupts,
                  OS_U32          NumPriorities,
                  OS_IRQ_HANDLER* apfISR[]);
```

Parameters

Parameter	Description
`BaseAddr`	PLIC base address
`NumInterrupts`	Number of supported global interrupt sources
`NumPriorities`	Number of supported global interrupt priorities
`apfISR`	Pointer to RAM vector table base

6.5.2.2.8 OS_PLIC_InstallISR()

Description

`OS_PLIC_InstallISR()` is used to install the specified global interrupt handler in the RAM vector table.

Prototype

```
OS_IRQ_HANDLER* OS_PLIC_InstallISR(OS_U32          IRQIndex,
                                   OS_IRQ_HANDLER* pfISR);
```

Parameters

Parameter	Description
`IRQIndex`	Interrupt index
`pfISR`	Address of interrupt handler

Return value

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

6.5.2.2.9 OS_PLIC_SetIntPriority()

Description

`OS_PLIC_SetIntPriority()` is used to configure the interrupt priority for the specified global interrupt.

Prototype

```
OS_U32 OS_PLIC_SetIntPriority(OS_U32 IRQIndex,
                              OS_U32 Prio);
```

Parameters

Parameter	Description
`IRQIndex`	Interrupt index
`Prio`	Interrupt priority

Return value

`OS_U32`: Previous priority which was assigned before

6.5.2.2.10 OS_PLIC_SetIntThreshold()

Description

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

Prototype

```
void OS_PLIC_SetIntThreshold(OS_U32 Threshold);
```

Parameters

Parameter	Description
`Threshold`	Desired interrupt priority threshold

6.5.3 Global interrupt handling using PIC

6.5.3.1 Global interrupt sources

An implementation of *Lattice's "Programmable Interrupt Controller" (PIC)* may support a minimum of 1 and a maximum of 8 global interrupt sources. When using embOS API functions for global interrupt sources, these may be specified using the following enumeration:

Global interrupt source	Numerical value
`IRQ_S0`	0
`IRQ_S1`	1
`IRQ_S2`	2
`IRQ_S3`	3
`IRQ_S4`	4
`IRQ_S5`	5
`IRQ_S6`	6
`IRQ_S7`	7

6.5.3.2 Global interrupt priority

With a *PIC* implementation, all global interrupt sources are executed at the same priority. Which interrupt service routine is executed first when several interrupts are pending at the same time depends on the low-level interrupt service routine.

6.5.3.3 Global interrupt polarity

The polarity of any global interrupt source may be configured using the following enumeration:

PIC interrupt polarity	Numerical value
`PIC_INTPOLARITY_HIGH`	0
`PIC_INTPOLARITY_LOW`	1

6.5.3.4 Implementing global interrupt handlers in "C"

6.5.3.4.1 Low-level interrupt service routine

The individual interrupt service routines for global interrupt sources need to be dispatched by a common low-level service routine. Since this low-level service routine needs to call embOS API functions, it must include a prologue and an epilogue as described in the generic embOS manual. Typically, embOS for RISC-V sample projects using PIC will implement the low-level service routine in their respective *RTOSInit*.c* as shown in the example below.

```
#if (EXTEND_GLOBAL_ISR_CONTEXT == 0)
static OS_IRQ_HANDLER*         _apfIRQHandler[PIC_NUM_INTERRUPTS];
#else
static OS_IRQ_HANDLER_CONTEXT* _apfIRQHandler[PIC_NUM_INTERRUPTS];
#endif

void ISR_M_External(void) {
  PIC_IRQn IRQIndex;

  OS_INT_Enter();
  //
  // By sequentially serving all pending global interrupts at once, this
  // exemplary implementation aims at accelerating interrupt handling since
  // interrupted contexts do not need to be saved and restored repeatedly.
  // By iterating from IRQ_S0 to PIC_NUM_INTERRUPTS, this exemplary implementation
  // prioritizes global interrupt sources by their index (in ascending order).
  //
  for (IRQIndex = IRQ_S0; IRQIndex < PIC_NUM_INTERRUPTS; IRQIndex++) {
    if (OS_PIC_GetIntPending(IRQIndex) == 1u) {
#if (EXTEND_GLOBAL_ISR_CONTEXT == 0)
      _apfIRQHandler[IRQIndex]();
#else
      _apfIRQHandler[IRQIndex]->pfISR(_apfIRQHandler[IRQIndex]->pContext);
#endif
      OS_PIC_ClearIntPending(IRQIndex);
    }
  }
  OS_INT_Leave();
}
```

Note

A low-level interrupt service routine for the PIC is, at the same time, the CLINT high-level interrupt service routine for *machine external interrupts*.

6.5.3.4.2 High-level interrupt service routines

The individual interrupt service routines for distinct global interrupt sources do not need to include a prologue and an epilogue as described in the manual, since these were already included in the low-level service routine. A high-level service routine for any global interrupt source may therefore be implemented as shown in the example below.

```
#if (EXTEND_GLOBAL_ISR_CONTEXT == 0)
void ISR_External_S0(void) {
#else
void ISR_External_S0(void* pContext) {
#endif
  //
  // Perform any functionality here.
  //
}
```

6.5.3.5 API functions for using PIC

Function	Description	main	Priv Task	Unpriv Task	ISR	SW Timer
OS_PIC_ClearIntPending()	Clears pending state of the specified global interrupt source.	●	●		●	●
OS_PIC_DisableInt()	Disables the specified global interrupt source.	●	●		●	●
OS_PIC_EnableInt()	Disables the specified global interrupt source.	●	●		●	●
OS_PIC_GetIntPending()	Returns the current pending status of the specified global interrupt source.	●	●	●	●	●
OS_PIC_GetIntPolarity()	Returns the current polarity of the specified global interrupt source.	●	●	●	●	●
OS_PIC_Init()	Initializes PIC interrupt handling.	●				
OS_PIC_Init_Ex()	Initializes extended PIC interrupt handling.	●				
OS_PIC_InstallISR()	Installs the specified interrupt service routine in a RAM vector table.	●	●			
OS_PIC_InstallISR_Ex()	Installs the specified extended interrupt service routine in a RAM vector table.	●	●			
OS_PIC_SetIntPending()	Sets the specified global interrupt source to pending state.	●	●		●	●
OS_PIC_SetIntPolarity()	Sets the specified polarity for the specified global interrupt source.	●	●		●	●

6.5.3.5.1 OS_PIC_ClearIntPending()

Description

`OS_PIC_ClearIntPending()` clears pending state of the specified global interrupt source.

Prototype

```
void OS_PIC_ClearIntPending(PIC_IRQn IRQIndex);
```

Parameters

Parameter	Description
`IRQIndex`	Specifies the interrupt source by its index.

6.5.3.5.2 OS_PIC_DisableInt()

Description

`OS_PIC_DisableInt()` disables the specified global interrupt source.

Prototype

```
void OS_PIC_DisableInt(PIC_IRQn IRQIndex);
```

Parameters

Parameter	Description
`IRQIndex`	Specifies the interrupt source by its index.

6.5.3.5.3 OS_PIC_EnableInt()

Description

`OS_PIC_EnableInt()` enables the specified global interrupt source.

Prototype

```
void OS_PIC_EnableInt(PIC_IRQn IRQIndex);
```

Parameters

Parameter	Description
`IRQIndex`	Specifies the interrupt source by its index.

6.5.3.5.4 OS_PIC_GetIntPending()

Description

`OS_PIC_GetIntPending()` returns the current pending status of the specified global interrupt source.

Prototype

```
OS_BOOL OS_PIC_GetIntPending(PIC_IRQn IRQIndex);
```

Parameters

Parameter	Description
`IRQIndex`	Specifies the interrupt source by its index.

Return value

= 0: Specified interrupt source is not pending.
= 1: Specified interrupt source is pending.

6.5.3.5.5 OS_PIC_GetIntPolarity()

Description

`OS_PIC_GetIntPolarity()` returns the currently configured polarity of the specified global interrupt source.

Prototype

```
PIC_INTPOLARITY OS_PIC_GetIntPolarity(PIC_IRQn IRQIndex);
```

Parameters

Parameter	Description
`IRQIndex`	Specifies the interrupt source by its index.

Return value

= `OS_PIC_INTPOLARITY_HIGH`: Specified interrupt source is configured to active high.
= `OS_PIC_INTPOLARITY_LOW`: Specified interrupt source is configured to active low.

6.5.3.5.6 OS_PIC_Init()

Description

`OS_PIC_Init()` initializes PIC interrupt handling.
Must not be called when using (`OS_PIC_Init_Ex()`, but must otherwise be called *prior* to `OS_Start()` and *before* calling any other `OS_PIC_*()` function.

Prototype

```
void OS_PIC_Init(OS_U32          BaseAddr,
                 OS_U8           NumInterrupts,
                 OS_IRQ_HANDLER* apfISR[]);
```

Parameters

Parameter	Description
`BaseAddr`	PIC base address.
`NumInterrupts`	Number of supported global interrupt sources. Requires a minimum of 1 and may not exceed 8.
`apfISR`	Pointer to a RAM vector table base. When using a ROM vector table, this parameter must be `NULL`.

Example

```
static OS_IRQ_HANDLER* _apfIRQHandler[PIC_NUM_INTERRUPTS];
void foo(void) {
  OS_PIC_Init(PIC_BASE_ADDR, PIC_NUM_INTERRUPTS, _apfIRQHandler);
}
```

6.5.3.5.7 OS_PIC_Init_Ex()

Description

`OS_PIC_Init_Ex()` initializes extended PIC interrupt handling.
Must not be called when using (`OS_PIC_Init()`, but must otherwise be called *prior* to `OS_Start()` and *before* calling any other `OS_PIC_*()` function.

Prototype

```
void OS_PIC_Init_Ex(OS_U32                  BaseAddr,
                    OS_U8                   NumInterrupts,
                    OS_IRQ_HANDLER_CONTEXT* apfISR[]);
```

Parameters

Parameter	Description
`BaseAddr`	PIC base address.
`NumInterrupts`	Number of supported global interrupt sources. Requires a minimum of 1 and may not exceed 8.
`apfISR`	Pointer to an extended RAM vector table base. When using a ROM vector table, this parameter must be `NULL`.

Example

```
static OS_IRQ_HANDLER_CONTEXT* _apfIRQHandler[PIC_NUM_INTERRUPTS];
void foo(void) {
  OS_PIC_Init_Ex(PIC_BASE_ADDR, PIC_NUM_INTERRUPTS, _apfIRQHandler);
}
```

6.5.3.5.8 OS_PIC_InstallISR()

Description

`OS_PIC_InstallISR()` installs the specified interrupt service routine for the specified global interrupt source in a RAM vector table that was configured via `OS_PIC_Init()`.
This function must not be called when using a ROM vector table.

Prototype

```
OS_IRQ_HANDLER* OS_PIC_InstallISR(PIC_IRQn        IRQIndex,
                                  OS_IRQ_HANDLER* pfISR);
```

Parameters

Parameter	Description
`IRQIndex`	Specifies the interrupt source by its index.
`pfISR`	Pointer to the interrupt service routine to be installed.

Return value

Pointer to the previously installed interrupt service routine.

Example

```
static void _ISR_External2(void) {
  //
  // Perform any functionality.
  //
}

void foo(void) {
  (void)OS_PIC_InstallISR(IRQ_S2, _ISR_External2);
}
```

6.5.3.5.9 OS_PIC_InstallISR_Ex()

Description

`OS_PIC_InstallISR_Ex()` installs the specified extended interrupt service routine for the specified global interrupt source in a RAM vector table that was configured via `OS_PIC_Init_Ex()`.
This function must not be called when using a ROM vector table.

Prototype

```
OS_IRQ_HANDLER_EX* OS_PIC_InstallISR_Ex(PIC_IRQn           IRQIndex,
                                        OS_IRQ_HANDLER_EX* pfISR,
                                        void*              pContext);
```

Parameters

Parameter	Description
`IRQIndex`	Specifies the interrupt source by its index.
`pfISR`	Pointer to the extended interrupt service routine to be installed.
`pContext`	Pointer to the context that should be passed to the interrupt service routine.

Return value

Pointer to the previously installed extended interrupt service routine.

Example

```
static void _ISR_External2(void* pContext) {
  if (((int)pContext) == 42) {
    //
    // Perform some functionality.
    //
  } else {
    //
    // Perform some other functionality.
    //
  }
}

void foo(void) {
  (void)OS_PIC_InstallISR_Ex(IRQ_S2, _ISR_External2, (void*)42);
}
```

6.5.3.5.10 OS_PIC_SetIntPending()

Description

`OS_PIC_SetIntPending()` sets the specified global interrupt source to pending state.

Prototype

```
void OS_PIC_SetIntPending(PIC_IRQn IRQIndex);
```

Parameters

Parameter	Description
`IRQIndex`	Specifies the interrupt source by its index.

6.5.3.5.11 OS_PIC_SetIntPolarity()

Description

`OS_PIC_SetIntPolarity()` configures the specified polarity for the specified global interrupt source.

Prototype

```
void OS_PIC_SetIntPolarity(PIC_IRQn        IRQIndex,
                           PIC_INTPOLARITY Polarity);
```

Parameters

Parameter	Description
`IRQIndex`	Specifies the interrupt source by its index.
`Polarity`	Polarity to configure (`PIC_INTPOLARITY_HIGH` or `PIC_INTPOLARITY_LOW`).

6.5.4 Core-local and global interrupt handling using ECLIC

When using *NucleiSys' "Enhanced Core-Local Interrupt Controller" (ECLIC)*, both core-local and global interrupt sources are managed by the ECLIC and behave the same way.

6.5.4.1 Interrupt levels and priorities

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, resulting in interrupt nesting. Furthermore, interrupts can be nested by synchronous exceptions. 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 pending interrupts with the same interrupt level.

6.5.4.2 API functions for using ECLIC

To handle ECLIC interrupts, embOS offers the following functions:

Function	Description
OS_ECLIC_DisableInt()	Disables the specified interrupt source.
OS_ECLIC_EnableInt()	Enables the specified interrupt source.
OS_ECLIC_GetNumLevelBits()	Returns how many bits of the interrupt control register are used for the interrupt level.
OS_ECLIC_GetIntPriority()	Returns the interrupt control value of the specified interrupt.
OS_ECLIC_GetIntThreshold()	Returns the current interrupt level threshold.
OS_ECLIC_Init()	Initializes the ECLIC interrupt controller.
OS_ECLIC_SetNumLevelBits()	Specifies how many bits of the interrupt control register shall be used for the interrupt level.
OS_ECLIC_SetIntPriority()	Sets the interrupt control value of the specified interrupt.
OS_ECLIC_SetIntThreshold()	Configures the IRQ threshold, masking lower-level interrupts.

6.5.4.2.1 OS_ECLIC_DisableInt()

Description

`OS_ECLIC_DisableInt()` disables the specified interrupt.

Prototype

```
void OS_ECLIC_DisableInt(OS_UINT IRQIndex);
```

Parameters

Parameter	Description
IRQIndex	Interrupt index.

6.5.4.2.2 OS_ECLIC_EnableInt()

Description

`OS_ECLIC_EnableInt()` enables the specified interrupt.

Prototype

```
void OS_ECLIC_EnableInt(OS_UINT IRQIndex);
```

Parameters

Parameter	Description
IRQIndex	Interrupt index.

6.5.4.2.3 OS_ECLIC_GetNumLevelBits()

Description

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

Prototype

```
OS_U8 OS_ECLIC_GetNumLevelBits(void);
```

Return value

The number of level bits.

6.5.4.2.4 OS_ECLIC_GetIntPriority()

Description

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

Prototype

```
OS_U8 OS_ECLIC_GetIntPriority(OS_UINT IRQIndex);
```

Parameters

Parameter	Description
IRQIndex	Interrupt index.

Return value

The interrupt control value containing the interrupt level and priority.

6.5.4.2.5 OS_ECLIC_GetIntThreshold()

Description

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

Prototype

```
OS_U8 OS_ECLIC_GetIntThreshold(void);
```

Return value

The current interrupt level threshold.

6.5.4.2.6 OS_ECLIC_Init()

Description

`OS_ECLIC_Init()` initializes the ECLIC interrupt controller.

Prototype

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

Parameters

Parameter	Description
pBaseAddr	Base address of the memory mapped ECLIC SFRs.
pVectorTable	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).
pTrapHandler	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
```

6.5.4.2.7 OS_ECLIC_SetNumLevelBits()

Description

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

Prototype

```
void OS_ECLIC_SetNumLevelBits(OS_U8 NumLevelBits);
```

Parameters

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

6.5.4.2.8 OS_ECLIC_SetIntPriority()

Description

`OS_ECLIC_SetIntPriority()` 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_ECLIC_SetNumLevelBits()`.

Prototype

```
void OS_ECLIC_SetIntPriority(OS_UINT IRQIndex,
                             OS_U8   InterruptPriority);
```

Parameters

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

6.5.4.2.9 OS_ECLIC_SetIntThreshold()

Description

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

Prototype

```
void OS_ECLIC_SetIntThreshold(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.

6.6 Interrupt-stack switching

embOS for RISC-V offers API functions for interrupt stack switching. Please refer to chapter *Interrupt stack* on page 30 for more information.

Chapter 7

RTT and SystemView

7.1 SEGGER Real Time Transfer

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

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

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

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

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

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

Note

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

Chapter 8

Technical data

8.1 Resource Usage

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

Module	Memory type	Memory requirements
embOS kernel	ROM	~2000 bytes
embOS kernel	RAM	~136 bytes
Task control block	RAM	36 bytes
Software timer	RAM	20 bytes
Task event	RAM	0 bytes
Event object	RAM	12 bytes
Mutex	RAM	16 bytes
Semaphore	RAM	8 bytes
RWLocks	RAM	28 bytes
Mailbox	RAM	24 bytes
Queue	RAM	32 bytes
Watchdog	RAM	12 bytes
Fixed Block Size Memory Pool	RAM	32 bytes