embOS

Real-Time Operating System

CPU & Compiler specifics for embOS-MPU Linux Simulation

Document: UM01087
Software Version: 5.18.3.0
Revision: 0
Date: August 27, 2024

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

The information written in this document is assumed to be accurate without guarantee. The information in this manual is subject to change for functional or performance improvements without notice. SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions in this document. SEGGER disclaims any warranties or conditions, express, implied or statutory for the fitness of the product for a particular purpose. It is your sole responsibility to evaluate the fitness of the product for any specific use.

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: August 27, 2024

Software	Revision	Date	By	Description
5.18.3.0	0	240827	MM	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.

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

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

To install it, proceed as follows:

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

Assuming that you are using an IDE to develop your application, no further installation steps are required. You will find a prepared sample start project, 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 the library-file 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 scripts or build systems without any problem.

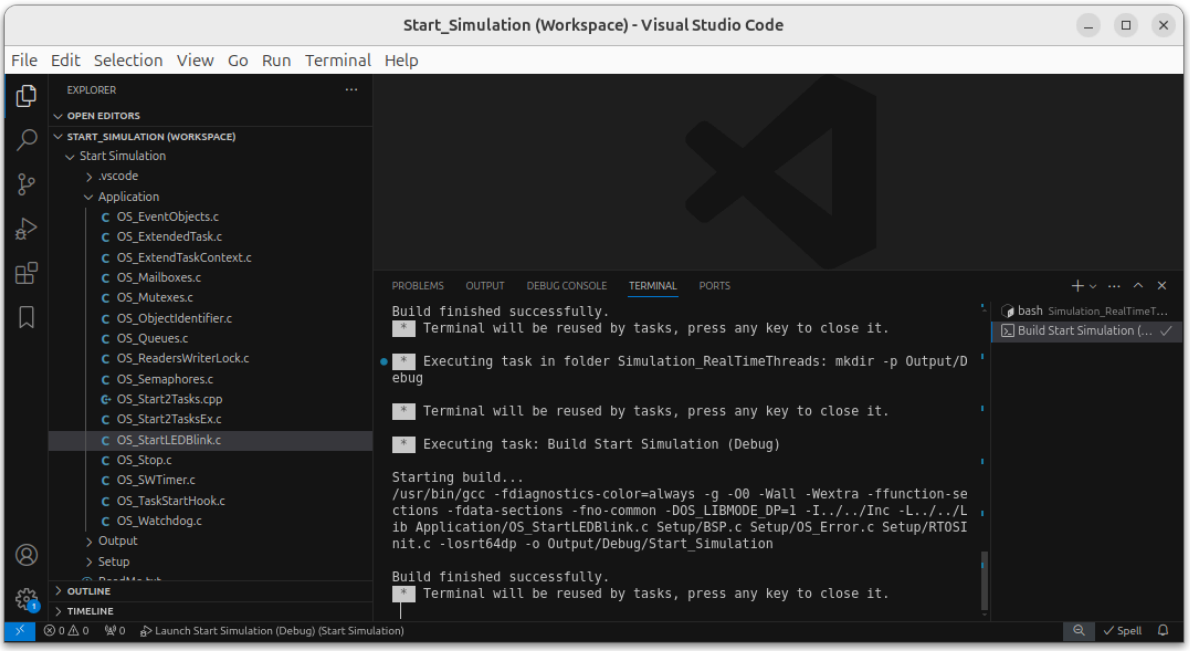
1.2 First Steps

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

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 the sample project in one of the BSPs in `Start\BoardSupport` 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:



1.3 The example application OS_StartLEDBlink.c

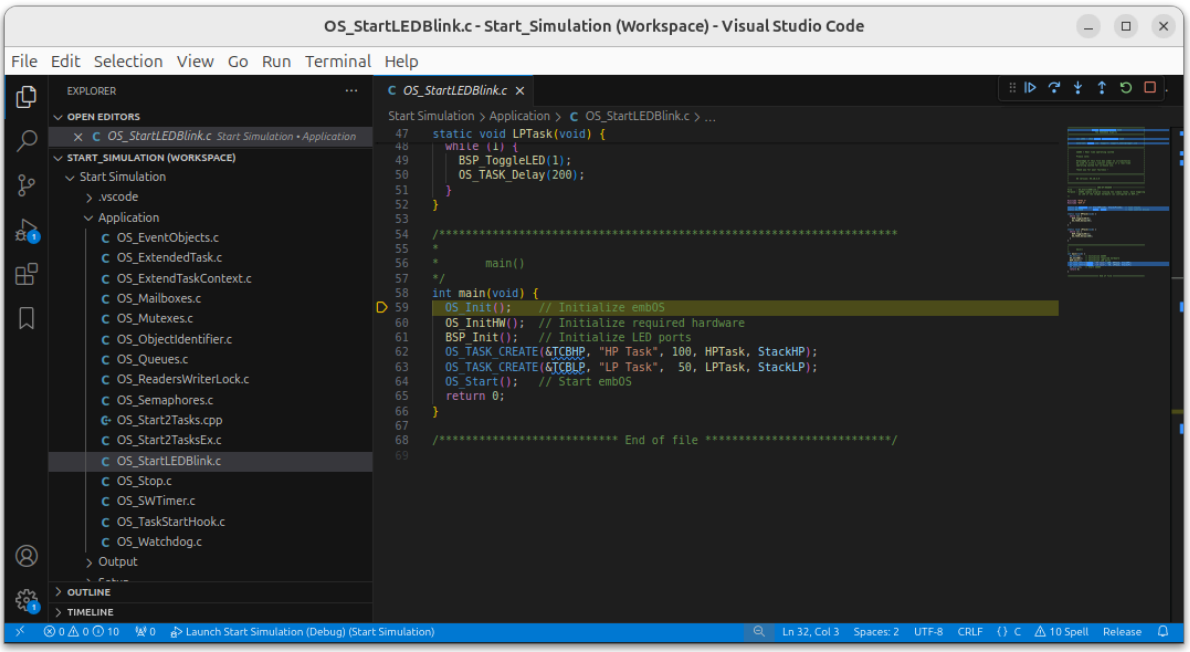
The following is a printout of the example application `OS_StartLEDBlink.c`. It is a good starting point for your application. Note that the file actually shipped with your port of embOS may look slightly different from this one.

What happens is easy to see:
After initialization of embOS two tasks are created and started. The two tasks are activated and executed until they run into the delay, 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

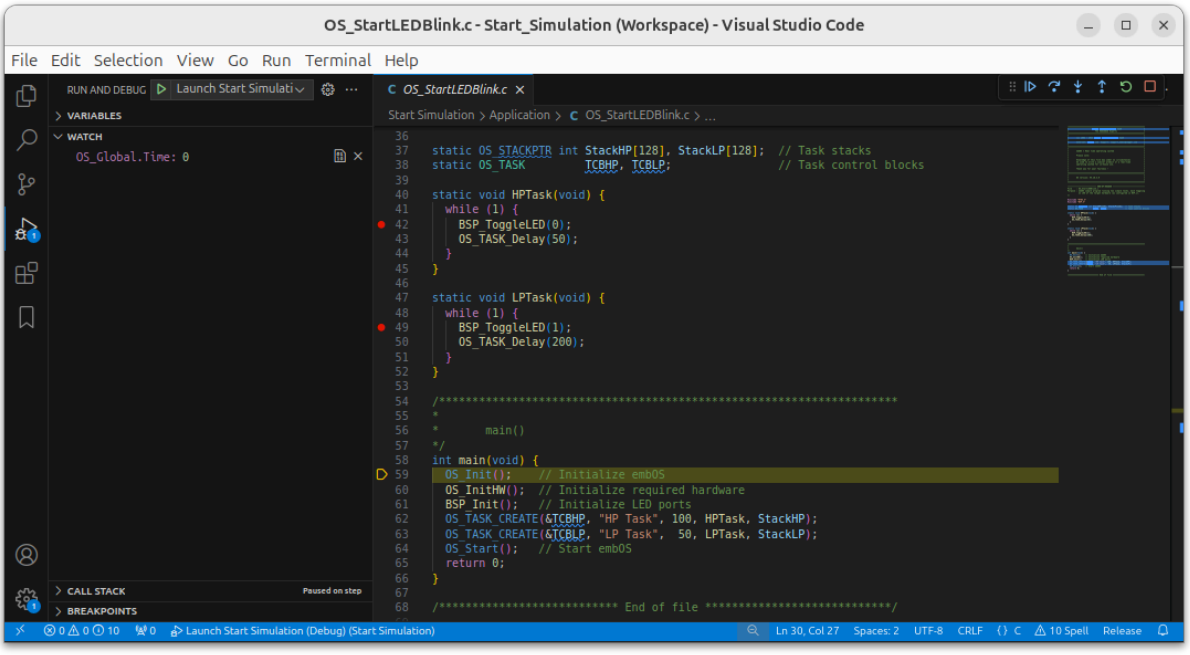
When starting the debugger, you will see the `main()` function (see example screen shot below). If the debugger does not halt at the `main()` function, set a breakpoint at the first instruction in the `main()` function.



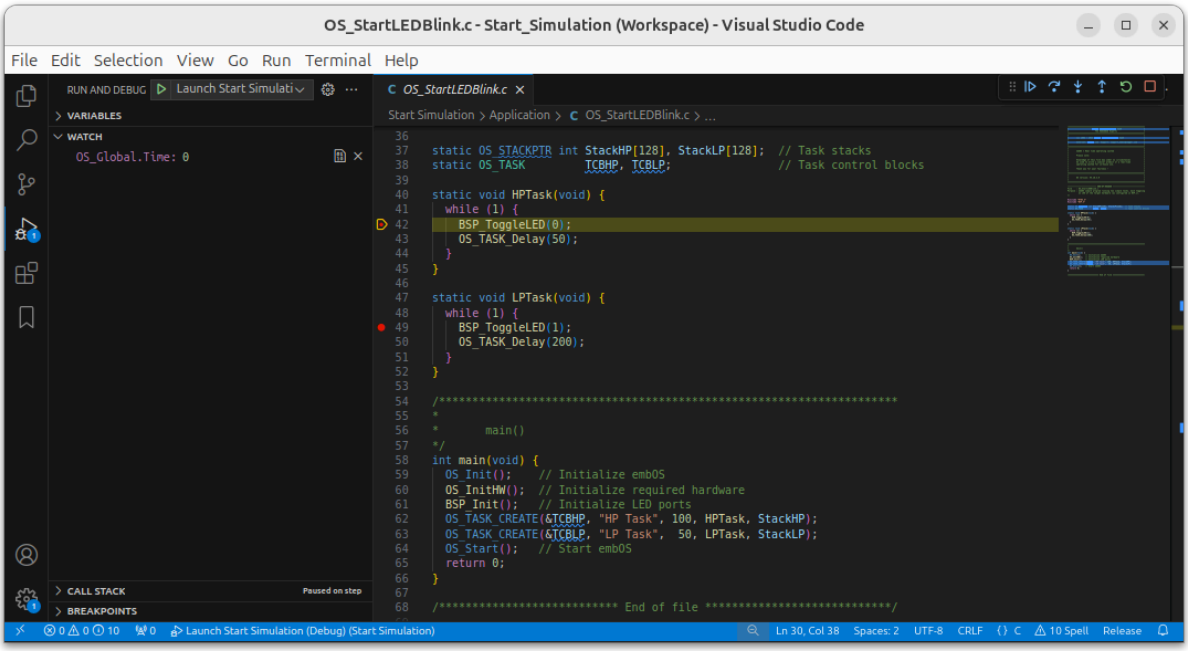
`OS_Init()` is part of the embOS library; you can therefore only step into it in disassembly mode. It initializes the relevant OS variables.

`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.

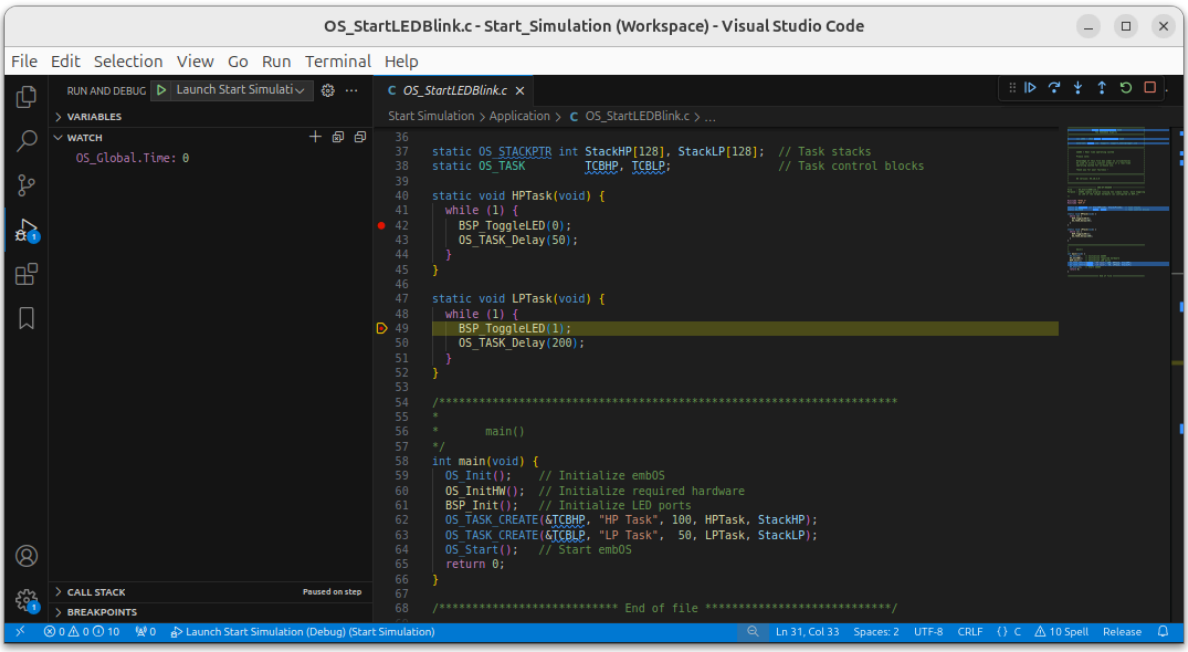
`OS_Start()` should be the last line in `main()`, because it starts multitasking and does not return, unless `OS_Stop()` is used. Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.



Step over `OS_Start()`, or step into `OS_Start()` in disassembly mode until you reach the highest priority task. If you continue stepping, the first LED of your device will be switched on, the `HPTask()` will run into its delay and therefore, embOS will start the task with the lower priority.



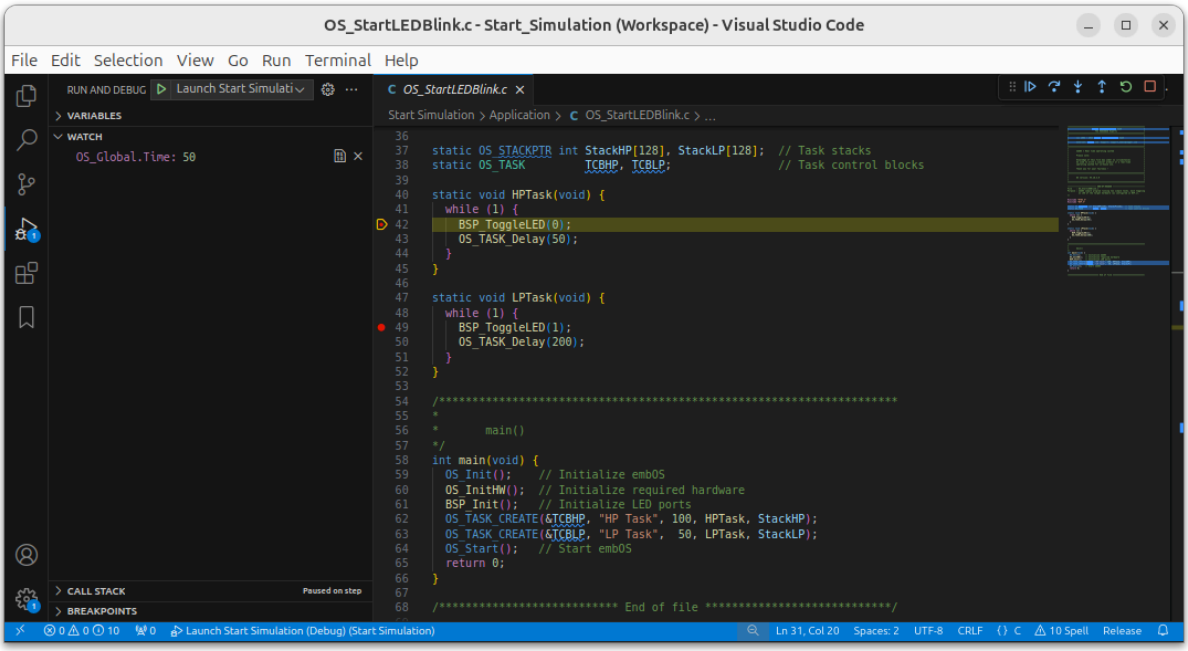
Continuing to step through the program, the `LPTask()` will switch on the other LED and then run into its delay.



As there is no other task ready for execution when `LPTask()` runs into its delay, embOS will suspend `LPTask()` and switch to the idle process, which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

The embOS simulation does not contain an `OS_Idle()` function which is implemented in normal embOS ports.

When you step over the `OS_TASK_Delay()` function of the `LPTask()`, you will arrive back in the `HPTask()`. 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.



Please note, that delays seem to be longer than expected. When the debugger stops at a breakpoint, it takes some time until the screen is updated and the `OS_Global.Time` variable is examined. Therefore `OS_Global.Time` may show larger values than expected.

You may now disable the two breakpoints in our tasks and continue the execution of the application to see how the simulated device runs in real time.

Chapter 2

Build your own application

2.1 Introduction

This chapter provides all information to set up your own embOS simulation project. To build your own application, you should always start with the supplied sample project. Therefore, select the embOS sample project 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 the embOS simulation, the following files from your embOS distribution are required and have to be included in your project:

File	Usage
`Start\Lib\`	
`lib*.a`	One of the embOS libraries.
`Start\Inc\`	
`RTOS.h`	Declares all embOS API functions and data types and has to be included in any source file using embOS functions.
`Start\BoardSupport\Simulation\Setup\`	
`RTOSInit.c`	Contains initialization code for the embOS timer interrupt handling and simulation.
`OS_Error.c`	Contains the `OS_Error()` function that is called when an application error occurs.

2.3 Select a start project configuration

The embOS simulation comes with a start project which includes the following configurations:

Configuration	Description
Debug, 32-bit, Signal Implementation	32-Bit signal implementation configuration that can be used for development and debugging.
Release, 32-bit, Signal Implementation	32-Bit signal implementation configuration used to build a release executable. It may be used for demonstration purposes.
Debug, 32-bit, Real-time Thread Implementation	32-Bit real-time thread implementation configuration that can be used for development and debugging.
Release, 32-bit, Real-time Thread Implementation	32-Bit real-time thread implementation configuration used to build a release executable. It may be used for demonstration purposes.

2.4 Add your own code

For your own code, you may add a new folder to the project or add additional files to the `Application` folder. You may modify or replace the sample application source file in the `Application` directory.

The `main()` function has to be used as an entry point of the embOS simulation. Your `main()` function has to initialize embOS by calling `OS_Init()` and `OS_InitHW()` prior to any other embOS functions that are called.

2.5 Rebuilding the embOS libraries

New libraries for the embOS simulation can only be built using the source version of the embOS simulation.

- Modify the `Prep.sh` bash script in the root directory of the embOS simulation source distribution to set the path to the compiler toolchain.
- Finally start `M.sh` to produce a new `Start\Lib\` folder which then contains the new libraries.

Chapter 3

embOS simulation implementations

3.1 Introduction

The embOS simulation for Linux provides two implementations, a signal and a real-time thread implementation. The signal implementation uses signals and semaphores to suspend and resume the execution of threads. The real-time thread implementation uses threads with the SCHED_FIFO scheduling policy, which enables strict scheduling of threads according to the priorities assigned to them. We recommend to use the real-time thread implementation, because of its better performance. But to be able to use this implementation the simulation requires a dedicated core that will be fully occupied by the embOS simulation and the user executing the simulation must provide a specific hard real-time priority limit. This chapter explains the characteristics of both implementations and how to configure the Linux for the real-time thread implementation.

3.1.1 Signal implementation

With the signal implementation the embOS simulation runs on all available cores. The embOS scheduler and each task and ISR have their own POSIX threads that are using the SCHED_OTHER scheduling policy which is the default scheduling policy. All threads using the SCHED_OTHER scheduling policy have a fixed static priority of 0, which is the lowest priority, and thus will always be preempted by real-time threads. For the embOS simulation to be able to control which thread is running signals are used which force threads to call blocking Linux functions until they receive another signal. However, whether a thread is executed still depends on the scheduling algorithm of the Linux scheduler. Due to the many signals that are sent and the synchronization of threads via semaphores many thread context switches are caused. Between these thread context switches, the Linux scheduler is free to schedule threads of other processes which delay the execution of the embOS simulation.

The signal implementation can be run as a normal user without the need for specific resource limits.

> **Note**
>
> When a thread receives a signal while it is in a waiting state due to a blocking non-embOS function call Linux will schedule it as soon as possible so that it can handle the signal. In this case the thread might return from the blocking function call with an error. Therefore, the return value and eventually errno always needs to be checked and if necessary the Linux API call must be repeated.

3.1.2 Real-time thread implementation

The real-time thread implementation makes use of the SCHED_FIFO scheduling policy which turns threads into real-time threads. Real-time threads have a static priority greater than 0 and preempt all threads with lower priority like all normal threads with a static priority of 0 due to the SCHED_OTHER scheduling policy. The priority of real-time threads can be assigned and changed at run time as desired. This allows the embOS simulation to control which thread is currently running. After changing the priority of a thread, the Linux scheduler immediately executes the thread with the highest priority that is ready to run. All this results in faster and more efficient task switches than with the signal implementation.

To prevent real-time threads from running simultaneously on multiple cores, the CPU affinity mask of the process is changed so that all of its threads run on the same core. Due to how the embOS simulation is implemented there is always one embOS simulation thread that is running. This means that the simulation will have a load of nearly 100% on the core it is running on, preventing other processes from being executed on it. Merely other real-time threads with equal or higher priority are executed on that core. To avoid the simulation from making the Linux and most applications unresponsive a CPU with at least 2 cores is required.

3.1.2.1 Real-time priorities and the hard limit

In most cases real-time priorities from 1 to 99 can be used with 1 being the lowest real-time priority. The embOS simulation requires the priorities 1 to 8 and thus can be preempted by all other real-time priorities. For Linux users other than the root to be able to run applications that use real-time threads up to the priority 8, the hard real-time priority limit for that user must be set to at least 8. You can check the currently set hard real-time priority limit for the current user with the following command:

```
ulimit -Hr
```

If the displayed value is 8 or higher, the user is already able to run embOS simulation applications. Is the value lower than 8, then the hard limit for this user needs to be configured. This can be accomplished by adding an entry for the user in the `/etc/security/limits.conf` file. Replace the `<domain>` in the following entry with the user you want to increase the hard real-time priority limit for and add the line to the `limits.conf` file.

```
<domain> hard rtprio 8
```

For more information on how to specify resource limits using the `limits.conf` file please refer to LIMITS.CONF(5) in the Linux man pages.

Chapter 4

Libraries

4.1 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<Implementation>32<LibMode>.a`

Parameter	Meaning	Values
`Implementation`	Specifies the implementation	: Signals rt : Real-time threads
`Architecture`	Specifies the architecture	32 : 32-bit embOS (x86)
`LibMode`	Specifies the library mode	xr : Extreme Release r : Release s : Stack check sp : Stack check + profiling d : Debug dp : Debug + profiling + Stack check dt : Debug + profiling + Stack check + trace

Example

`libos32dp.lib` is the library for the 32-bit embOS simulation using the signal implementation with debug and profiling support.

`libosrt32r.lib` is the library for the 32-bit embOS simulation using the real-time thread implementation and the embOS release build.

Chapter 5

CPU and compiler specifics

5.1 Interrupt and thread safety

Using embOS with specific calls to standard library functions (e.g. heap management functions) requires thread-safe system libraries if these functions are called from several tasks or interrupts. The thread safety provided by GNU C library is not enough to ensure thread safety for the embOS simulation. Therefore, system library functions that need to be thread-safe and that are called by the simulation need to be overwritten and made thread-safe.

The Setup directory in the embOS BSP contains the file `OS_ThreadSafe.c` which overwrites heap management and some other functions. This is done by providing own definitions for the system library functions used by the simulation. These definitions then ensure thread safety before calling the actual library function whose address was previously loaded from the shared library. System library functions that are not yet overwritten in `OS_ThreadSafe.c` must be made thread-safe in the same way as it is done for the other functions in `OS_ThreadSafe.c`.

Overwriting functions by providing custom definitions works only as long as the identifiers of the system library functions do not change. Should the identifier of a function change, e.g. due to optimization, then the function with the changed identifier is called instead of the thread-safe custom function implementation. The include header files of the GNU C library use the `_FORTIFY_SOURCE` macro to control code hardening, which also results in function inlining. Inlining the calls to system library functions can result in the call of different function identifiers as expected which again results in the thread-safe custom function implementations `OS_ThreadSafe.c` being omitted. Therefore, `_FORTIFY_SOURCE` must be defined to 0 when the compiler is called.

```
gcc -U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=0 ...
```

With GNU C library versions prior to 2.34 it is not possible to overwrite heap management functions and retrieve their address from the shared library. In this case malloc hooks are used which were removed with version 2.34. `OS_ThreadSafe.c` automatically checks the GNU C library version and uses the appropriate implementation.

Chapter 6

Stacks

6.1 Task stacks

Every embOS task has to have its own stack. Task stacks can be located in any RAM memory location. In embOS simulation, every task runs as a separate thread. The real “task” stack is managed by Linux. Declaration and size of task stacks in your application are necessary for generic embOS functions, but do not affect the stack size of the generated Linux thread. A stack check and stack overflows are not simulated.

6.2 System stack

The system stack used during startup is managed by Linux. A stack check and stack overflows are not simulated.

6.3 Interrupt stack

Simulated interrupts in the embOS simulation run as Linux thread. ISR stacks are managed by Linux. Since every ISR has its own stack, embOS cannot simulate interrupt stack check and stack overflows.

Chapter 7

Interrupts

7.1 Introduction

With the embOS simulation, interrupts have to be simulated and thus differ from those used in your embedded application. The following chapter describes interrupt simulation and handling in the embOS simulation.

7.2 How interrupt simulation works

With the embOS simulation, all interrupt handler functions are started as individual threads. Because embOS might have to disable interrupts when embOS internal operations are performed, the embOS simulation has to be able to suspend and resume interrupt handler threads. This requires, that all interrupt handler threads have to be created and installed by the special embOS simulation API function `OS_SIM_CreateISRThread()`.

Interrupt simulation under embOS simulation works as follows:

- An interrupt handler is written as a normal "C"-function without parameters or return value.
- The interrupt handler function is initialized and started as a thread by using the embOS API function `OS_SIM_CreateISRThread()`.
- The interrupt handler function should contain an endless loop which calls a blocking function (e.g. usleep()) that returns when the ISR has to be executed. Blocking functions may return unexpectedly when the thread receives a signal. Thus, the return value and errno should be checked and the blocking call eventually be repeated before executing the ISR by mistake.
- Interrupts can be deleted by letting the ISRs return from the ISR or by calling `OS_SIM_DeleteISRThread()` which will send a cancellation request that lets the ISR thread terminate asynchronously.

7.3 Defining interrupt handlers for simulation

Interrupt handlers used in the embOS simulation can not handle the real hardware normally used in your target application. The interrupt handler functions of your real target application have to be replaced by a modified version which can be used in the simulation.

Simple ISR example:

```
static void _ISRTimerThread(void) {
  // Perform one-time initialization here
  while (1) {
    usleep(1000);       // Suspend until delay expires
    OS_INT_Enter();     // Tell embOS that interrupt code is running
    DoTimerHandling();  // Any functionality can be added here
    OS_INT_Leave();     // Tell embOS that interrupt code ends
  }
}
```

7.4 Interrupt priorities

All interrupts have the same priority and nesting of interrupts is not supported. With the real-time thread implementation, the real-time priority is above all other threads. If two ISR threads are ready for execution it is up to the Linux scheduler which one is being executed. With the non-real-time thread implementation, ISR threads have the same priority as all other threads and the scheduling of them is up to the Linux scheduler. In between calls to `OS_INT_Enter*()` and `OS_INT_Leave()`, no other ISR has the chance to run, as interrupts are disabled. However, the code outside these calls are not seen as part of the interrupt handler and may be preempted by other threads.

7.5 API functions

Routine	Description	main	Task	ISR	Timer
OS_SIM_CreateISRThread()	Installs an ISR handler.	●	●	●	●
OS_SIM_CreateISRThreadEx()	Installs an ISR handler and sets a name.	●	●	●	●
OS_SIM_DeleteISRThread()	Uninstalls an ISR handler.	●	●		

7.5.1 OS_SIM_CreateISRThread()

Description

`OS_SIM_CreateISRThread()` installs an embOS simulation ISR handler.

Prototype

```
void* OS_SIM_CreateISRThread(OS_ISR_HANDLER* pfStartAddress);
```

```
typedef void OS_ISR_HANDLER(void);
```

Parameters

Parameter	Description
`pStartAddress`	Pointer to void function which serves as simulated interrupt handler.

Return Value

A handle to the created interrupt simulation thread which can be used with `OS_SIM_DeleteISRThread()`.

7.5.2 OS_SIM_CreateISRThreadEx()

Description

`OS_SIM_CreateISRThreadEx()` installs an embOS simulation ISR handler.

Prototype

```
void* OS_SIM_CreateISRThreadEx(      OS_ISR_HANDLER* pfStartAddress,
                               const char*           sThreadName);
```

```
typedef void OS_ISR_HANDLER(void);
```

Parameters

Parameter	Description
`pStartAddress`	Pointer to void function which serves as simulated interrupt handler.
`sThreadName`	Interrupt handler name.

Return Value

A handle to the created interrupt simulation thread which can be used with `OS_SIM_DeleteISRThread()`.

7.5.3 OS_SIM_DeleteISRThread()

Description

`OS_SIM_DeleteISRThread()` uninstalls an embOS simulation ISR handler.

Prototype

```
void OS_SIM_DeleteISRThread(void* pThreadHandle);
```

Parameters

Parameter	Description
`ThreadHandle`	Handle to the ISR which was returned by `OS_SIM_CreateISRThread()`.

Additional Information

The ISR thread is detached and will be terminated asynchronously. This can result in undefined behavior, why terminating an ISR by letting it return from the ISR function is recommended.

Example

The following example shows an ISR handler and how it can be uninstalled by adding a new function `DeInitSystemTick()`. `DeInitSystemTick()` can then be called after `OS_Stop()` and `OS_DeInit()`.

```
static void* _ISRThreadHandle;

/*********************************************************************
*
*       _ISRThread()
*/
static void _ISRThread(void) {
  //
  // ... initialization
  //
  while (1) {
    usleep(1000);
    OS_INT_Enter();
    //
    // ...
    //
    OS_INT_Leave();
  }
}

/*********************************************************************
*
*       _InitMyISR()
*/
static void _InitMyISR(void) {
  _ISRThreadHandle = OS_SIM_CreateISRThread(_ISRTickThread);
}

/*********************************************************************
*
*       DeInitSystemTick()
*/
static void DeInitMyISR(void) {
  OS_SIM_DeleteISRThread(_ISRThreadHandle);
}
```


Chapter 8

MPU support

8.1 embOS-MPU Cortex-M support

embOS-MPU Sim Linux can compile the code of applications that are written for embOS-MPU and Cortex-M architectures. It provides the same port-specific embOS-MPU API functions, structures and macros so that no modifications to the code is required. The simulation also performs all non-hardware related debug checks and assertions to locate incorrect use of the embOS-MPU API. However, the behavior of the Cortex-M MPU hardware is not simulated.

Chapter 9

Calling blocking non-embOS functions from tasks

9.1 Introduction

The embOS simulation is typically used to simulate real embedded applications. This may require the usage of potentially blocking non-embOS functions from tasks. Calling blocking embOS functions will suspend the task for the time it is waiting and allows tasks with lower priority to be scheduled by embOS. Calling blocking non-embOS functions will freeze the calling task and no other task with lower priority will be scheduled. This may cause the whole simulation to stop until the blocking task continues execution. To avoid this, two embOS API functions are available to manage the call of blocking non-embOS functions.

Similar to handling critical regions, there is one entry function (`OS_SIM_EnterSysCall()`), which has to be called before the blocking non-embOS function, and one exit function (`OS_SIM_LeaveSysCall()`), which has to be called after the blocking non-embOS function.

The Application folder of the embOS shipment contains the sample application `OS_Sim-Blocked.c`, which demonstrates these functions' usage on blocking non-embOS function calls.

9.2 API functions

Routine	Description	main	Task	ISR	Timer
`OS_SIM_EnterSysCall()`	Must be called prior to calling any blocking non-embOS function from a task.		•		
`OS_SIM_LeaveSysCall()`	Must be called after calling any blocking non-embOS API function from a task, and before any other embOS API function is called.		•		

9.2.1 OS_SIM_EnterSysCall()

Description

`OS_SIM_EnterSysCall()` has to be called before a blocking non-embOS function is called from a task.

Prototype

```
void OS_SIM_EnterSysCall(void);
```

Additional information

After calling `OS_SIM_EnterSysCall()`, no further embOS API function except `OS_SIM_LeaveSysCall()` must be called.

Example

```
...
OS_SIM_EnterSysCall();
// Any blocking non-embOS function may be called now.
...
recv (socket, pBuf, len, flags);
// Any other code may follow.
// No embOS function must be called except OS_SIM_LeaveSysCall().
...
OS_SIM_LeaveSysCall();
// From now on, calling other embOS functions is allowed.
...
```


9.2.2 OS_SIM_LeaveSysCall()

Description

`OS_SIM_LeaveSysCall()` has to be called after execution of a blocking non-embOS function, before any other embOS function is called.

Prototype

```
void OS_SIM_LeaveSysCall(void);
```

Additional information

It must be called only when `OS_SIM_LeaveSysCall()` has been called by the same task before.

Example

```
...
OS_SIM_EnterSysCall();
// Any blocking non-embOS function may be called now.
...
recv (socket, pBuf, len, flags);
// Any other code may follow.
// No embOS function must be called except OS_SIM_LeaveSysCall().
...
OS_SIM_LeaveSysCall();
// From now on, calling other embOS functions is allowed.
...
```


Chapter 10

RTT and SystemView

10.1 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.

10.2 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 11

Technical data

11.1 Resource Usage

The memory requirements of embOS for RAM 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	RAM	156 bytes
Task control block	RAM	332 bytes
Software timer	RAM	20 bytes
Task event	RAM	0 bytes
Event object	RAM	16 bytes
Mutex	RAM	16 bytes
Semaphore	RAM	8 bytes
RWLock	RAM	28 bytes
Mailbox	RAM	24 bytes
Queue	RAM	32 bytes
Watchdog	RAM	12 bytes
Fixed Block Size Memory Pool	RAM	32 bytes

