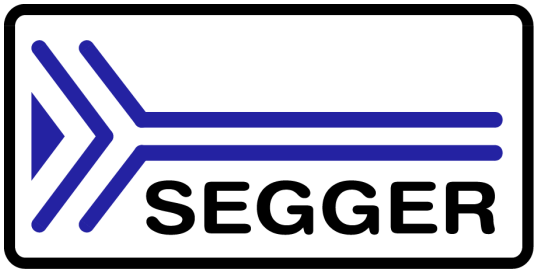
embOS

Real-Time
Operating System

CPU & Compiler
specifics for Cortex M
using
Sourcery CodeBench

Document: UM01027
Software version 4.16
Revision: 0
Date: March 15, 2016



A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Disclaimer

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

Copyright notice

Trademarks

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

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

Contact address

SEGGER Microcontroller GmbH & Co. KG

In den Weiden 11
D-40721 Hilden

Germany

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

Manual versions

This manual describes the current software version. If any error occurs, inform us and we will try to assist you as soon as possible.
Contact us for further information on topics or routines not yet specified.

Print date: March 15, 2016

Software	Revision	Date	By	Description
4.16	0	160315	MC	Update to latest embOS generic sources.
4.06b	0	150410	SC	Update to latest embOS generic sources.
4.04a	0	150127	SC	Update to latest embOS generic sources.
3.88f	1	140312	SC	Chapter numbers corrected.
3.88f	0	130923	TS	Update to latest embOS generic sources.
3.88c	0	130816	TS	Update to latest embOS generic sources.
3.88	0	130225	TS	Update to latest embOS generic sources.
3.82u	0	110706	TS	Update to latest embOS generic sources.
3.82g	1	100701	TS	Chapter Stacks: Task stack size corrected.
3.82g	0	100511	TS	First version.

About this document

Assumptions

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

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

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

How to use this manual

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

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command-prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in programm 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 1.1: Typographic conventions

SEGGER Microcontroller GmbH & Co. KG develops and distributes software development tools and ANSI C software components (middleware) for embedded systems in several industries such as telecom, medical technology, consumer electronics, automotive industry and industrial automation.

SEGGER's intention is to cut software development time for embedded applications by offering compact flexible and easy to use middleware, allowing developers to concentrate on their application.

Our most popular products are emWin, a universal graphic software package for embedded applications, and embOS, a small yet efficient real-time kernel. emWin, written entirely in ANSI C, can easily be used on any CPU and most any display. It is complemented by the available PC tools: Bitmap Converter, Font Converter, Simulator and Viewer. embOS supports most 8/16/32-bit CPUs. Its small memory footprint makes it suitable for single-chip applications.

Apart from its main focus on software tools, SEGGER develops and produces programming tools for flash micro controllers, as well as J-Link, a JTAG emulator to assist in development, debugging and production, which has rapidly become the industry standard for debug access to ARM cores.

Corporate Office:
http://www.segger.com

United States Office:
http://www.segger-us.com

EMBEDDED SOFTWARE (Middleware)

emWin
Graphics software and GUI
emWin is designed to provide an efficient, processor- and display controller-independent graphical user interface (GUI) for any application that operates with a graphical display.

embOS
Real Time Operating System
embOS is an RTOS designed to offer the benefits of a complete multitasking system for hard real time applications with minimal resources.

embOS/IP
TCP/IP stack
embOS/IP a high-performance TCP/IP stack that has been optimized for speed, versatility and a small memory footprint.

emFile
File system
emFile is an embedded file system with FAT12, FAT16 and FAT32 support. Various Device drivers, e.g. for NAND and NOR flashes, SD/MMC and CompactFlash cards, are available.

USB-Stack
USB device/host stack
A USB stack designed to work on any embedded system with a USB controller. Bulk communication and most standard device classes are supported.

SEGGER TOOLS

Flasher
Flash programmer
Flash Programming tool primarily for micro controllers.

J-Link
JTAG emulator for ARM cores
USB driven JTAG interface for ARM cores.

J-Trace
JTAG emulator with trace
USB driven JTAG interface for ARM cores with Trace memory. supporting the ARM ETM (Embedded Trace Macrocell).

J-Link / J-Trace Related Software
Add-on software to be used with SEGGER's industry standard JTAG emulator, this includes flash programming software and flash breakpoints.

Table of Contents

Chapter 1

Using embOS

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

1.1 Installation

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

To install it, proceed as follows:

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

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

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

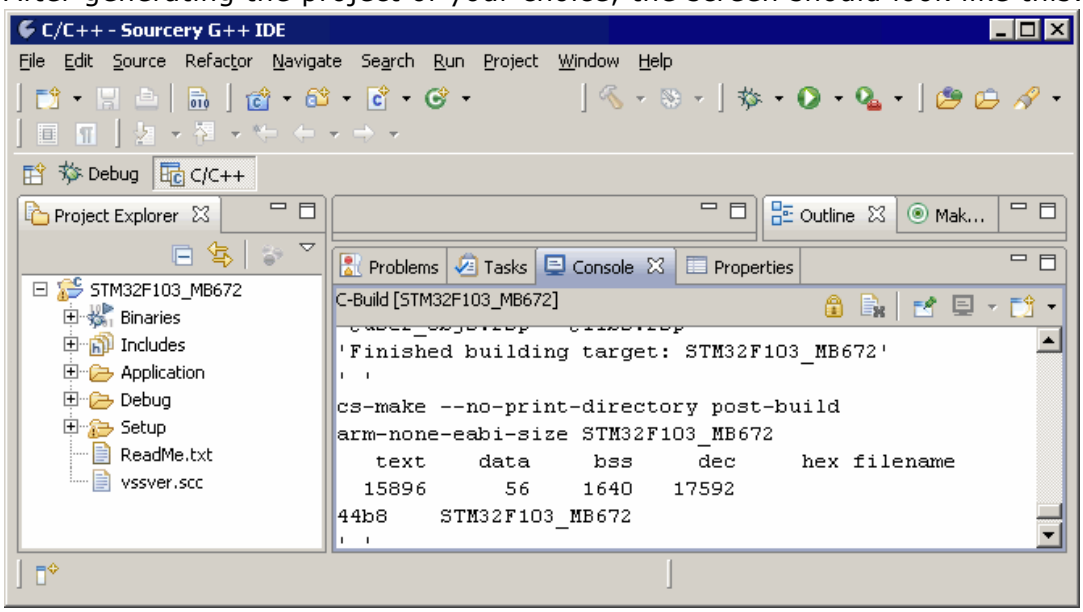
After installation of embOS you can create your first multitasking application. You have received several ready to go sample start workspaces and projects and every other files needed in the subfolder **Start**. It is a good idea to use one of them as a starting point for all of your applications. The subfolder **BoardSupport** contains the workspaces and projects which are located in manufacturer- and CPU-specific sub-folders.

To start with, you may use any project from **BoardSupport** subfolder:

To get your new application running, you should proceed as follows:

- Create a work directory for your application, for example `c:\work`.
- Copy the whole folder **Start** which is part of your embOS distribution into your work directory.
- Clear the read-only attribute of all files in the new **Start** folder.
- Open one sample workspace/project in **Start\BoardSupport\<DeviceManufactor>\<CPU>** with your IDE (for example, by double clicking it).
- Build the project. It should be built without any error or warning messages.

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



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:

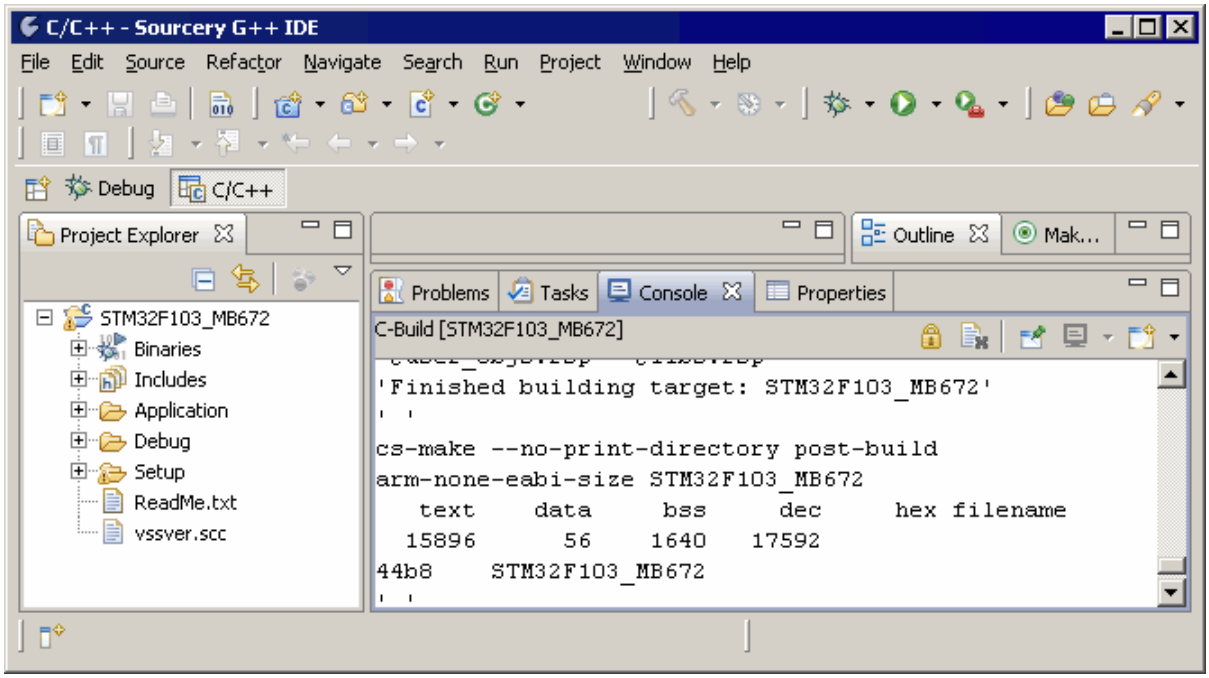
After initialization of embOS; two tasks are created and started.
The two tasks are activated and execute until they run into the delay, then suspend for the specified time and continue execution.

```
/*********************************************************************
*                SEGGER Microcontroller GmbH & Co. KG                *
*                        The Embedded Experts                        *
**********************************************************************
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).
--------- END-OF-HEADER ------------------------------------------------
*/

#include "RTOS.h"
#include "BSP.h"

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

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

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

/*********************************************************************
*
*       main()
*/
int main(void) {
  OS_IncDI();                      /* Initially disable interrupts  */
  OS_InitKern();                   /* Initialize OS                 */
  OS_InitHW();                     /* Initialize Hardware for OS    */
  BSP_Init();                      /* Initialize LED ports          */
  /* You need to create at least one task before calling OS_Start() */
  OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
  OS_Start();                      /* Start multitasking            */
  return 0;
}

/****** End Of File *************************************************/
```

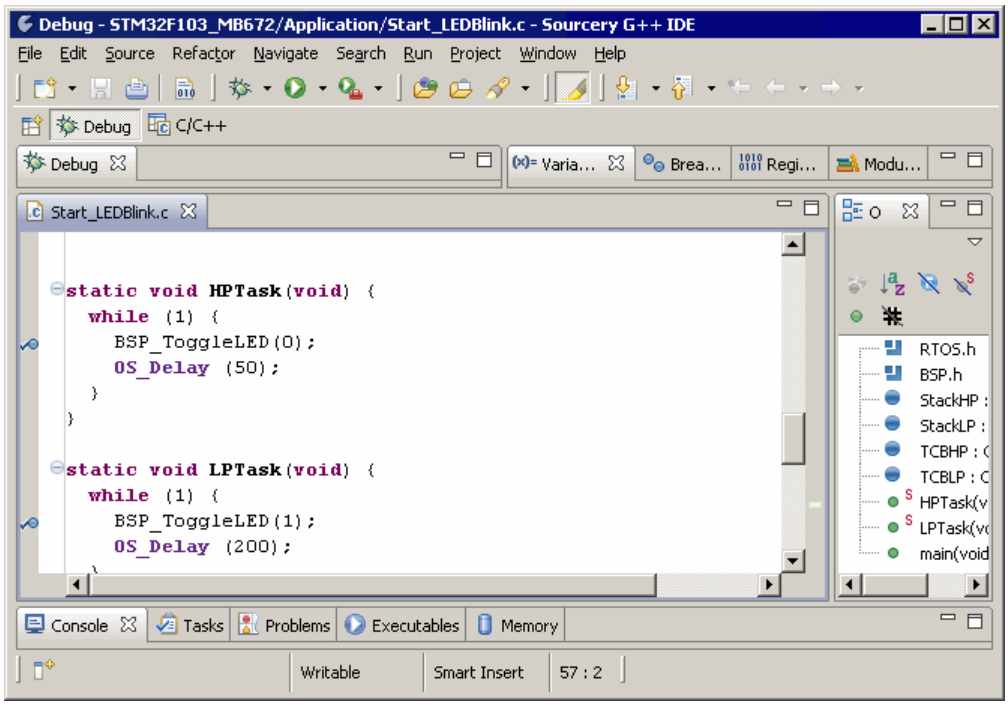

1.4 Stepping through the sample application

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_IncDI()` initially disables interrupts.
`OS_InitKern()` 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. Because of the previous call of `OS_IncDI()`, interrupts are not enabled during execution of `OS_InitKern()`.
`OS_InitHW()` is part of `RTOSInit_*.c` and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the system tick interrupt for embOS. Step through it to see what is done.

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

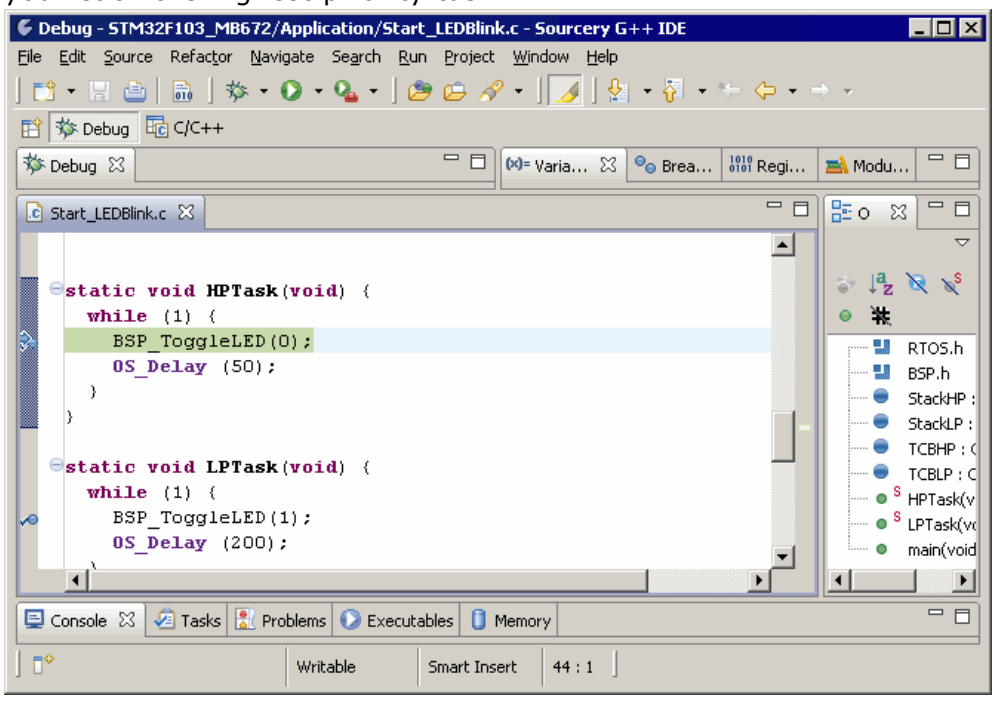


Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.

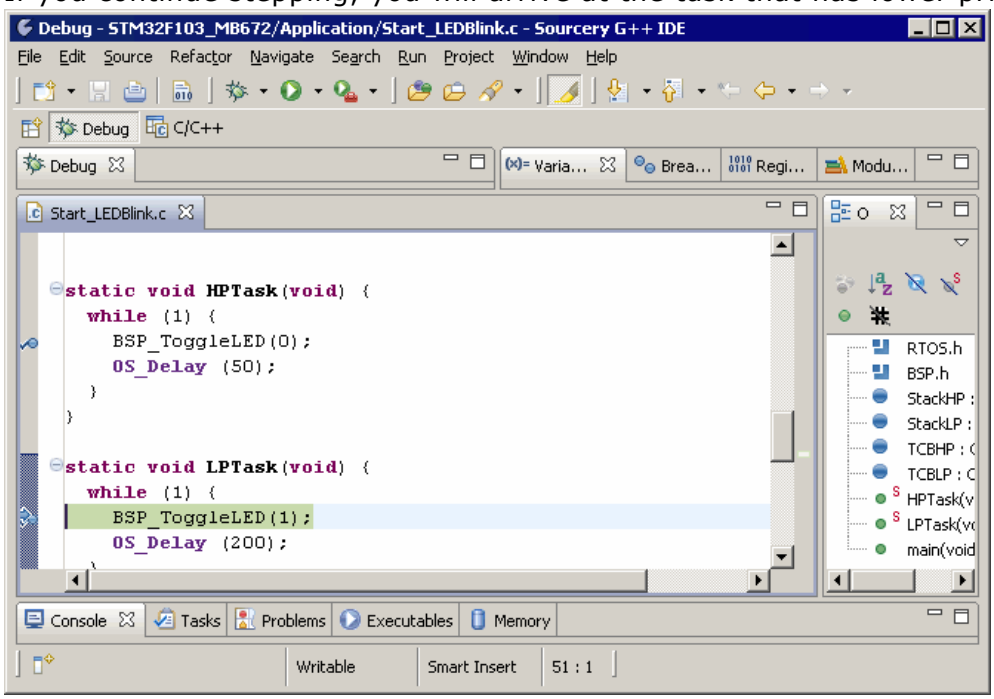


As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only. Y

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

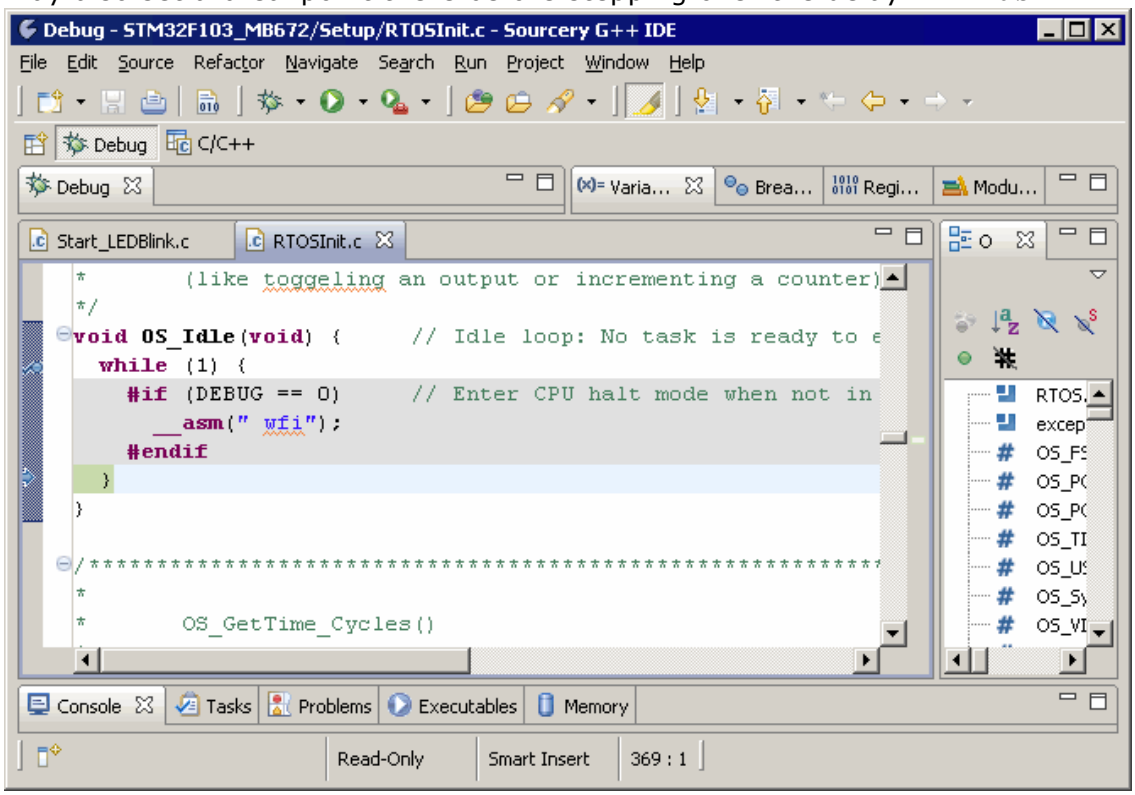


If you continue stepping, you will arrive at the task that has lower priority:



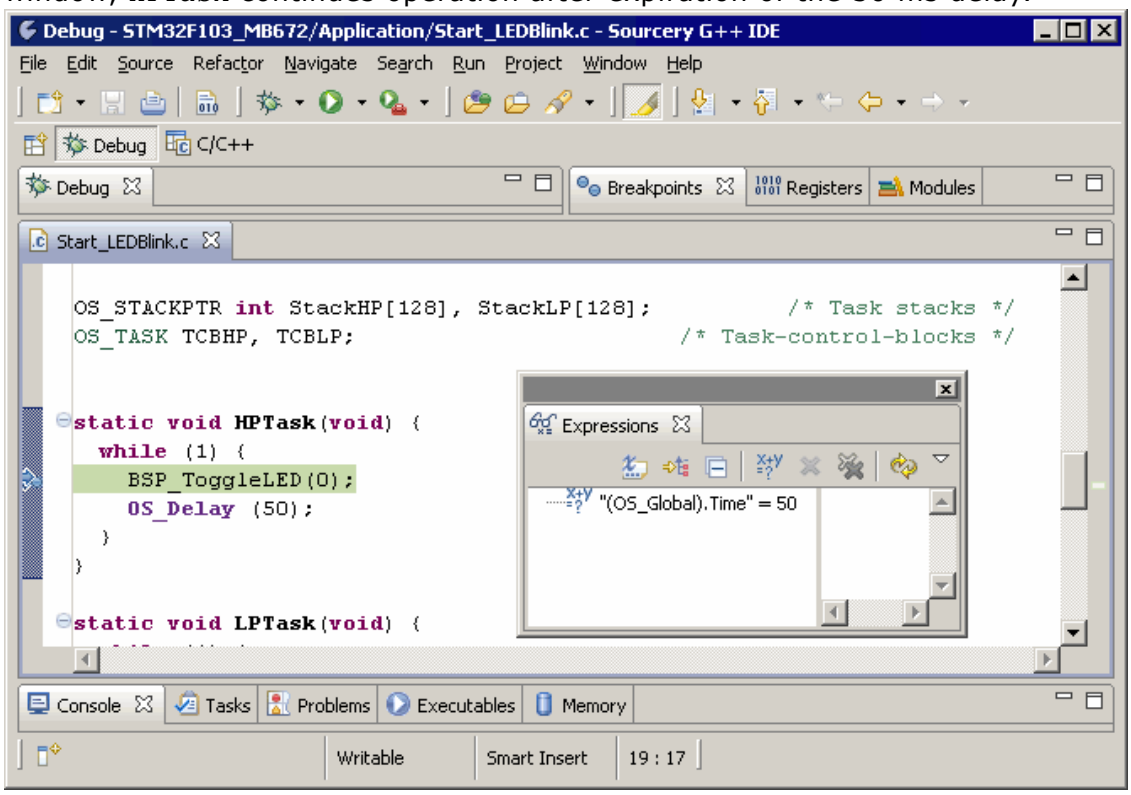
Continue to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

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



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_Time`, shown in the **Watch** window, `HPTask` continues operation after expiration of the 50 ms delay.



Chapter 2

Build your own application

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

2.1 Introduction

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

2.2 Required files for an embOS

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

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

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

2.3 Change library mode

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

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

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

2.4 Select another CPU

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

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

Chapter 3

Libraries

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

3.1 Naming conventions for prebuilt libraries

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

The libraries are named as follows:

libos<CpuMode><Arch><ByteOrder><LibMode>.a

Parameter	Meaning	Values
CpuMode	Specifies the CPU mode	T: Always thumb
Arch	CPU Architecture	7: Cortex M3/M4 7V: Cortex M4F with VFP
ByteOrder	Endianess	B: Big endian L: Little endian
LibMode	Library mode	XR: eXtreme Release R: Release S: Stack check D: Debug SP: Stack check + Profiling DP: Debug + Profiling DT: Debug + Trace

Example:

libosT7LDP.a is the library for a project using a Cortex-M3 or Cortex-M4 core without VFP , thumb mode, little endian mode with debug and profiling support.

Chapter 4

CPU and Compiler specifics

4.1 Standard system libraries

embOS for Cortex-M and GCC compiler may be used with standard GNU system libraries for most of all projects without any modification.

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, if non thread safe functions are used from different tasks.

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

4.2 Reentrancy, thread safe heap management

The heap management functions in the system libraries are not thread-safe without implementation of additional locking functions.
The GCC library calls two hook functions to lock and unlock the mutual access of the heap-management functions.
The empty locking functions from the system library may be overwritten by the application to implement a locking mechanism.

A locking is required when multiple tasks access the heap, or when objects are created dynamically on the heap by multiple tasks.
The locking functions are implemented in the source module `OS_MallocLock.c` which is included in the "Setup" subfolder in every embOS start project.
If thread safe heap management is required, the module has to be compiled and linked with the application.

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 resource semaphore.

4.2.2 __malloc_unlock()

`__malloc_unlock()` is the 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 resource semaphore.

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.

Chapter 5

Stacks

This chapter describes how embOS uses the different stacks of the Cortex M CPU.

5.1 Task stack for Cortex-M

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

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

For the Cortex M CPUs, this minimum basic task stack size is about 72 bytes. Because any function call uses some amount of stack and every exception also pushes at least 32 bytes onto the current stack, the task stack size has to be large enough to handle one exception too. We recommend at least 256 bytes stack as a start.

5.2 System stack for Cortex-M

The embOS system executes in thread mode, the scheduler executes in handler mode. The minimum system stack size required by embOS is about 136 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 256 bytes for the CSTACK.

5.3 Interrupt stack for Cortex-M

If a normal hardware exception occurs, the Cortex M core switches to handler mode which uses the main stack pointer.
With embOS, the main stack pointer is initialized to use the CSTACK which is defined in the linker command file.
The main stack is also used as stack by the embOS scheduler and during idle times, when no task is ready to run and OS_Idle() is executed.

Chapter 6

Interrupts

The Cortex M core comes with a built-in vectored interrupt controller which supports up to 32 separate interrupt sources. The real number of interrupt sources depends on the specific target CPU.

6.1 What happens when an interrupt occurs?

- The CPU-core receives an interrupt request from the interrupt controller.
- As soon as the interrupts are enabled, the interrupt is accepted and executed.
- The CPU pushes temporary registers and the return address onto the current stack.
- The CPU switches to handler mode and main stack.
- The CPU saves an exception return code and current flags onto the main stack.
- The CPU jumps to the vector address delivered by the NVIC.
- The interrupt handler is processed.
- The interrupt handler ends with a return from interrupt by reading the exception return code.
- The CPU switches back to the mode and stack which was active before the exception was called.
- The CPU restores the temporary registers and return address from the stack and continues the interrupted function.

6.2 Defining interrupt handlers in C

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

Example

Simple interrupt routine:

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

6.3 Interrupt vector table

After Reset, the ARM Cortex M CPU uses an initial interrupt vector table which is located in ROM at address 0x00. It contains the address for the main stack and addresses for all exceptions handlers.

The interrupt vector table is located in a C source or assembly file in the CPU specific subfolder. All interrupt handler function addresses have to be inserted in the vector table, as long as a RAM vector table is not used.

The vector table may be copied to RAM to enable variable interrupt handler installation. The compile time switch OS_USE_VARINTTABLE is used to enable usage of a vector table in RAM.

To save RAM, the switch is set to zero per default in RTOSInit_*.c. It may be overwritten by project settings to enable the vector table in RAM. The first call of OS_InstallISRHandler() will then automatically copy the vector table into RAM. When using your own interrupt vector table, ensure that the addresses of the embOS exception handlers OS_Exception() and OS_Systick() are included.
When the vector table is not located at address 0x00, the vector base register in the NVIC controller has to be initialized to point to the vector table base address.

6.4 Interrupt-stack switching

Since Cortex M core based controllers have two separate stack pointers, and embOS runs the user application on the process stack, there is no need for explicit stack-switching in an interrupt routine which runs on the main stack. The routines OS_EnterIntStack() and OS_LeaveIntStack() are supplied for source code compatibility to other processors only and have no functionality.

6.5 Zero latency interrupts

6.5.1 Zero latency interrupts with Cortex-M

Instead of disabling interrupts when embOS does atomic operations, the interrupt level of the CPU is set to 128. Therefore all interrupt priorities higher than 128 can still be processed. Please note that lower priority numbers define a higher priority. All interrupts with priority level from 0 to 127 are never disabled. These interrupts are named zero latency interrupts. You must not execute any embOS function from within a zero latency interrupt function.

6.6 Interrupt priorities

This chapter describes interrupt priorities supported by the CortexM CPU cores.
The priority is any number between 0 and 255 as seen by the CPU core. With embOS and its own setup functions for the interrupt controller and priorities, there is no difference in the priority values regardless of the different preemption level of specific devices.
Using the CMSIS functions to set up interrupt priorities requires different values for the priorities. These values depend on the number of preemption levels of the specific chip. a description is found in the chapter CMSIS.

6.6.1 Interrupt priorities with Cortex-M cores

The Cortex M3 / M4 and M4F support up to 256 levels of programmable priority with a maximum of 128 levels of preemption. Most Cortex M chips have fewer supported levels, for example 8, 16, 32, and so on. The chip designer can customize the chip to obtain the levels required. There is a minimum of 8 preemption levels. Every interrupt with a higher preemption level may preempt any other interrupt handler running on a lower preemption level. Interrupts with equal preemption level may not preempt each other.

With introduction of Zero latency interrupts, interrupt priorities usable for interrupts using embOS API functions are limited.

- Any interrupt handler using embOS API functions has to run with interrupt priorities from 128 to 255. These embOS interrupt handlers have to start with OS_EnterInterrupt() or OS_EnterNestableInterrupt() and have to end with OS_LeaveInterrupt() or OS_LeaveNestableInterrupt().
- Any zero latency interrupt (running at priorities from 0 to 127) must not call any embOS API function. Even OS_EnterInterrupt() and OS_LeaveInterrupt() must not be called.

Interrupt handlers running at low priorities (from 128 to 255) not calling any embOS API function are allowed, but must not reenable interrupts! The priority limit between embOS interrupts and zero latency interrupts is fixed at compile time to 128 and can only be changed by recompiling the embOS libraries! This is done for efficiency reasons. Basically the define OS_IPL_DI_DEFAULT in RTOS.h and the RTOS.s file must be modified. There might be other modifications necessary. Please contact the embOS support if you like to change this threshold.

6.6.2 Priority of the embOS scheduler

The embOS scheduler runs on the lowest interrupt priority. The scheduler may be preempted by any other interrupt with higher preemption priority level. The application interrupts shall run on higher preemption levels to ensure short reaction time.

During initialization, the priority of the embOS scheduler is set to 0x03 for Cortex M0 and to 0xFF for Cortex M3 / M4 and M4F, which is the lowest preemption priority regardless of the number of preemption levels.

6.6.3 Priority of the embOS system timer

The embOS system timer runs on the second lowest preemption level. Thus, the embOS timer may preempt the scheduler. Application interrupts which require fast reaction should run on a higher preemption priority level.

6.6.4 Priority of embOS software timers

The embOS software timer callback functions are called from the scheduler and run on the scheduler's preemption priority level which is the lowest interrupt priority level. To ensure short reaction time of other interrupts, other interrupts should run on a higher preemption priority level and the software timer callback functions should be as short as possible.

6.6.5 Priority of application interrupts for Cortex M3 / M4 / M7 core

Application interrupts using embOS functions may run on any priority level between 255 to 128. However, interrupts which require fast reaction should run on higher priority levels than the embOS scheduler and the embOS system timer to allow preemption of theses interrupt handlers. The interrupt handlers which require the fastest reaction may run on higher priorities than 128, but must not call any embOS function (->zero latency interrupts). We recommend that application interrupts should run on a higher preemption level than the embOS scheduler, at least at the second lowest preemption priority level.

As the number of preemption levels is chip specific, the second lowest preemption priority varies depending on the chip. If the number of preemption levels is not documented, the second lowest preemption priority can be set as follows, using embOS functions:

```
unsigned char Priority;
OS_ARM_ISRSetPrio(_ISR_ID, 0xFF);              // Set to lowest level, ALL BITS set
Priority  = OS_ARM_ISRSetPrio(_ID_TICK, 0xFF); // Read priority back
Priority -= 1;                                 // Lower preemption level
OS_ARM_ISRSetPrio(_ISR_ID, Priority);
```

6.7 Interrupt nesting

The Cortex M CPU uses a priority controlled interrupt scheduling which allows nesting of interrupts per default. Any interrupt or exception with a higher preemption priority may interrupt an interrupt handler running on a lower preemption priority. An interrupt handler calling embOS functions has to start with an embOS prolog function: it informs embOS that an interrupt handler is running. For any interrupt handler, the user may decide individually whether this interrupt handler may be preempted or not by choosing the prolog function.

6.7.1 OS_EnterInterrupt()

Description

`OS_EnterInterrupt()` disables nesting

Prototype

```
void OS_EnterInterrupt(void);
```

Return value

None.

Additional Information

`OS_EnterInterrupt()` has to be used as prolog function, when the interrupt handler should not be preempted by any other interrupt handler that runs on a priority below the fast interrupt priority. An interrupt handler that starts with `OS_EnterInterrupt()` has to end with the epilog function `OS_LeaveInterrupt()`.

Example

Interrupt-routine that can not be preempted by other interrupts.

```
static void _Systick(void) {
  OS_EnterInterrupt(); // Inform embOS that interrupt code is running
  OS_HandleTick();     // Can not be interrupted by higher priority interrupts
  OS_LeaveInterrupt(); // Inform embOS that interrupt handler is left
}
```

6.7.2 OS_EnterNestableInterrupt()

Description

`OS_EnterNestableInterrupt()` enables nesting.

Prototype

```
void OS_EnterNestableInterrupt(void);
```

Return value

None.

Additional Information

`OS_EnterNestableInterrupt()` allows nesting. `OS_EnterNestableInterrupt()` may be used as prolog function, when the interrupt handler may be preempted by any other interrupt handler that runs on a higher interrupt priority. An interrupt handler that starts with `OS_EnterNestableInterrupt()` has to end with the epilog function `OS_LeaveNestableInterrupt()`.

Example

Interrupt-routine that can be preempted by other interrupts.

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

6.7.3 Required embOS system interrupt handler

embOS for Cortex M core needs two exception handlers which belong to the system itself. Both are delivered with embOS. Ensure that they are referenced in the vector table.

6.7.3.1 OS_Exception() the scheduler entry

`OS_Exception()` is the scheduler entrance of embOS. It runs on the lowest interrupt priority. Whenever scheduling is required, this exception is triggered by embOS. `OS_Exception()` has to be called by the PendSV exception of the Cortex M CPU.

Ensure that the address of `OS_Exception()` is inserted in the vector table at the correct position. The vector tables which come with embOS are already setup and should be used and modified for the application.

6.7.3.2 OS_Systick() the embOS system timer handler

`OS_Systick()` is the interrupt handler which manages the system timer. The system timer is initialized during `OS_InitHW()`. The embOS system timer uses the SYSTICK timer of the Cortex M CPU and runs on a low preemption priority level which is one level higher than the lowest preemption priority level. Ensure that the address of `OS_Systick()` is inserted in the vector table at the correct position. The vector tables which come with embOS are already setup and should be used and modified for the application.

6.8 Interrupt handling with vectored interrupt controller

For Cortex M core, which has a built in vectored interrupt controller, embOS delivers additional functions to install and set up interrupt handler functions. To handle interrupts with the vectored interrupt controller, embOS offers the following functions:

6.8.1 OS_ARM_EnableISR(): Enable specific interrupt

Description

`OS_ARM_EnableISR()` is used to enable interrupt acceptance of a specific interrupt source in a vectored interrupt controller.

Prototype

```
void OS_ARM_EnableISR(int ISRIndex);
```

Parameter	Description
ISRIndex	Index of the interrupt source which should be enabled

Table 6.1: OS_EnterInterrupt() parameter list

Return value

None.

Additional Information

This function just enables the interrupt inside the interrupt controller. It does not enable the interrupt of any peripherals. This has to be done elsewhere.

6.8.2 OS_ARM_DisableISR(): Disable specific interrupt

Description

`OS_ARM_DisableISR()` is used to disable interrupt acceptance of a specific interrupt source in a vectored interrupt controller which is not of the VIC type.

Prototype

```
void OS_ARM_DisableISR(int ISRIndex);
```

Parameter	Description
ISRIndex	Index of the interrupt source which should be disabled

Table 6.2: OS_EnterInterrupt() parameter list

Return value

None.

Additional Information

This function just disables the interrupt in the interrupt controller. It does not disable the interrupt of any peripherals. This has to be done elsewhere.

6.8.3 OS_ARM_ISRSetPrio(): Set priority of specific interrupt

Description

`OS_ARM_ISRSetPrio()` is used to set or modify the priority of a specific interrupt source by programming the interrupt controller.

Prototype

```
int OS_ARM_ISRSetPrio(int ISRIndex, int Prio);
```

Parameter	Description
ISRIndex	Index of the interrupt source which should be modified.
Prio	The priority which should be set for the specific interrupt.

Table 6.3: OS_EnterInterrupt() parameter list

Return value

None.

Additional Information

This function sets the priority of an interrupt channel by programming the interrupt-controller. Please refer to CPU-specific manuals about allowed priority levels.
Note that the ISRIndex counts from 0 for the first entry in the vector table.
The first peripheral index therefore has the `ISRIndex` 16, because the first peripheral interrupt vector is located after the 16 generic vectors in the vector table.
This differs from index values used with CMSIS.
The priority value is independent of the chip-specific preemption levels. Any value between 0 and 255 can be used, were 255 always is the lowest priority and 0 is the highest priority.
The function can be called to set the priority for all interrupt sources, regardless of whether embOS is used in the specified interrupt handler or not.
Note that interrupt handlers running on priorities from 127 or higher must not call any embOS function.

6.8.4 High priority non maskable exceptions

High priority non maskable exceptions with non configurable priority like Reset, NMI and HardFault can not be used with embOS functions. These exceptions are never disabled by embOS.
Never call any embOS function from an exception handler of one of these exceptions.

Chapter 7

Vector Floating Point support

7.1 Vector Floating Point support VFPv4

Some Cortex M4 / M4F MCUs come with an integrated vectored floating point unit VFPv4.
When selecting the CPU and activating the VFPv4 support in the project options, the compiler and linker will add efficient code which uses the VFP when floating point operations are used in the application.
With embOS, the VFP registers have to be saved and restored when preemptive or cooperative task switches are performed.
For efficiency reasons, embOS does not save and restore the VFP registers for every task automatically. The context switching time and stack load are therefore not affected when the VFP unit is not used or needed.
Saving and restoring the VFP registers can be enabled for every task individually by extending the task context of the tasks which need and use the VFP.

7.1.1 OS_ExtendTaskContext_VFP()

Description

OS_ExtendTaskContext_VFP() has to be called as first function in a task, when the VFP is used in the task and the VFP registers have to be added to the task context.

Prototype

```
void OS_ExtendTaskContext_VFP(void)
```

Return value

None.

Additional Information

OS_ExtendTaskContext_VFP() extends the task context to save and restore the VFP registers during context switches.
Additional task context extension for a task by calling OS_ExtendTaskContext() is not allowed and will call the embOS error handler OS_Error() in debug builds of embOS.
There is no need to extend the task context for every task. Only those tasks using the VFP for calculation have to be extended.
When Thread-local Storage (TLS) is also needed in a task, the new embOS function OS_ExtendTaskContext_TLS_VFP() has to be called to extend the task context for TLS and VFP.

7.1.2 Using embOS libraries with VFP support

When VFP support is selected as project option, one of the embOS libraries with VFP support have to be used in the project.
The embOS libraries for VFP support require that the VFP is switched on during start-up and remains switched on during program execution.
Using your own startup code, ensure that the VFP is switched on during startup.
When the VFP unit is not switched on, the embOS scheduler will fail.
The debug version of embOS checks whether the VFP is switched on when embOS is initialized by calling OS_InitKern().
When the VFP unit is not detected or not switched on, the embOS error handler OS_Error() is called with error code OS_ERR_CPU_STATE_ILLEGAL.

7.1.3 Using the VFP in interrupt service routines

Using the VFP in interrupt service routines requires additional functions to save and restore the VFP registers.
The implementation of VFP support in embOS disables the automatic context saving

of VFP registers which is normally activated after reset.
embOS disables the VFP context saving feature of the Cortex M4F at all. This has the advantage that no additional stack is needed in tasks not using the VFP unit.

As the GNU compiler does not add additional code to save and restore the VFP registers on entry and exit of interrupt service routines, it is the users responsibility to save the VFP registers on entry of an interrupt service routine when the VFP is used in the ISR.
embOS delivers two functions to save and restore the VFP context in an interrupt service routine.

7.1.3.1 OS_VFP_Save()

Description

OS_VFP_Save() has to be called as first function in an interrupt service routine, when the VFP is used in the interrupt service routine. The function saves the temporary VFP registers on the stack.

Prototype

`void` OS_VFP_Save(`void`)

Return value

None.

Additional Information

OS_VFP_Save() declares a local variable which reserves space for all temporary floating point registers and stores the registers in the variable.
After calling the OS_VFP_Save() function, the interrupt service routine may use the VFP for calculation without destroying the saved content of the VFP registers.
To restore the registers, the ISR has to call OS_VFP_Restore() at the end.
The function may be used in any ISR regardless the priority. It is not restricted to low priority interrupt functions.

7.1.3.2 OS_VFP_Restore()

Description

OS_VFP_Restore() has to be called as last function in an interrupt service routine, when the VFP registers were saved by a call of OS_VFP_Save() at the beginning of the ISR. The function restores the temporary VFP registers from the stack.

Prototype

`void` OS_VFP_Restore(`void`)

Return value

None.

Additional Information

OS_VFP_Restore() restores the temporary VFP registers which were saved by a previous call of OS_VFP_Save().
It has to be used together with OS_VFP_Save() and should be the last function called in the ISR.

Example of a low priority interrupt service routine using VFP:

```
void ADC_ISR_Handler(void) {
  OS_VFP_Save();     // Save VFP registers
  OS_EnterInterrupt();
  DoSomeFloatOperation();
  OS_LeaveInterrupt();
  OS_VFP_Restore();  // Restore VFP registers.
}
```


In low priority interrupt service routines, OS_EnterInterrupt() is called to inform embOS that an interrupt handler is running and blocks task switches until OS_LeaveInterrupt() is called.
After calling OS_EnterInterrupt(), or OS_EnterNestableInterrupt(), any embOS function which is allowed to be called from an ISR may be called.

Example of a high priority interrupt service routine using VFP:

```
void ADC_ISR_Handler(void) {
  OS_VFP_Save();      // Save VFP registers
  DoSomeFloatOperation();
  OS_VFP_Restore();   // Restore VFP registers.
}
```

In interrupt service routines running at higher priority, no embOS functions except OS_VFP_Save() and OS_VFP_Restore may be called. Not even OS_EnterInterrupt().

Chapter 8

Technical data

This chapter lists technical data of embOS used with Cortex M CPUs.

8.1 Memory requirements

These values are neither precise nor guaranteed, but they give you a good idea of the memory requirements. They vary depending on the current version of embOS. The minimum ROM requirement for the kernel itself is about 2.500 bytes.

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

embOS resource	RAM [bytes]
Task control block	32
Resource semaphore	16
Counting semaphore	8
Mailbox	24
Software timer	20

Table 8.1: embOS memory requirements

Chapter 9

RTT and SystemView

This chapter contains information about SEGGER Real Time Transfer and SEGGER SystemView.

9.1 SEGGER Real Time Transfer

SEGGER's Real Time Transfer (RTT) is the new technology for interactive user I/O in embedded applications. RTT can be used with any J-Link model and any supported target processor which allows background memory access.

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

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

9.1.1 Shipped files related to SEGGER RTT

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

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

9.2 SEGGER SystemView

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

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

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

9.2.1 Shipped files related to SEGGER SystemView

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

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

Index

