# embOS

Real-Time
Operating System

CPU & Compiler
specifics for Cortex M
using GCC / emIDE

Document: UM01039
Software version 4.00
Revision: 0
Date: July 4, 2014

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: July 4, 2014

| Software | Revision | Date | By | Description |
|---|---|---|---|---|
| 4.00 | 0 | 140704 | MC | New software version. |
| 3.88g | 0 | 131128 | TS | New software version. |
| 3.88f | 0 | 130923 | TS | New software version. |
| 3.88c | 0 | 130815 | TS | New software version. |
| 3.88 | 0 | 130307 | TS | New software version. |
| 3.86g | 1 | 130218 | TS | Typos corrected. |
| 3.86g | 0 | 120814 | 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 with GCC / emIDE

The following chapter describes how to start with and use embOS for Cortex-M and GCC / emIDE. You should follow these steps to become familiar with embOS for Cortex-M and GCC / emIDE.

# 1.1 Installation

embOS is shipped on CD-ROM or as a zip-file in electronic form.

To install it, proceed as follows:

If you received a CD, copy the entire contents to your hard-drive into any folder of your choice. When copying, keep all files in their respective sub directories. Make sure the files are not read only after copying. If you received a zip-file, extract it to any folder of your choice, preserving the directory structure of the zip-file.

Assuming that you are using the emIDE project manager to develop your application, no further installation steps are required. You will find a lot of prepared sample start projects, which you should use and modify to write your application. So follow the instructions of section “First steps” on page 11.

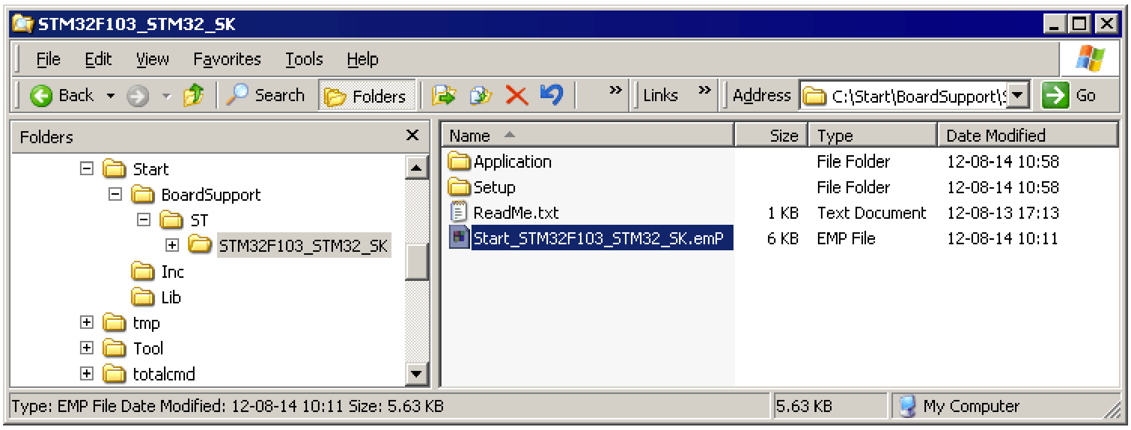
You should do this even if you do not intend to use the project manager for your application development to become familiar with embOS.

If you will not work with the emIDE, you should: Copy either all or only the library-file that you need to your work-directory. This has the advantage that when you switch to an updated version of embOS later in a project, you do not affect older projects that use embOS also. embOS does in no way rely on the emIDE project manager, it may be used without the project manager using batch files or a make utility without any problem.

## 1.2 First steps

After installation of embOS you are able to create your first multitasking application. You received ready to go sample emIDE project files and it is a good idea to use one of these as a starting point of all your applications.
Your embOS distribution contains one folder "Start\BoardSupport" which contains the sample project files and every additional files used to build your application.

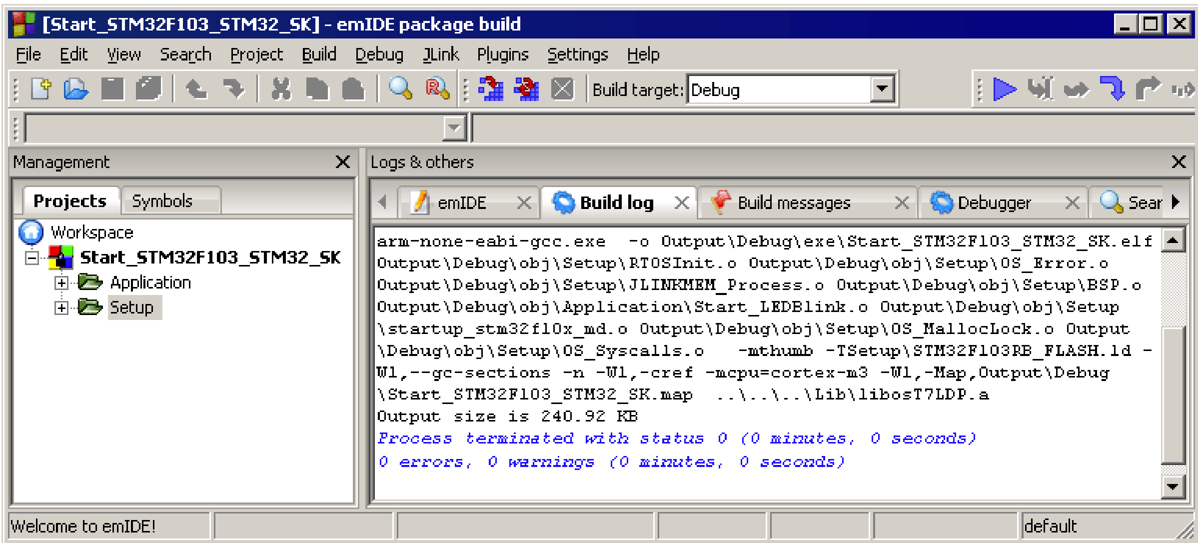
For the first step, you may use the project for STM32F103 CPU:




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 workspace **Start\BoardSupport\ST\STM32F103_STM32_SK** with the emIDE project manager (for example, by double clicking it).
- Build the start project. It should be build 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 sample application Start_LEDBlink.c

The following is a printout of the example application `Start_2Tasks.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.

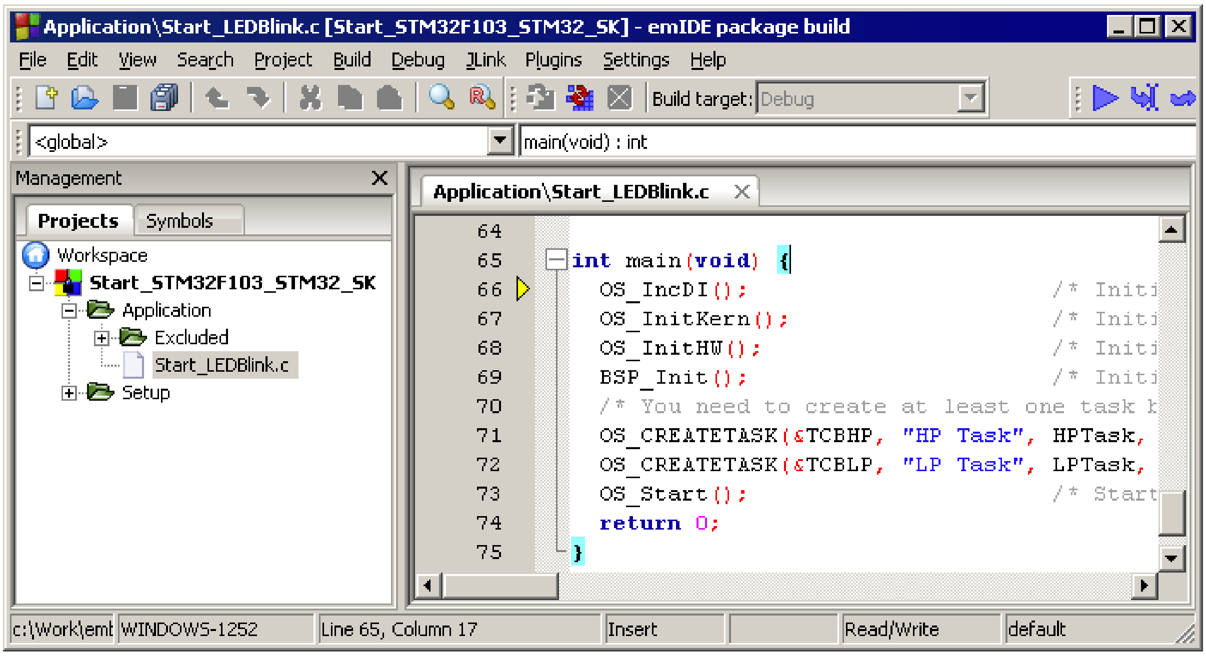
```
/**********************************************************
* SEGGER MICROCONTROLLER SYSTEME GmbH & Co.KG
* Solutions for real time microcontroller applications
**********************************************************
File    : Start2Tasks.c
Purpose : Skeleton program for embOS
--------- END-OF-HEADER ---------------------------------*/

#include "RTOS.h"

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

void HPTask(void) {
  while (1) {
    OS_Delay (10);
  }
}

void LPTask(void) {
  while (1) {
    OS_Delay (50);
  }
}

/**********************************************************
*
* main
*
**********************************************************/

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

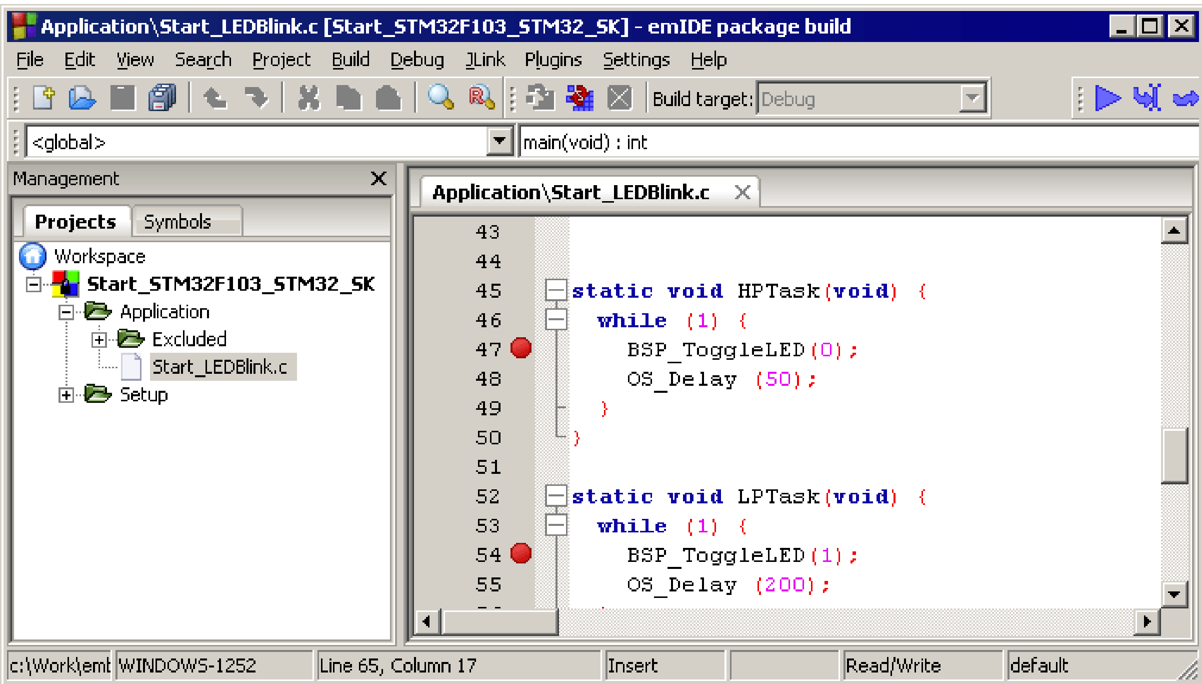
## 1.4 Stepping through the sample application

When starting the debugger, you will see the `main` function (see example screenshot below). The `main` function appears as long as the C-SPY option **Run to main** is selected, which it is 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 timer-tick-interrupt for embOS. Step through it to see what is done.

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



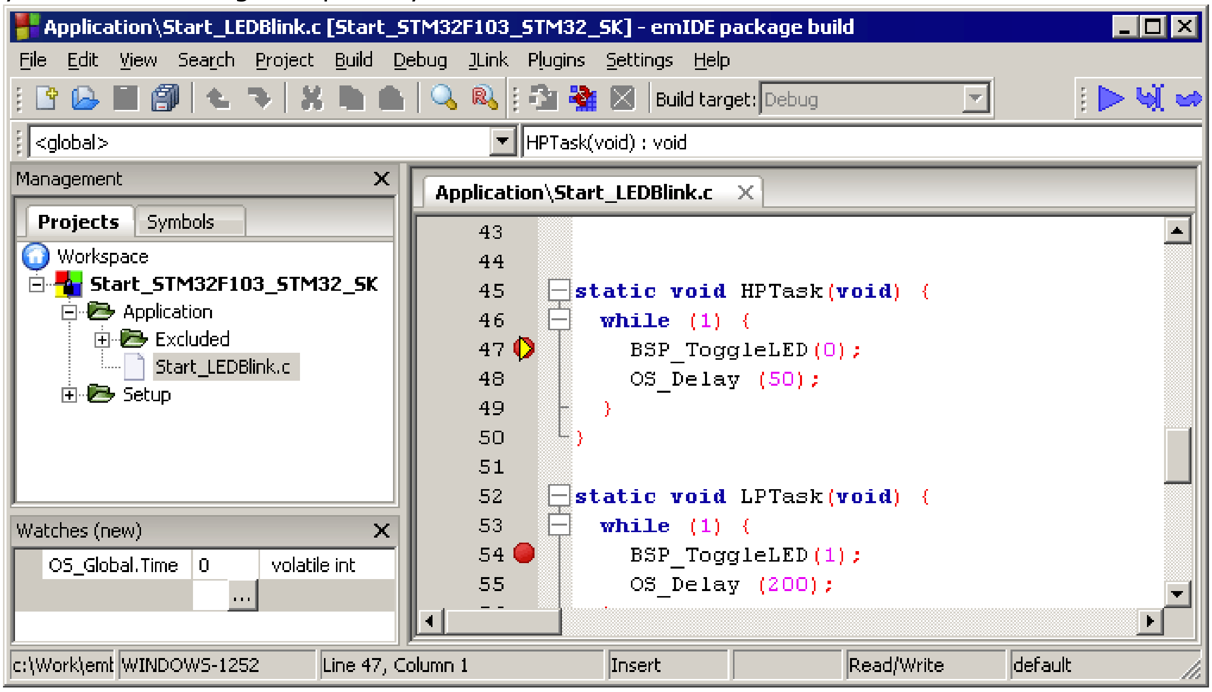

Before you continue stepping, you should set two break points in the two tasks as shown below:



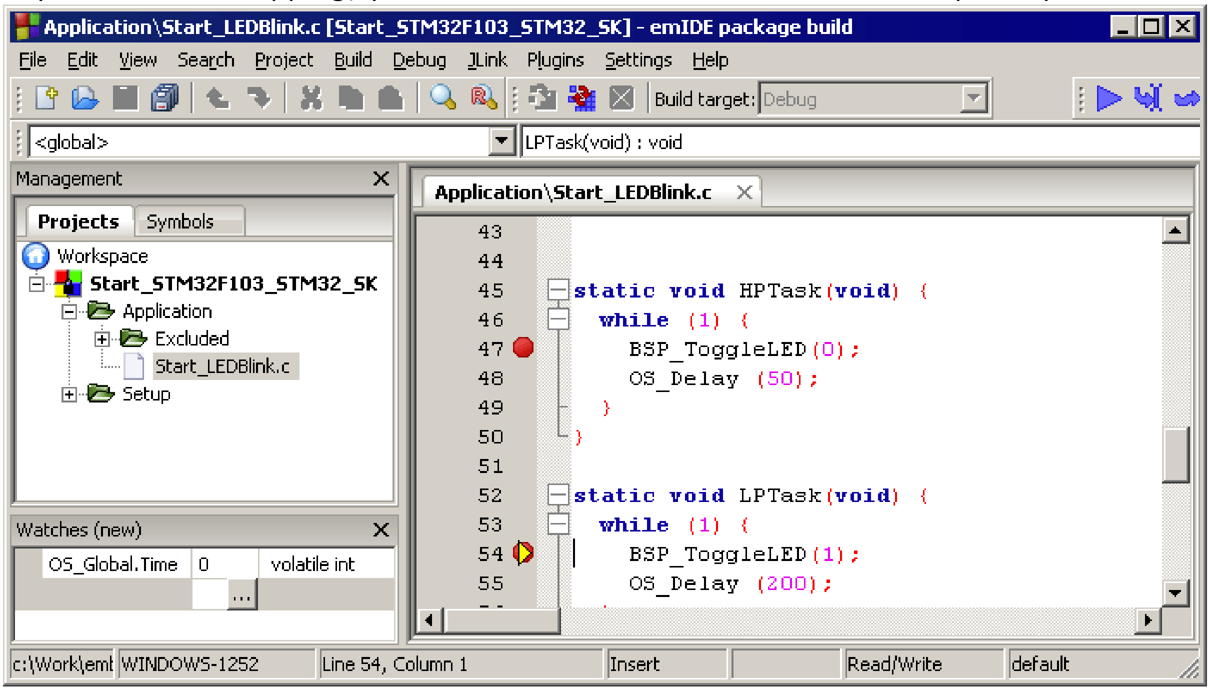

 

As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only. Y

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



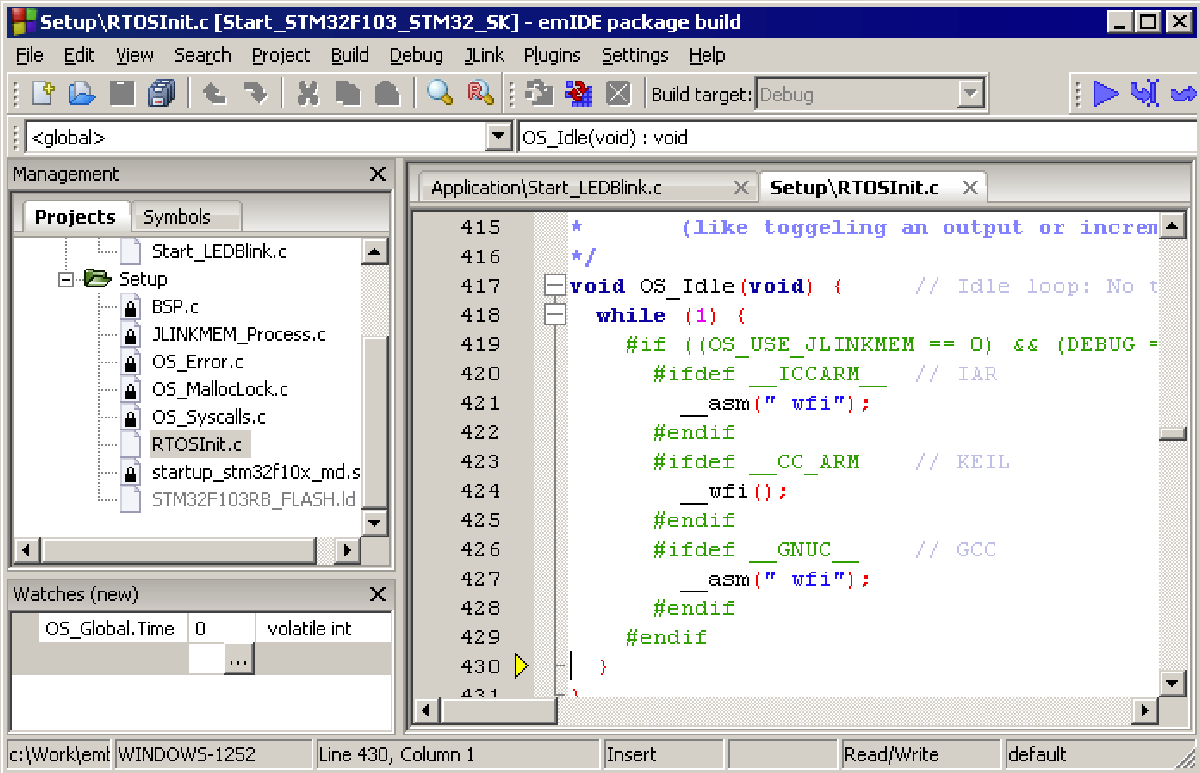
If you continue stepping, you will arrive in the task that has lower priority:



Continuing 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 which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

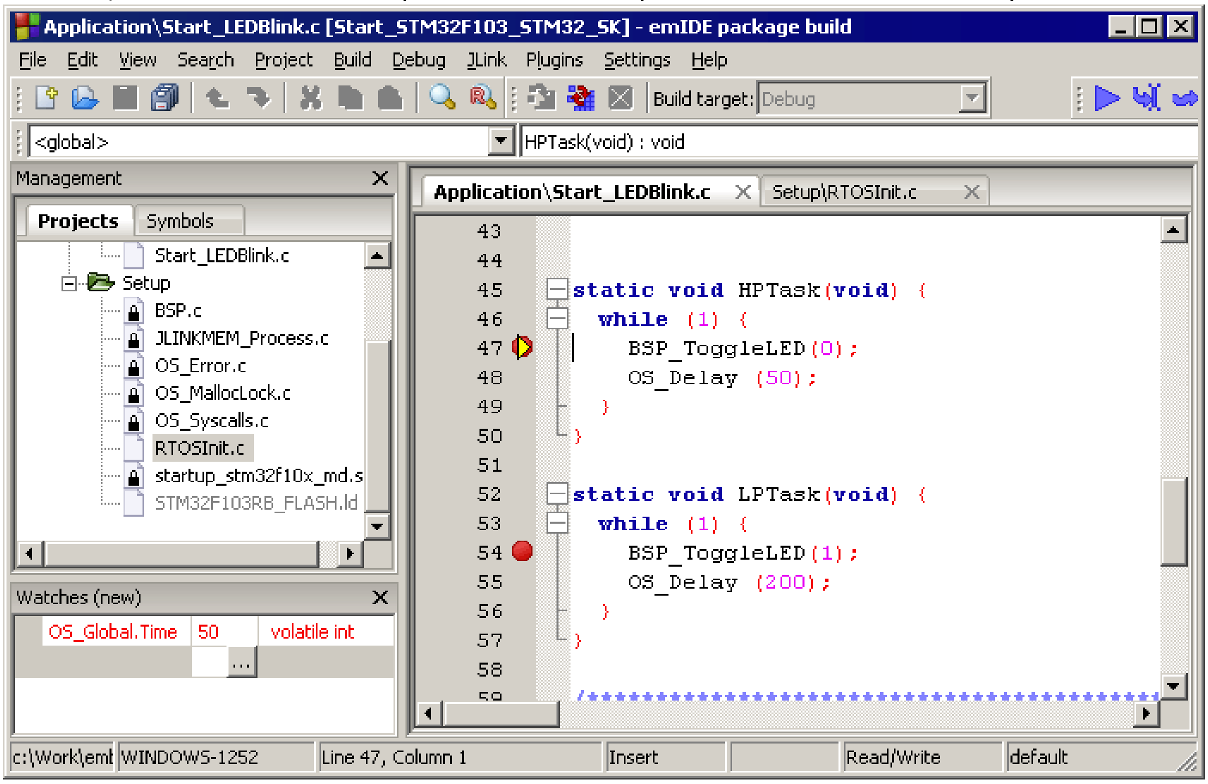
 

You will arrive there when you step into the `OS_Delay()` function in disassembly mode. `OS_Idle()` is part of `RTOSInit*.c`.You may also set a breakpoint there before you step over the delay in `LPTask`.



If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay. Press GO to enter the highest priority task again.

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 setup 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 First steps on page 9 and modify the project to fit your needs. Using a sample 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 for Cortex M

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, 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 library other than Release build library is used in your project.
- Additional low level init code 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.
Also ensure, that `main()` is called with the CPU running in supervisor or system mode.
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 `Start_2Task.c` source file in the subfolder `Application\`.

# 2.3 Change library mode

For your application you may wish to choose an other 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 library is already contained in your project, just select the appropriate configuration or enable the library and disable others.
- To add a library, you may add a library to the Lib group in the project and exclude other libraries in the group from build.
- Check and set `OS_DEBUG` define as preprocessor option. You may modify the OS_Config.h file to select an other library mode.

## 2.4 Select another CPU

embOS contains CPU-specific code for various Cortex M 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 Cortex M CPU is currently not supported, examine all `RTOSInit` files in the CPU-specific subfolders and select one which almost fits your CPU. You may have to modify `OS_InitHW()`, `OS_COM_Init()`, and the interrupt service routines for embOS timer tick and communication to embOSView and `__low_level_init()`.

The easiest way to get embOS running on an unsupported CPU is using the generic CMSIS start project and adding the device specific files from the CPU vendor.

# Chapter 3

# ARM Cortex-M specifics

## 3.1 CPU modes

embOS supports all memory and code model combinations that emIDE / GCC Cortex-M compiler supports.

## 3.2 Available libraries

embOS for Cortex-M and GCC / emIDE is shipped with 14 different libraries, one for each CPU mode / CPU core / endian mode combination.

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 | 6: Cortex M0<br>7: Cortex M3/M4<br>7V: Cortex M4F with VFP |
| ByteOrder | Endianess | B: Big endian<br>L: Little endian |
| LibMode | Library mode | XR: eXtreme Release<br>R: Release<br>S: Stack check<br>D: Debug<br>SP: Stack check + Profiling<br>DP: Debug + Profiling<br>DT: Debug + Trace |

### Example:

libosT7LDP.a is the library for a project using a CortexM3 or CortexM4 core without VFP , thumb mode, little endian mode with debug and profiling support.

# Chapter 4

# 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 local storage

The GCC newlib supports usage of thread-local storage located in a _reent structure as local variable for every task.
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 GNU is prepared to support the 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 or specific tasks.
The embOS implementation of thread-local storage allows activation of TLS separately for every task.
Only tasks that call functions using TLS need to activate the TLS by defining a local variable and calling an initialization function when the task is started.
The _reent structure is stored on the task stack and have to be considered when the task stack size is defined. The structure may contain up to 800 bytes.

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

- 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.

## 4.2.1 OS_ExtendTaskContext_TLS()

### Description

`OS_ExtendTaskContext_TLS()` may be called from a task which needs thread local storage to initialize and use Thread-local storage.

### Prototype

```
void  OS_ExtendTaskContext_TLS(struct _reent * pReentStruct)
```

### Parameter

pReentStruct is a pointer to the thread local storage. It is the address of the variable of type struct `_reent` which holds the thread local data.

### Return value

None.

### Additional Information

`OS_ExtendTaskContext_TLS()` shall be the first function called from a task when TLS should be used in the specific task. The function must not be called multiple times from one task. The thread-local storage has to be defined as local variable in the task.

### Example

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

  OS_ExtendTaskContext_TLS(&TaskReentStruct);*/
  while (1) {
    ...  /* Task functionality.  */
  }
}
```

Please ensure sufficient task stack to hold the `_reent` structure variable.

For details on the `_reent` structure, `_impure_ptr`, and library functions which require precautions on reentrance, refer to the GNU documentation.

## 4.2.2 OS_ExtendTaskContext_TLS_VFP()

### Description

`OS_ExtendTaskContext_TLS_VFP()` has to be called as first function in a task, when thread-local storage and thread safe floatingpoint processor support is needed in the task.

### Prototype

`void` OS_ExtendTaskContext_TLS_VFP(`struct _reent * pReentStruct)`

### Parameter

pReentStruct is a pointer to the thread local storage. It is the address of the variable of type struct `_reent` which holds the thread local data.

### Return value

None.

### Additional Information

`OS_ExtendTaskContext_TLS_VFP()` shall be the first function called from a task when TLS and VFP should be used in the specific task.
The function must not be called multiple times from one task.
The thread-local storage should be defined as local variable in the task.
The task specific TLS management is generated as embOS task extension together with the storage needed for the VFP registers. The VFP registers are automatically saved onto the task stack when the task is suspended, and restored, when the task is resumed. Additional task extension by a call of `OS_ExtendTaskContext()` is impossible.
The function is available in all embOS libraries with VFP support which are named libosT7Vx_xx

# 4.3 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.

## 4.3.1 __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.

## 4.3.2 __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.

## 4.4 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.

### 4.4.1 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.

### 4.4.2 Using embOS libraries with VFP support

When VFP support is selected as project option, one of the embOS libraries with VFP support has to be used in the project.
These are named libosT7Vx_xx.a.
The embOS libraries for VFP support require that the VFP is switched on during startup and remains switched on during program execution.
When selecting the VFP support in the project options, the startup code has to enable the VFP according the project options. Using CMSIS, the generic CMSIS code will automatically activate the VFP unit.
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 initailized 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`.

## 4.4.3 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 GCC 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.

### 4.4.3.1 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.

### 4.4.3.2 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()`.

## 4.4.4 Compiler and linker options.

The selection of different CPU cores or options like VFP support has to be done by linker, compiler and assembler options.
The options have to be passed to the tool by definitions in the make-files, or when using the Eclipse IDE, the options have to be defined in the "Settings" dialog for the project.
The options passed to the tools have to be defined for compiler, linker and assembler separately and have to be the same for all tools.
Beside other options, the most important options are the options to select the CPU core and the floating point support.

### 4.4.4.1 Options to select a Cortex M3 core

-mcpu=cortex-M3 -mthumb

### 4.4.4.2 Options to select a Cortex M4 core

mcpu=cortex-M4 -mthumb

### 4.4.4.3 Options to select a Cortex M4 core with VFP support

-mcpu=cortex-M4 -mthumb -mfpu=fpv4-sp-d16 -mfloat-abi=softfp

# Chapter 5

# Stacks

## 5.1 Task stack for Cortex-M

All embOS tasks execute in thread mode using the process stack pointer. The stack itself is located in any RAM location. Each task uses its individual stack. The stack-size required 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 embOS-routines.

For the Cortex-M CPU, 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 exceptions 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 your *.ld file.

## 5.3 Interrupt stack for Cortex-M

If a normal hardware exception occurs, the Cortex-M core switches to handler mode mode, which uses the main stack pointer. With embOS, the main stack pointer is initialized to use the system-stack which is defined in the linker command file. A separate irq-stack is not used, interrupts run on the system stack.

# Chapter 6

# Interrupts

The Cortex-M core comes with an built in vectored interrupt controller which supports up to 496 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 form 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 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.

The interrupt vector table is located in a “C” source file in the CPU specific sub-folder. All interrupt handler function addresses have to be inserted in the vector table.

When using your own interrupt vector table, ensure that the addresses of the embOS exception handlers `OS_Exception()` and `OS_Systick()` are inserted in the vector table in the correct position.

## 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 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 fast interrupt function.

# 6.6 Interrupt priorities

The Cortex-M supports 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.

At least, 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 **Fast interrupts**, interrupt priorities useable 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 Fast 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 handler running at low priorities (from 128 to 255) not calling any embOS API function are allowed, but must not re-enable interrupts!

The priority limit between embOS interrupts and **Fast interrupts** is fixed to 128 and can only be changed by recompiling embOS libraries!

## 6.6.1 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 0xFF which is always the lowest preemption priority, regardless of the number of preemption levels.

## 6.6.2 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.

## 6.6.3 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.

## 6.6.4 Priority of application interrupts

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.

Interrupt handler which require fastest reaction may run on higher priorities than 128, but must not call any embOS function (->Fast 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);
```

## 6.6.5 Priority grouping

The number of preemption levels may be limited by programming the priority group level in the application interrupt and reset control register of the chip. embOS does not modify this register, thus allowing the maximum number of preemption levels which are implemented by the chip design.

It is recommended, not to change the priority grouping setting.

# 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 that 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(), disable nesting

`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(), allow 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 allows preemption by higher prioritized interrupts

```
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
}
```

# 6.8 Required embOS system interrupt handler

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

## 6.8.1 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.

## 6.8.2 OS_Systick() the embOS system timer handler

`OS_Systick()` is the interrupt handler which manages the system time. 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.

# 6.9 Interrupt handling with vectored interrupt controller

For Cortex-M, which has a built in vectored interrupt controller, embOS delivers additional functions to install and setup interrupt handler functions.

To handle interrupts with the vectored interrupt controller, embOS offers the following functions:

## 6.9.1 OS_ARM_InstallISRHandler(): Install an interrupt handler

### Description

`OS_ARM_InstallISRHandler()` is used to install a specific interrupt vector when ARM CPUs with vectored interrupt controller are used.

### Prototype

```
OS_ISR_HANDLER* OS_ARM_InstallISRHandler(int              ISRIndex,
                                         OS_ISR_HANDLER * pISRHandler);
```

| Parameter | Meaning |
|---|---|
| ISRIndex | Index of the interrupt source, normally the interrupt vector number. |
| pISRHandler | Address of the interrupt handler function. |

### Return value

`OS_ISR_HANDLER *`: the address of the previous installed interrupt function, which was installed at the addressed vector number before.

### Add. information

This function just installs the interrupt vector but does not modify the priority and does not automatically enable the interrupt.

When the interrupt vector table should be located in RAM, the first call of this function copies the vector table into RAM and programs the interrupt controller to use the RAM table.

When the interrupt vector table should reside in ROM, the function does nothing and always returns “NULL”.

## 6.9.2 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 | Meaning |
|---|---|
| ISRIndex | Index of the interrupt source which should be enabled. |

### Return value

NONE.

### Add. 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.

## 6.9.3 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 | Meaning |
|---|---|
| ISRIndex | Index of the interrupt source which should be disabled. |

### Return value

NONE.

### Add. 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.

## 6.9.4 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);
```

### Return value

Previous priority which was assigned before the call of `OS_ARM_ISRSetPrio()`.

### Add. 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.

## 6.10 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

# CMSIS

ARM introduced the Cortex Microcontroller Software Interface Standard (CMSIS) as a vendor independent hardware abstraction layer for simplifying software re-use.
The standard enables consistent and simple software interfaces to the processor, for peripherals, for real time operating systems as embOS and other middleware.
As SEGGER is one of the CMSIS partners, embOS for Cortex M is fully CMSIS compliant.
embOS comes with a generic CMSIS start projects which should run on any Cortex M CPU. All other start projects are also fully CMSIS compliant and can be used as starting points for CPU specific CMSIS projects.
How to use the generic project and adding vendor specific files to this or other projects is explained in the following chapters.

# 7.1 The generic CMSIS start projects

The folder Start\BoardSupport\CMSIS contains a generic CMSIS start projects that should run on any Cortex M core.
The subfolder DeviceSupport\ contains the device specific source and header files which have to be replaced by the device specific files of the Cortex M vendor to make the CMSIS sample start projects device specific.

## 7.2 Device specific files needed for embOS with CMSIS

- ***Device*.h**: Contains the device specific exception and interrupt numbers and names. embOS needs the Cortex M generic exception names PendSV_IRQn and SysTick_IRQn only which are vendor independent and common for all devices. The generic sample files delivered with embOS do not contain any peripheral interrupt vector numbers and names as those are not needed by embOS.
  To make the embOS CMSIS samples device specific and allow usage of peripheral interrupts, the Device.h file has to be replaced by the one which is delivered from the CPU vendor.
- **System_*Device*.h**: Declares at least the two required system timer functions which are used to initialize the CPU clock system and one variable which allows the application software to retrieve information about the current CPU clock speed. The names of the clock controlling functions and variables are defined by the CMSIS standard and are therefore identical in all vendor specific implementations.
- **System_*Device*.c**: Implements the core specific functions to initialize the CPU, at least to initialize the core clock. The sample file delivered with embOS contains empty dummy functions and has to be replaced by the vendor specific file which contains the initialization functions for the core.
- **Startup_*Device*.s**: The startup file which contains the initial reset sequence and contains exception handler and peripheral interrupt handler for all interrupts.
  The handler functions are declared weak, so they can be overwritten by the application which implements the application specific handler functionality.
  The sample which comes with embOS only contains the generic exception vectors and handler and has to be replaced by the vendor specific startup file.

The reset handler HAS TO CALL the **SystemInit()** function which is delivered with the core specific system functions.

## 7.3 Device specific functions/variables needed for embOS with CMSIS

The embOS system timer is triggered by the Cortex M generic system timer. The correct core clock and pll system is device specific and has to be initialized by a low level init function called from the startup code.
embOS calls the CMSIS function *SysTick_Config()* to set up the system timer. The function relies on the correct core clock initialization performed by the low level initialization function *SystemInit()* and the value of the core clock frequency which has to be written into the *SystemCoreClock* variable during initialization.

- **SystemInit()**:The system init function is delivered by the vendor specific CMSIS library and is normally called from the reset handler in the startup code. The system init function has to initialize the core clock and has to write the CPU frequency into the global variable *SystemCoreClock*.
- **SystemCoreClock**: Contains the current system core clock frequency and is initialized by the low level initialization function *SystemInit()* during startup. embOS for CMSIS relies on the value in this variable to adjust its own timer and all time related functions.

Any other files or functions delivered with the vendor specific CMSIS library may be used by the application, but are not required for embOS.

## 7.4 CMSIS generic functions needed for embOS with CMSIS

The embOS system timer is triggered by the Cortex M generic system timer which has to be initialized to generate periodic interrupts in a specified interval. The configuration function *SysTick_Config()* for the system timer relies on correct initialization of the core clock system which is performed during startup.

- **SystemCoreClockUpdate**: This CMSIS function has to update the *SystemCoreClock* variable according the current system timer initialization. The function is device sepcific and may be called before the *SystemCoreClock* variable is accessed or any function which relies on the correct setting of the system core clock variable is called. embOS calls this function during the hardware initialization function *OS_InitHW()* before the system timer is initialized.
- **SysTick_Config**: This CMSIS generic function is declared an implemented in the core_cmX.h file. It initializes and starts the SysTick counter and enables the SysTick interrupt. For embOS it is recommended to run the SysTick interrupt at the second lowest preemption priority. Therefore, after calling the *SysTick_Config()* function from *OS_InitHW()*, the priority is set to the second lowest preemption priority ba a call of *NVIC_SetPriority()*.
  The embOS function *OS_InitHW()* has to be called after initialization of embOS during main and is implemented in the *RTOSInit_CMSIS.c* file.
- **SysTick_Handler**: The embOS timer interrupt handler, called periodically by the interrupt generated from the SysTick timer. The SysTick_Handler is declared weak in the CMSIS startup code and is replaced by the embOS Systick_Handler function implemented in RTOSInit_CMSIS.c which comes with the embOS start project.
- **PendSV_Handler**: The embOS scheduler entry function. It is declared weak in the CMSIS startup code and is replaced by the embOS internal function contained in the embOS library. The embOS initialization code enables the *PendSV* exception and initializes the priority. The application **MUST NOT** change the *PendSV* priority.

## 7.5 Customizing the embOS CMSIS generic start project

The embOS CMSIS generic start projects run on every Cortex M CPU. As the generic device specific functions delivered with embOS do no not initialize the core clock system and the pll, the timing is not correct, a real CPU will run very slow.
To run the sample project on a specific Cortex M CPU, replace all files in the *Device-Support* folder by the versions delivered by the CPU vendor. The vendor and CPU specific files should be found in the CMSIS release package, or are available from the core vendor.
No other changes are necessary on the start project or any other files.
To run the generic CMSIS start project on a Cortex M0, you have to replace the embOS libraries by libraries for Cortex M0 and have to add Cortex M0 specific vendor files.

## 7.6 Adding CMSIS to other embOS start projects

All CPU specific start projects are fully CMSIS compatible. If required or wanted in the application, the CMSIS files for the specific CPU may be added to the project without any modification on existing files.
Note that the *OS_InitHW()* function or *__low_level_init()* in the RTOSInit file initialize the core clock system and pll of the specific CPU. The system clock frequency and core clock frequency are defined in the RTOSInit file.
If the application needs access to the *SystemCoreClock*, the core specific CMSIS startup code and core specific initialization function *SystemInit* has to be included in the project.
In this case, the *__low_level_init()* function and the *OS_InitHW()* function in RTOSInit may be replaced, or the CMSIS generic *RTOSInit_CMSIS.c* file may be used in the project.

## 7.7 Interrupt and exception handling with CMSIS

The embOS CPU specific projects come with CPU specific vector tables and empty exception and interrupt handlers for the specific CPU. All handlers are named according the names of the CMSIS device specific handlers and are declared weak and can be replaced by an implementation in the application source files.
The CPU specific vector table and interrupt handler functions in the embOS start projects can be replaced by the CPU specific CMSIS startup file of the CPU vendor without any modification on other files in the project.
embOS uses the two Cortex M generic exceptions PendSV and SysTick and delivers its own handler functions to handle these exceptions.
All peripheral interrupts are device specific and are not used with embOS except for profiling support and system analysis with embOSView using a UART.

# 7.8 Enable and disable interrupts

The generic CMSIS functions *NVIC_EnableIRQ()* and *NVIC_DisableIRQ()* can be used instead of the embOS functions *OS_ARM_EnableISR()* and *OS_ARM_DisableISR()* functions which are implemented in the CPU specific RTOSInit files delivered with embOS.
To enable and disable interrupts in general, the embOS functions OS_IncDI() and OS_DecRI() or other embOS functions described in the generic embOS manual should be used instead of the intrinsic functions from the CMSIS library.

## 7.9 Setting the Interrupt priority

With CMSIS, the CMSIS generic function *NVIC_SetPriority()* can be used instead of the *OS_ARM_ISRSetPrio()* function which is implemented in the CPU specific *RTOSInit* files delivered with embOS.
About interrupt priorities in an embOS project, read chapter 6.5 and 6.6.

# Chapter 8

# STOP / WAIT Mode

## 8.1 Introduction

In case your controller does support some kind of power saving mode, it should be possible to use it also with embOS, as long as the timer keeps working and timer interrupts are processed. To enter that mode, you usually have to implement some special sequence in the function `OS_Idle()`, which you can find in embOS module `RTOSInit.c`.
Per default, the `wfi` instruction is executed in `OS_Idle()` to put the CPU into a low power mode.

# Chapter 9

# Technical data

# 9.1 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 kernel itself has a minimum ROM size requirement of about 1.700 bytes.

In the table below, which is for 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 | 48 |
| Resource semaphore | 16 |
| Counting semaphore | 8 |
| Mailbox | 24 |
| Software timer | 20 |

# Chapter 10

# Files shipped with embOS

## List of files shipped with embOS

| Directory | File | Explanation |
|---|---|---|
| root | *.pdf | Generic API and target specific documentation. |
| root | Release.html | Version control document. |
| root | embOSView.exe | Utility for runtime analysis, described in generic documentation. |
| Start\BoardSupport\ | | Sample workspaces and project files emIDE, contained in manufacturer specific sub folders. |
| Start\Inc | RTOS.h<br>BSP.h | Include file for embOS, to be included in every C-file using embOS functions. |
| Start\Lib | os*.a | embOS libraries. |
| Start\BoardSupport\..\Setup | OS_Error.c | embOS runtime error handler used in stack check or debug builds. |
| Start\BoardSupport\...\Setup\ | *.* | CPU specific hardware routines for various CPUs. |

Any additional files shipped serve as example.

# Index