embOS

Real-Time Operating System

CPU & Compiler specifics for Cortex-M using Cypress PSoC Creator

Document: UM01041
Software Version: 5.00
Revision: 0
Date: June 13, 2018

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

Copyright notice

Trademarks

Contact address

SEGGER Microcontroller GmbH

In den Weiden 11
D-40721 Hilden

Germany

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

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please inform us and we will try to assist you as soon as possible. Contact us for further information on topics or functions that are not yet documented.

Print date: June 13, 2018

Software	Revision	Date	By	Description
5.00	0	180613	MM	New software version.
4.38	0	171019	MC	New software version.
4.34	0	170509	MC	New software version.
4.06b	0	150416	SC	Typos corrected.
4.04a	0	150213	SC	New software version.
3.88f	0	131011	TS	First FrameMaker version.

About this document

Assumptions

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

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

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

How to use this manual

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

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections.

Table of contents

Chapter 1

Using embOS

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* on page 11.

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

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

1.2 First Steps

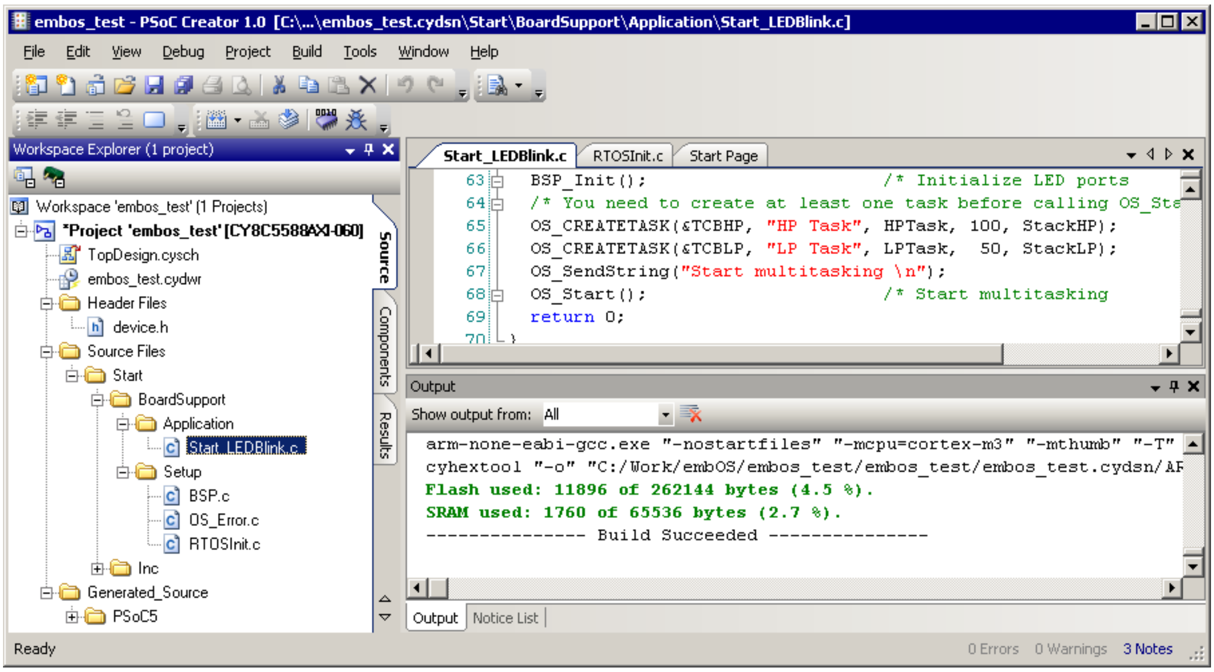
After installation of embOS you can create your first multitasking application. You have received several ready to go sample start workspaces and projects and every other files needed in the subfolder `Start`. It is a good idea to use one of them as a starting point for all of your applications. The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

To start with, you may use any project from `BoardSupport` subfolder.

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

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

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



For additional information you should open the ReadMe.txt file which is part of every specific project. The ReadMe file describes the different configurations of the project and gives additional information about specific hardware settings of the supported eval boards, if required.

1.3 The example application OS_StartLEDBlink.c

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

What happens is easy to see:

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

```
/*********************************************************************
*                 SEGGER Microcontroller GmbH & Co. KG               *
*                         The Embedded Experts                       *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------
File    : OS_StartLEDBlink.c
Purpose : embOS sample program running two simple tasks, each toggling
          a LED of the target hardware (as configured in BSP.c).
*/

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

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

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

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

/*********************************************************************
*
*       main()
*/
int main(void) {
  OS_Init();    // Initialize embOS
  OS_InitHW();  // Initialize required hardware
  BSP_Init();   // Initialize LED ports
  OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
  OS_TASK_CREATE(&TCBLP, "LP Task",  50, LPTask, StackLP);
  OS_Start();   // Start embOS
  return 0;
}

/*************************** End of file ****************************/
```

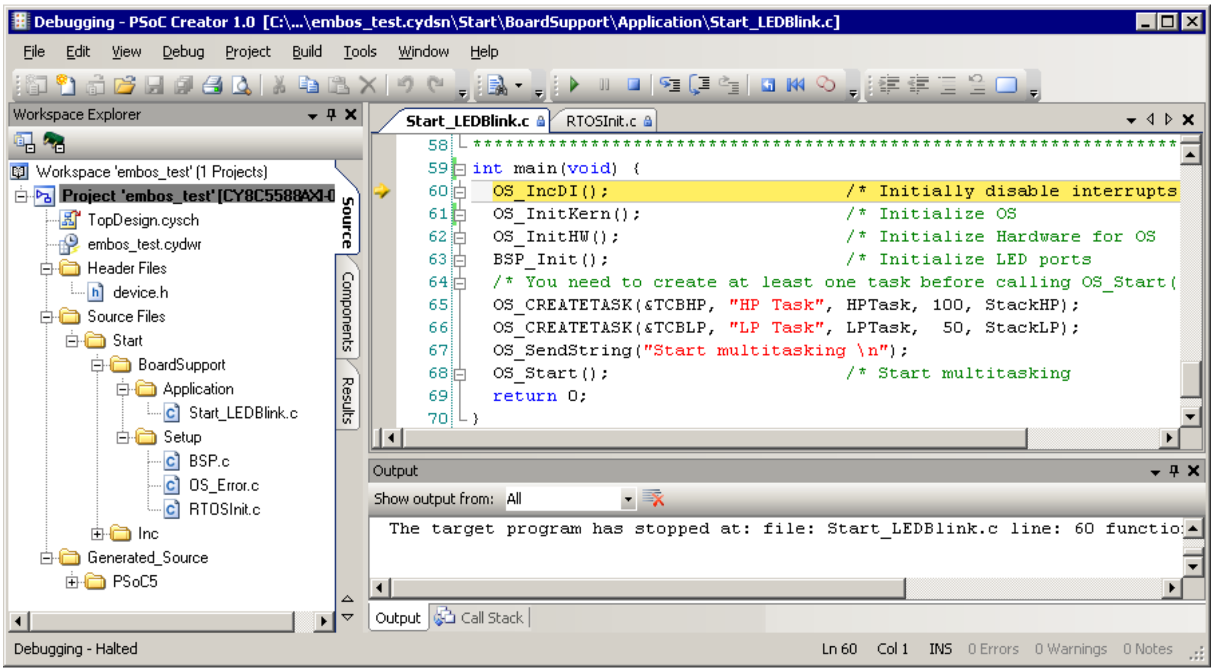
1.4 Stepping through the sample application

When starting the debugger, you will see the `main()` function (see example screen shot below). The `main()` function appears as long as project option `Run to main` is selected, which it is enabled by default. Now you can step through the program.

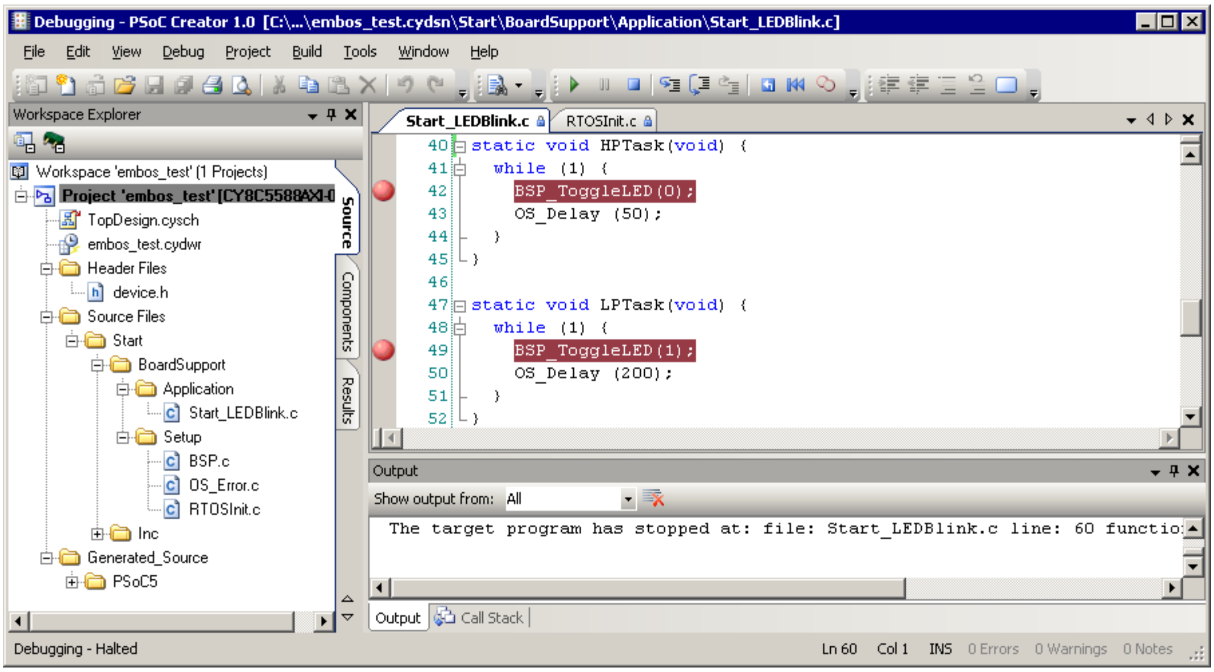
`OS_Init()` is part of the embOS library and written in assembler; you can there fore only step into it in disassembly mode. It initializes the relevant OS variables.

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

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

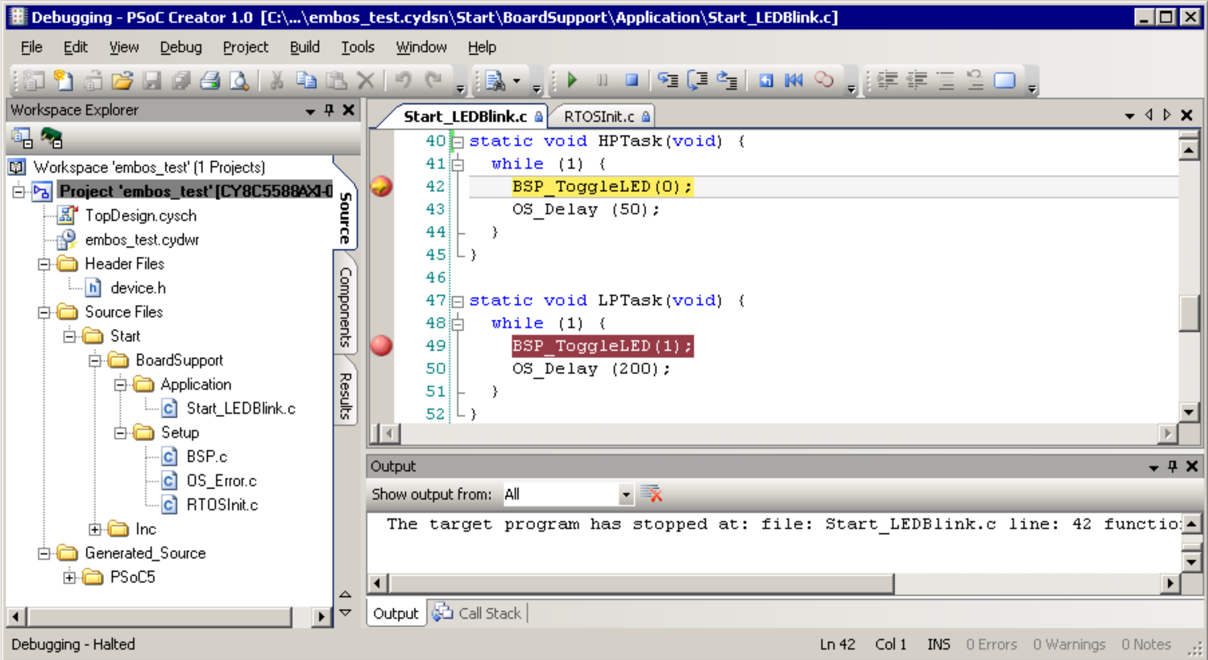


Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.

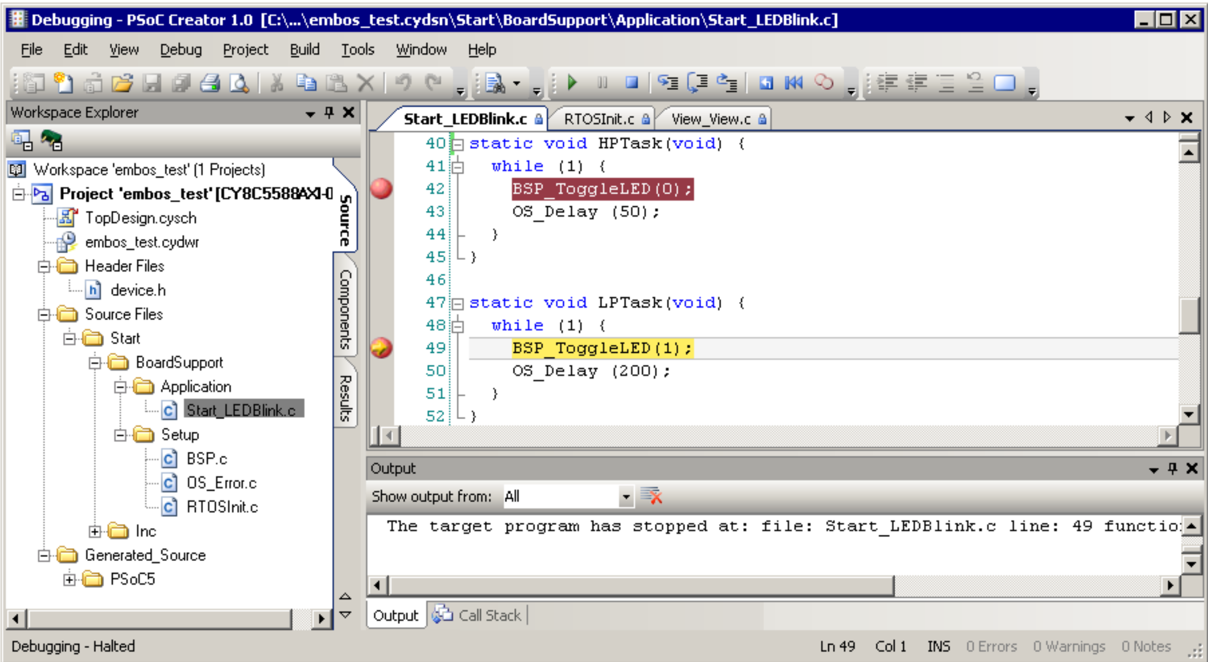


As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only.

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

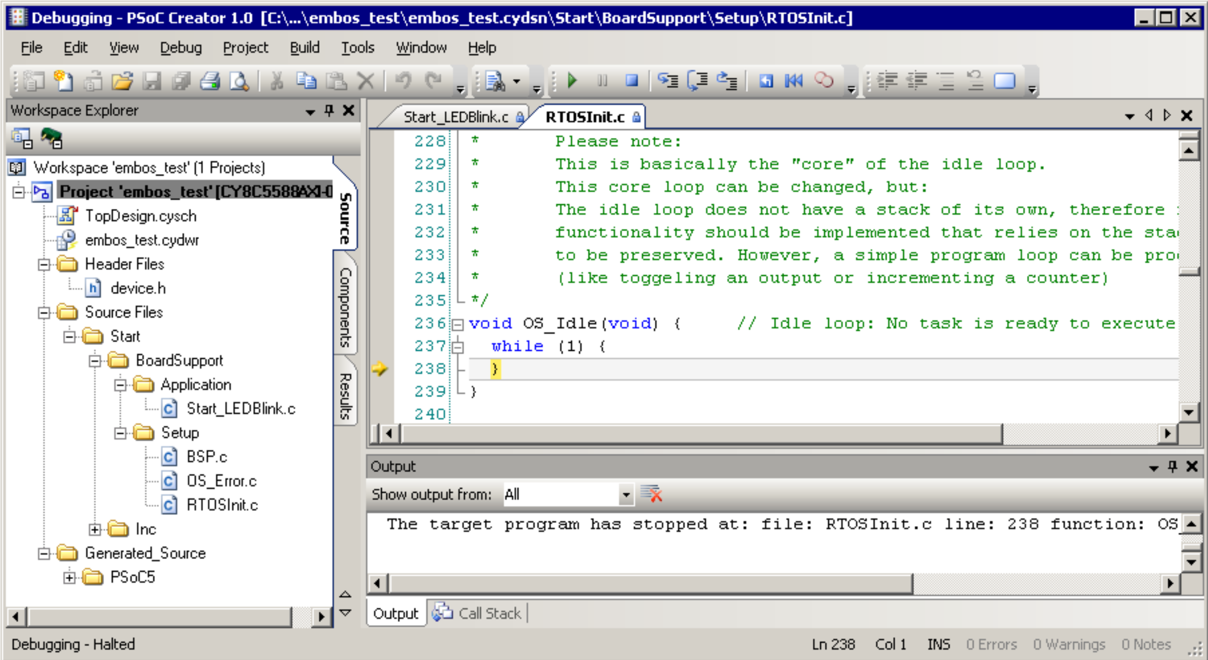


If you continue stepping, you will arrive at the task that has lower priority:



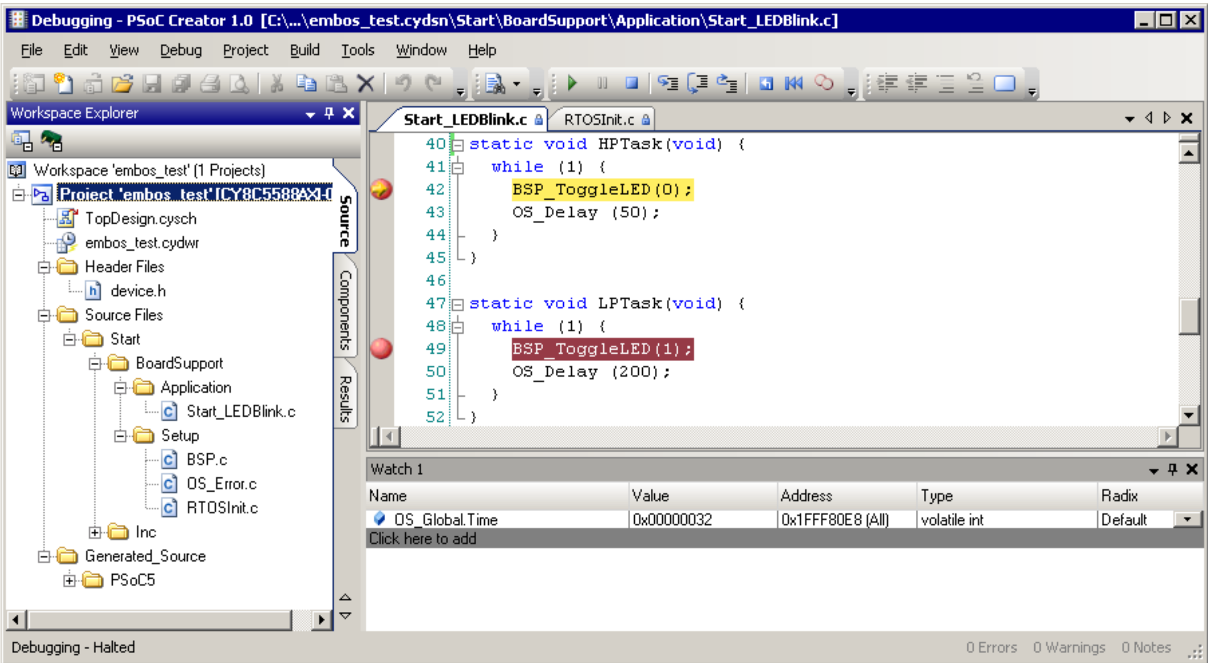
Continue to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

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



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

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



Chapter 2

Build your own application

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* on page 11 and modify the project to fit your needs. Using an embOS start project as starting point has the advantage that all necessary files are included and all settings for the project are already done.

2.2 Required files for an embOS

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

- `RTOS.h` from 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.
- `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.
- One embOS library from the subfolder `Lib\`.
- Additional CPU and compiler specific files may be required according to CPU.

When you decide to write your own startup code or use a low level `init()` function, ensure that non-initialized variables are initialized with zero, according to C standard. This is required for some embOS internal variables. Your `main()` function has to initialize embOS by calling `OS_Init()` and `OS_InitHW()` prior to any other embOS functions that are called.

2.3 Change library mode

For your application you might want to choose another library. For debugging and program development you should 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	6 : Cortex-M0 / M0+ / M1, PSoC4 7 : Cortex-M3 / M4 / M7, PSoC5
ByteOrder	Endianess	B : Big endian L : Little endian
LibMode	Library mode	XR: Extreme Release R : Release S : Stack check SP: Stack check + profiling D : Debug DP: Debug + profiling DT: Debug + profiling + 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.

> **Note**
>
> The libraries for Cortex M3 can also be used for Cortex M4.

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 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_TASK_SetContextExtensionTLS()

Description

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

Prototype

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

Parameters

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

Additional information

`OS_TASK_SetContextExtensionTLS()` 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_TASK_SetContextExtensionTLS(&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.

4.3 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 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.

4.4.1 Options to select a Cortex M3 core

```
-mcpu=cortex-M3 -mthumb
```

4.4.2 Options to select a Cortex M4 core

```
mcpu=cortex-M4 -mthumb
```

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

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 112 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 512 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 softwaretimers 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 an built-in vectored interrupt controller which supports up to 240 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_INT_EnterNestable();  // Inform embOS that interrupt code is running
  OS_HandleTick();         // May be interrupted
  OS_INT_LeaveNestable();  // Inform embOS that interrupt handler is left
}
```

6.3 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 stackswitching in an interrupt routine which runs on the main stack. The routines `OS_INT_EnterIntStack()` and

OS_INT_LeaveIntStack() are supplied for source code compatibility to other processors only and have no functionality.

6.5 Zero latency interrupts

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 fast interrupts. You must not execute any embOS function from within a fast interrupt function.

6.6 Interrupt priorities

This chapter describes interrupt priorities supported by the Cortex-M core. 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 preemtion 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 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 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_INT_Enter() or OS_INT_EnterNestable() and have to end with OS_INT_Leave() or OS_INT_LeaveNestable().
- Any Fast interrupt (running at priorities from 0 to 127) must not call any embOS API function. Even OS_INT_Enter() and OS_INT_Leave() 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 Fast interrupts is fixed to 128 and can only be changed by recompiling 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 schedulers 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-M cores

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 pri- ority levels than the embOS scheduler and the embOS system timer to allow preemp- tion of theses interrupt handlers. Interrupt handlers which require fast reaction may run on higher priorities than 128, but must not call any embOS function (fast inter- rupts). 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 docu- mented, 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_INT_Enter()

Description

`OS_INT_Enter()` disables nesting

Prototype

```
void OS_INT_Enter (void);
```

Additional information

`OS_INT_Enter()` 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_INT_Enter()` has to end with the epilog function `OS_INT_Leave()`.

Example

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

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

6.7.2 OS_INT_EnterNestable()

Description

`OS_INT_EnterNestable()` enables nesting.

Prototype

```
void OS_INT_EnterNestable (void);
```

Additional information

`OS_INT_EnterNestable()`, allow nesting. `OS_INT_EnterNestable()` 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_INT_EnterNestable()`} has to end with the epilog function `OS_INT_LeaveNestable()`.

Example

Interrupt-routine that can be preempted by other interrupts.

```
static void _Systick(void) {
  OS_INT_EnterNestable();  // Inform embOS that interrupt code is running
  OS_HandleTick();         // Can be interrupted by higher priority interrupts
  OS_INT_LeaveNestable();  // 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.

6.8 Interrupt handling API

For the Cortex-M core, 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:

6.8.1 OS_ARM_ISRInit(): Initializes interrupt handling

Description

`OS_ARM_ISRInit()` is used to initialize the interrupt handling.

Prototype

```
void OS_ARM_ISRInit(OS_U32          IsVectorTableInRAM,
                    OS_U32          NumInterrupts,
                    OS_ISR_HANDLER* VectorTableBaseAddr[],
                    OS_ISR_HANDLER* RAMVectorTableBaseAddr[]);
```

Parameters

Parameter	Description
IsVectorTableInRAM	Defines whether a RAM vector table is used. 0: Vector table in Flash. 1: Vector table in RAM.
NumInterrupts	Number of implemented interrupts.
VectorTableBaseAddr	Flash vector table address.
RAMVectorTableBaseAddr	RAM vector table address.

Additional information

This function must be called before `OS_ARM_EnableISR()`, `OS_ARM_InstallISRHandler()`, `OS_ARM_DisableISR()`, `OS_ARM_ISRSetPrio()` can be called.

Example

```
void OS_InitHW(void) {
  OS_ARM_ISRInit(1u, 82, (void*)__Vectors, pRAMVectTable);
  OS_ARM_InstallISRHandler(OS_ISR_ID_TICK, OS_Systick);
  OS_ARM_ISRSetPrio(OS_ISR_ID_TICK, 0xE0u);
  OS_ARM_EnableISR(OS_ISR_ID_TICK);
}
```


6.8.2 OS_ARM_InstallISRHandler

Description

`OS_ARM_InstallISRHandler()` installs an interrupt handler

Prototype

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

Parameters

Parameter	Description
ISRIndex	Index of the interrupt source which should be installed. Note that the index counts from 0 for the first entry in the vector table.
pISRHandler	Address of the interrupt handler.

Additional information

Sets an interrupt handler in the RAM vector table. Does nothing when the vector table is in Flash. `OS_ARM_InstallISRHandler()` copies the vector table from Flash to RAM when it is called for the first time and RAM vector table is enabled.

Example

```
void OS_InitHW(void) {
  OS_ARM_ISRInit(1u, 82, (void*)__Vectors, pRAMVectTable);
  OS_ARM_InstallISRHandler(OS_ISR_ID_TICK, OS_Systick);
  OS_ARM_ISRSetPrio(OS_ISR_ID_TICK, 0xE0u);
  OS_ARM_EnableISR(OS_ISR_ID_TICK);
}
```


6.8.3 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);
```

Parameters

Parameter	Description
ISRIndex	Index of the interrupt source which should be enabled. Note that the index counts from 0 for the first entry in the vector table.

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. 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.

6.8.4 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);
```

Parameters

Parameter	Description
`ISRIndex`	Index of the interrupt source which should be disabled. Note that the index counts from 0 for the first entry in the vector table.

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. 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.

6.8.5 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);
```

Parameters

Parameter	Description
ISRIndex	Index of the interrupt source which should be modified. Note that the index counts from 0 for the first entry in the vector table.
Prio	The priority which should be set for the specific interrupt. Prio ranges from 0 (highest priority) to 255 (lowest priority).

Additional information

This function sets the priority of an interrupt channel by programming the interruptcontroller. 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 or not in the specified interrupt handler. Note that interrupt handlers running on priorities from 127 or higher must not call any embOS function.

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 project which should run on any Cortex-M3 CPU. All other start projects, even those not based on CMSIS, 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 project

The folder `Start\BoardSupport\CMSIS` contains a generic CMSIS start project that should run on any Cortex-M3 / M4 / M4F core. The subfolder `DeviceSupport\` contains the device specific source and header files which have to be replaced by the device specific files of the CM3 / CM4 vendor to make the CMSIS sample start project 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-M3 generic exception names `PendSV_IRQn` and `SysTick_IRQn` only which are vendor independent and common for all devices. The sample file delivered with embOS does not contain any peripheral interrupt vector numbers and names as those are not needed by embOS. To make the embOS CMSIS sample device specific and allow usage of peripheral interrupts, this 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.

Startup code requirements:

The reset handler HAS TO CALL the **`SystemInit()`** function which is delivered with the core specific system functions. When using a Cortex-M4 or M4F CPU which may have a VFPv4 floating point unit equipped, please ensure that the reset handler activates the VFP when VFPv4 is selected in the project options. When VFP-support is not selected, the VFP should not be switched on. Otherwise, the `SystemInit()` function delivered from the device vendor should also honor the project settings and enable the VFP or keep it disabled according the project settings. Using CMSIS compliant startup code from the chip vendors may require modification if it enables the VFP unconditionally.

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 or after calling `SystemCoreClockUpdate()`.

- **`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 sys- tem 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 specific 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_cm3.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 project should run on every Cortex-M3 / M4 or M4F CPU. As the generic device specific functions delivered with embOS do 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-M3 / M4 / M4F CPU, replace all files in the `DeviceSupport\` 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 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, `OS_InitHW()` function in RTOSInit may be replaced, or the CMSIS generic `RTOSInit_CMSIS.c` file may be used in the project.

7.6.1 Differences between embOS projects and CMSIS

Several embOS start projects are not based on CMSIS but are fully CMSIS compliant and can be mixed with CMSIS libraries from the device vendors. Switching from embOS to CMSIS, or mixing embOS with CMSIS functions is possible without problems, but may require some modification when the interrupt controller setup functions from CMSIS shall be used instead of the embOS functions.

7.6.1.1 Different peripheral ID numbers

Using CMSIS, the peripheral IDs to setup the interrupt controller start from 0 for the first peripheral interrupt. With emboS, the first peripheral is addressed with ID number 16. embOS counts the first entry in the interrupt vector table from 0, so, the first peripheral interrupt following the 16 Cortex system interrupt entries, is 16. When the embOS functions should be replaced by the CMSIS functions, this correction has to be taken into account, or if available, the symbolic peripheral id numbers from the CPU specific CMSIS device header file may be used with CMSIS. Note that using these IDs with the embOS functions will work only, when 16 is added to the IDs from the CMSIS device header files.

7.6.1.2 Different interrupt priority values

Using embOS functions, the interrupt priority value ranges from 0 to 255 and is written into the NVIC control registers as is, regardless the number of priority bits. 255 is the lowest priority, 0 is the highest priority. Using CMSIS, the range of interrupt priority levels used to setup the interrupt controller depends on the number of priority bits implemented in the specific CPU. The number of priority bits for the specific device shall be defined in the device specific CMSIS header file as `__NVIC_PRIO_BITS`. If it is not defined in the device specific header files, a default of 4 is set in the generic CMSIS core header file. A CPU with 4 priority bits supports up to 16 preemption levels. With CMSIS, the range of interrupt priorities for this CPU would be 0 to 15, where 0 is the highest priority and 15 is the lowest. To convert an embOS priority value into a value for the CMSIS functions, the value has to be shifted to the right by (`8 - __NVIC_PRIO_BITS`). To convert an CMSIS value for the interrupt priority into the value used with the embOS functions, the value has to be shifted to the left by (`8 - __NVIC_PRIO_BITS`). In any case, half of the priorities with lower values (from zero) are high priorities which must not be used with any interrupt handler using embOS functions.

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. Note that the CMSIS functions use different peripheral ID indices to address the specific interrupt number. embOS counts from 0 for the first entry in the interrupt vector table, CMSIS counts from 0 for the first peripheral interrupt vector, which is ID number 16 for the embOS functions. About these differences, also read chapter 7.7.1 To enable and disable interrupts in general, the embOS functions OS_INT_IncDI() and OS_INT_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. Note that with the CMSIS function, the range of valid interrupt priority values depends on the number of priority bits defined and implemented for the specific device. The number of priority bits for the specific device shall be defined in the device specific CMSIS header file as __NVIC_PRIO_BITS. If it is not defined in the device specific header files, a default of 4 is set in the generic CMSIS core header file. A CPU with 4 priority bits supports up to 16 preemption levels. With CMSIS, the range of interrupt priorities for this CPU would be 0 to 15, where 0 is the highest priority and 15 is the lowest. About interrupt priorities in an embOS project, read chapter 6.5 and 6.6., about the differences between interrupt priority and ID values used to setup the NVIC controller, read chapter 7.7.1

Chapter 8

RTT and SystemView

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

8.1 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*.

8.1.1 Shipped files related to SEGGER RTT

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

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

8.2 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.

8.2.1 Shipped files related to SEGGER SystemView

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

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

Chapter 9

Technical data

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

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 minimum ROM requirement for the kernel itself is about 1.700 bytes.

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

embOS resource	RAM [bytes]
Task control block	36
Software timer	20
Mutex	16
Semaphore	8
Mailbox	24
Queue	32
Task event	0
Event object	12

