embOS

Real-Time Operating System

CPU & Compiler specifics
for ARM Aarch64 using GCC

Document: UM01073
Software Version: 5.10.0.0
Revision: 0
Date: June 5, 2020

A product of SEGGER Microcontroller GmbH

www.segger.com

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel.	+49 2173-99312-0
Fax.	+49 2173-99312-28
E-mail:	`support@segger.com`[*]
Internet:	*www.segger.com*

[*]By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at https://www.segger.com/legal/privacy-policy/.

Manual versions

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

Print date: June 5, 2020

Software	Revision	Date	By	Description
5.10.0.0	0	200605	MC	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 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 many prepared sample start projects, which you should use and modify to write your application. So follow the instructions of section *First Steps* on page 10.

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

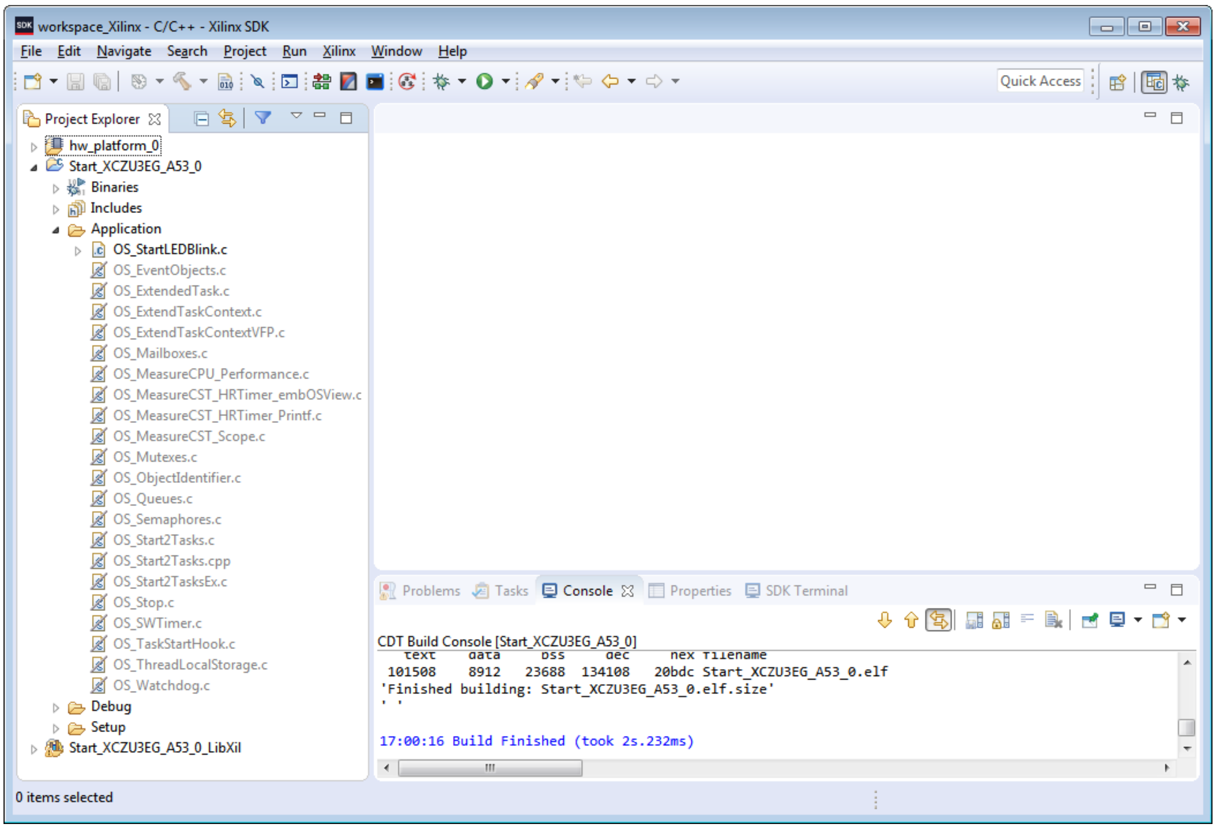
After installation of embOS you can create your first multitasking application. You have received several ready to go sample start workspaces and projects and every other files needed in the subfolder `Start`. It is a good idea to use one of them as a starting point for all of your applications. The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



CDT Build Console [Start_XCZU3EG_A53_0]

```
   text    data     bss     dec     hex filename
 101508    8912   23688  134108   20bdc Start_XCZU3EG_A53_0.elf
'Finished building: Start_XCZU3EG_A53_0.elf.size'
' '

17:00:16 Build Finished (took 2s.232ms)
```

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

1.3 The example application OS_StartLEDBlink.c

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

What happens is easy to see:

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

```
/*********************************************************************
*                     SEGGER Microcontroller GmbH                    *
*                        The Embedded Experts                        *
**********************************************************************

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

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

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

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

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

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

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

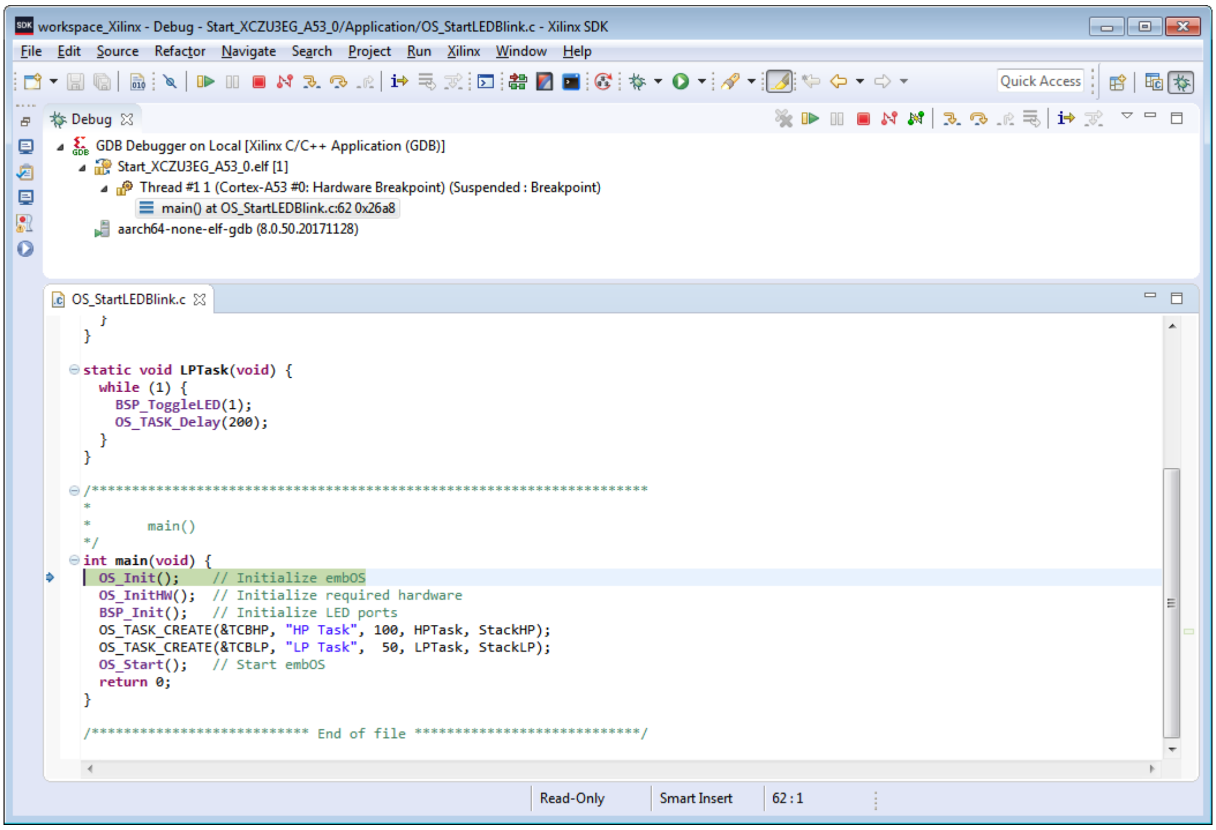
1.4 Stepping through the sample application

When starting the debugger, you will see the `main()` function (see example 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.

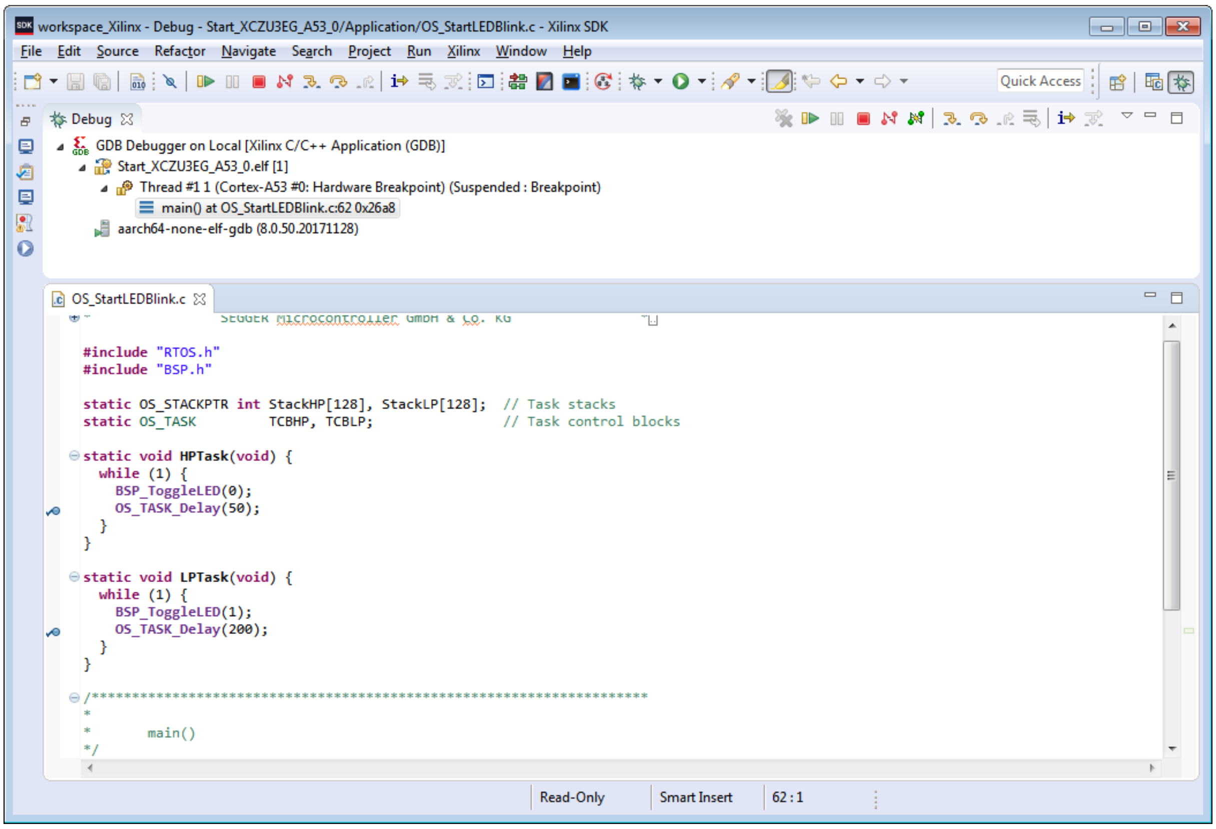
`OS_Start()` should be the last line in `main()`, because it starts multitasking and does not return.


workspace_Xilinx - Debug - Start_XCZU3EG_A53_0/Application/OS_StartLEDBlink.c - Xilinx SDK
File Edit Source Refactor Navigate Search Project Run Xilinx Window Help
Quick Access
Debug
OS_StartLEDBlink.c
Read-Only
Smart Insert
62 : 1

```
GDB Debugger on Local [Xilinx C/C++ Application (GDB)]
  Start_XCZU3EG_A53_0.elf [1]
    Thread #1 1 (Cortex-A53 #0: Hardware Breakpoint) (Suspended : Breakpoint)
      main() at OS_StartLEDBlink.c:62 0x26a8
  aarch64-none-elf-gdb (8.0.50.20171128)
```

```
  }
}

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

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


workspace_Xilinx - Debug - Start_XCZU3EG_A53_0/Application/OS_StartLEDBlink.c - Xilinx SDK
File Edit Source Refactor Navigate Search Project Run Xilinx Window Help
Quick Access
Debug
OS_StartLEDBlink.c

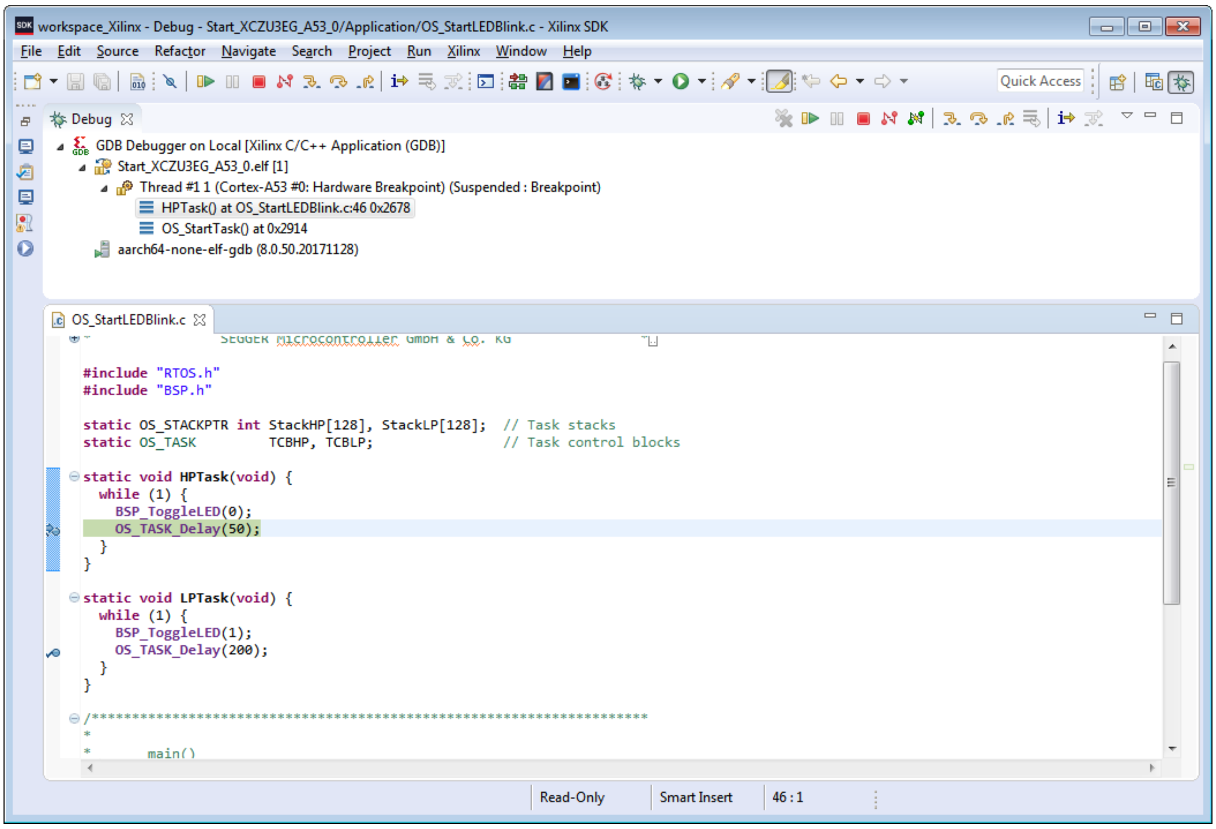
```
GDB Debugger on Local [Xilinx C/C++ Application (GDB)]
  Start_XCZU3EG_A53_0.elf [1]
    Thread #1 1 (Cortex-A53 #0: Hardware Breakpoint) (Suspended : Breakpoint)
      main() at OS_StartLEDBlink.c:62 0x26a8
  aarch64-none-elf-gdb (8.0.50.20171128)
```

```
              SEGGER Microcontroller GmbH & Co. KG

#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()
*/
```

Read-Only | Smart Insert | 62 : 1

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.

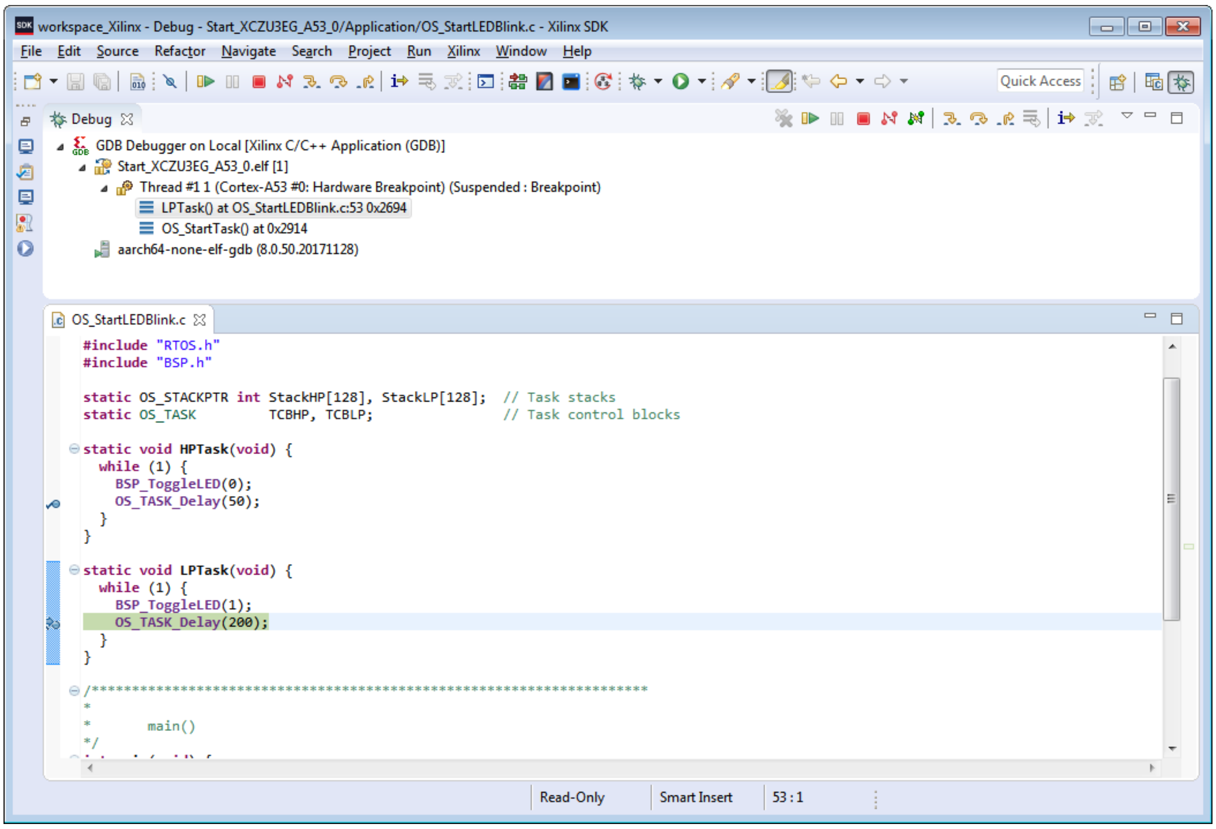

workspace_Xilinx - Debug - Start_XCZU3EG_A53_0/Application/OS_StartLEDBlink.c - Xilinx SDK
File Edit Source Refactor Navigate Search Project Run Xilinx Window Help
Quick Access
Debug
OS_StartLEDBlink.c

```
GDB Debugger on Local [Xilinx C/C++ Application (GDB)]
  Start_XCZU3EG_A53_0.elf [1]
    Thread #1 1 (Cortex-A53 #0: Hardware Breakpoint) (Suspended : Breakpoint)
      HPTask() at OS_StartLEDBlink.c:46 0x2678
      OS_StartTask() at 0x2914
  aarch64-none-elf-gdb (8.0.50.20171128)
```

```
              SEGGER Microcontroller GmbH & Co. KG

#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()
```

Read-Only | Smart Insert | 46 : 1

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


File Edit Source Refactor Navigate Search Project Run Xilinx Window Help
Quick Access
Debug
OS_StartLEDBlink.c

workspace_Xilinx - Debug - Start_XCZU3EG_A53_0/Application/OS_StartLEDBlink.c - Xilinx SDK

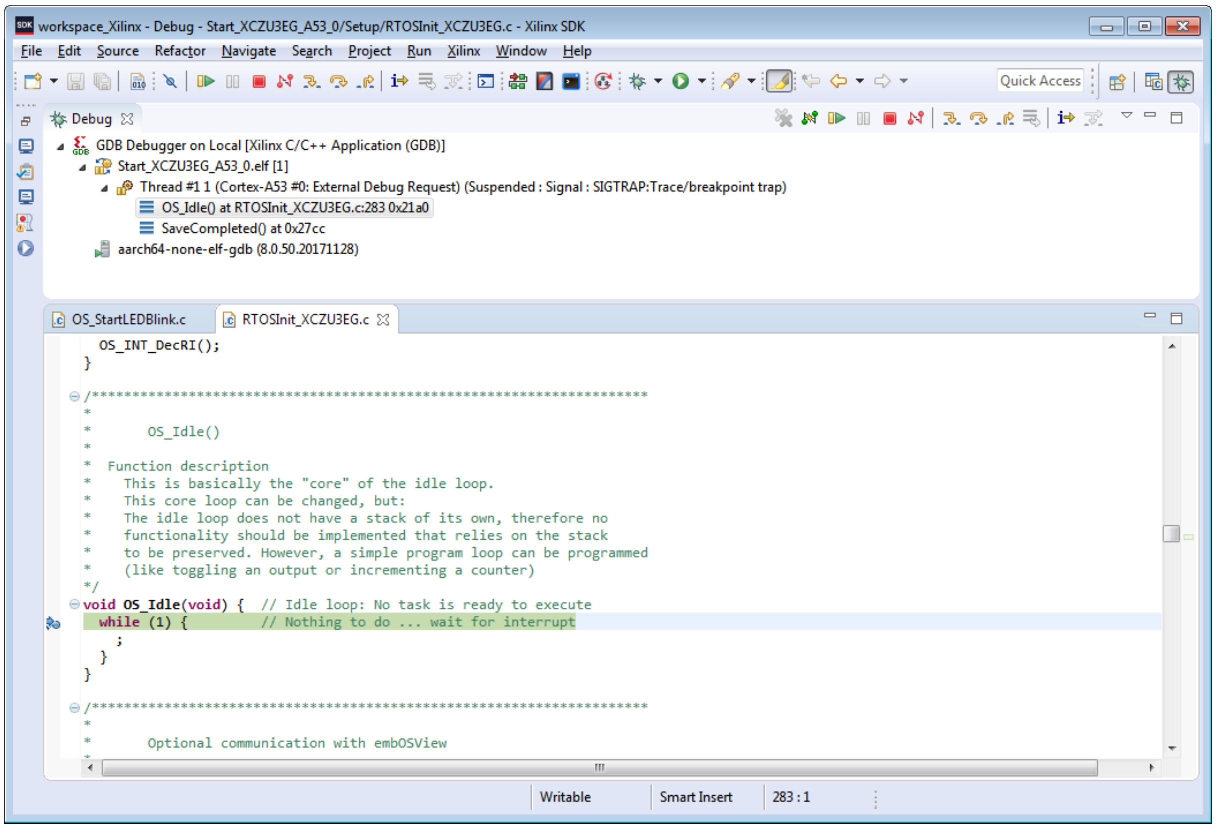
```
GDB Debugger on Local [Xilinx C/C++ Application (GDB)]
  Start_XCZU3EG_A53_0.elf [1]
    Thread #1 1 (Cortex-A53 #0: Hardware Breakpoint) (Suspended : Breakpoint)
      LPTask() at OS_StartLEDBlink.c:53 0x2694
      OS_StartTask() at 0x2914
  aarch64-none-elf-gdb (8.0.50.20171128)
```

```
#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()
*/
```

Read-Only | Smart Insert | 53 : 1

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()`.


File Edit Source Refactor Navigate Search Project Run Xilinx Window Help
Quick Access
Debug
OS_StartLEDBlink.c
RTOSInit_XCZU3EG.c

workspace_Xilinx - Debug - Start_XCZU3EG_A53_0/Setup/RTOSInit_XCZU3EG.c - Xilinx SDK

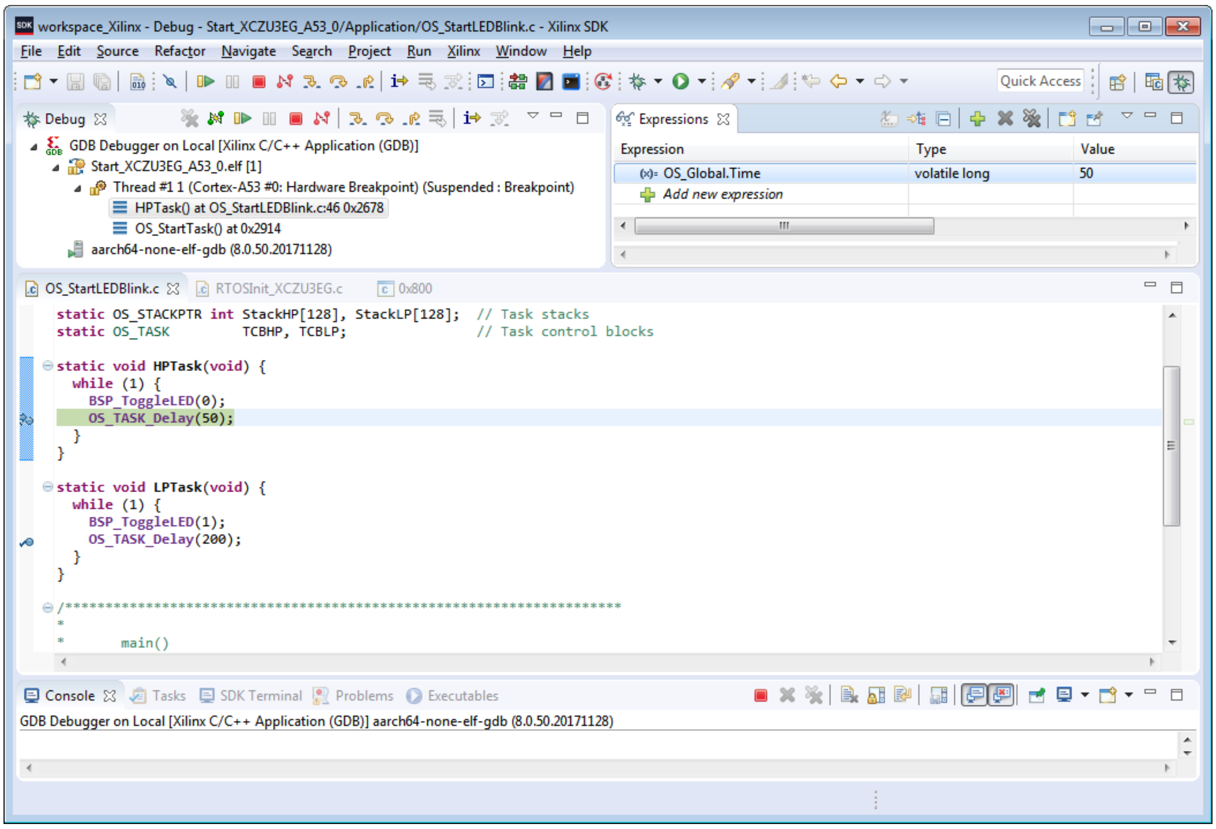
```
GDB Debugger on Local [Xilinx C/C++ Application (GDB)]
  Start_XCZU3EG_A53_0.elf [1]
    Thread #1 1 (Cortex-A53 #0: External Debug Request) (Suspended : Signal : SIGTRAP:Trace/breakpoint trap)
      OS_Idle() at RTOSInit_XCZU3EG.c:283 0x21a0
      SaveCompleted() at 0x27cc
  aarch64-none-elf-gdb (8.0.50.20171128)
```

```
  OS_INT_DecRI();
}

/*********************************************************************
*
*       OS_Idle()
*
*  Function description
*    This is basically the "core" of the idle loop.
*    This core loop can be changed, but:
*    The idle loop does not have a stack of its own, therefore no
*    functionality should be implemented that relies on the stack
*    to be preserved. However, a simple program loop can be programmed
*    (like toggling an output or incrementing a counter)
*/
void OS_Idle(void) {  // Idle loop: No task is ready to execute
  while (1) {         // Nothing to do ... wait for interrupt
    ;
  }
}

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

Writable | Smart Insert | 283 : 1

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.


workspace_Xilinx - Debug - Start_XCZU3EG_A53_0/Application/OS_StartLEDBlink.c - Xilinx SDK
File Edit Source Refactor Navigate Search Project Run Xilinx Window Help
Quick Access
Debug
Expressions
OS_StartLEDBlink.c
RTOSInit_XCZU3EG.c
0x800
Console
Tasks
SDK Terminal
Problems
Executables

```
GDB Debugger on Local [Xilinx C/C++ Application (GDB)]
  Start_XCZU3EG_A53_0.elf [1]
    Thread #1 1 (Cortex-A53 #0: Hardware Breakpoint) (Suspended : Breakpoint)
      HPTask() at OS_StartLEDBlink.c:46 0x2678
      OS_StartTask() at 0x2914
  aarch64-none-elf-gdb (8.0.50.20171128)
```

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

```
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()
```

```
GDB Debugger on Local [Xilinx C/C++ Application (GDB)] aarch64-none-elf-gdb (8.0.50.20171128)
```

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 10 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<Architecture><ExecState><Endianness><Libmode>.a`

Parameter	Meaning	Values
`Architecture`	Specifies the ARM architecture	A8 : Always ARMv8-A
`ExecState`	Specifies the execution state	A : Always Aarch64
`Endianness`	Byte order	B : Big endian L : Little endian
`Libmode`	Specifies the library mode	XR : Extreme Release R : Release S : Stack check SP : Stack check + profiling D : Debug DP : Debug + profiling + stack check DT : Debug + profiling + Stack check + trace

Example

`libosA8ALDP.a` is the library for an ARMv8-A core, Aarch64 execution state, little endian mode, with debug and profiling support.

Chapter 4

CPU and compiler specifics

4.1 Thread-safe system libraries

embOS ARM64 GCC 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.

After using this function, any further task context extensions cannot be added by calling `OS_TASK_SetContextExtension()`, but can be added using `OS_TASK_AddContextExtension()` instead.

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.

Chapter 5

Stacks

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

5.1 Task stack

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 any size used by exceptions. The basic stack size is the size of memory required to store the CPU registers.

The minimum basic task stack size is about 288 bytes. Because any function call uses some amount of stack and every exception also pushes at least 256 bytes onto the current stack, the task stack size has to be large enough to handle these, too. We recommend at least 640 bytes stack as a start.

5.2 System stack

embOS executes in exception level 3 and uses `__el3_stack` as system stack. The minimum system stack size required by embOS is about 512 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 exceptions 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 stack size define in your linker file. We recommend a minimum stack size of 768 bytes for the `__el3_stack`.

5.3 Stack specifics

The system stack has to be declared in the linker script file:

- `__el3_stack` is the EL3 system stack.

The `__el3_stack` is used during startup, during `main()` and `OS_Idle()`, as well as embOS internal functions. Tasks are executed in exception level 3 as well, but they do not use the `__el3_stack`. Tasks utilize a dedicated stack instead, which typically is defined in the application as a variable in any RAM location.

Chapter 6

Interrupts

6.1 What happens when an interrupt occurs?

- The CPU-core receives an IRQ.
- As soon as the IRQs in PSTATE are enabled, the interrupt is executed.
- The CPU saves PC and PSTATE into the registers ELR_EL3 and SPSR_EL3. IRQ, FIQ, as well as SError exceptions are disabled in PSTATE.
- The CPU jumps to VBAR_EL3 + 0x280 (i.e. offset 0x280 in the vector table, which is implemented in RTOSVect.S), thus fetches the embOS_IRQHandler() and executes it.
- embOS_IRQHandler(): save core registers.
- embOS_IRQHandler(): save ELR_EL3 and SPSR_EL3.
- embOS_IRQHandler(): call OS_irq_handler() (implemented in RTOSInit_*.c).
- OS_irq_handler(): inform embOS that interrupt code is running by a call to OS_INT_Enter().
- OS_irq_handler(): check for interrupt source and execute appropriate ISR.
- OS_irq_handler(): inform embOS that interrupt handling ended by a call to OS_INT_Leave().
- OS_irq_handler(): return to embOS_IRQHandler().
- embOS_IRQHandler(): restore core registers.
- Return from interrupt, which restores PC from ELR_EL3 and PSTATE from SPSR_EL3 (re-enabling IRQ, FIQ, as well as SError if appropriate).

Please ensure that embOS_IRQHandler() is called for IRQs and embOS_IRQHandler() for synchronous exceptions (each on exception level 3 using the current stack pointer).

6.2 Defining interrupt handlers in C

The low-level interrupt handler embOS_IRQHandler calls the high-level interrupt handler OS_irq_handler() in RTOSInit*.c. That handler first calls OS_INT_Enter() to inform embOS that interrupt code is running and, subsequently, examines the source of the interrupt in order to call an appropriate, user-defined interrupt handler function.

These handler functions must be implemented as regular C-functions which do not take parameters and do not return any value. Depending on the interrupting source, it may also be required to reset the interrupt pending condition of any related peripheral in these handler functions.

After returning from the appropriate interrupt handler function, the high-level interrupt handler OS_irq_handler() in RTOSInit*.c calls OS_INT_Leave() and returns to the low-level interrupt handler embOS_IRQHandler().

Example

A simple, user-defined interrupt handler function could therefore be implemented as follows:

```
void Timer_irq_func(void) {
  if (__INTPND & 0x0800) {   // Interrupt pending ?
    __INTPND = 0x0800;       // reset pending condition
    DoSomething();           // handle interrupt
  }
}
```

6.3 Interrupt handling with embOS

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

Function	Description
`OS_ARM_InstallISRHandler()`	Installs an interrupt handler
`OS_ARM_EnableISR()`	Enables a specific interrupt
`OS_ARM_DisableISR()`	Disables a specific interrupt
`OS_ARM_ISRSetPrio()`	Sets the priority of a specific interrupt
`OS_ARM_ClearPendingFlag()`	Clears an interrupt pending flag
`OS_ARM_IsPending()`	Checks if an interrupt is pending

6.3.1 OS_ARM_InstallISRHandler()

Description

`OS_ARM_InstallISRHandler()` is used to install a specific interrupt handler.

Prototype

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

Parameters

Parameter	Description
`ISRIndex`	Index of the interrupt source.
`pISRHandler`	Address of the interrupt handler function.

Return Value

`OS_ISR_HANDLER*`: The address of the interrupt handler that was previously installed at the addressed interrupt source.

Additional Information

This function just installs the interrupt handler, but neither modifies the priority nor automatically enables the interrupt.

6.3.2 OS_ARM_EnableISR()

Description

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

Prototype

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

Parameters

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

Additional Information

This function just enables the interrupt inside the interrupt controller. It does not enable the interrupt in any peripheral.

6.3.3 OS_ARM_DisableISR()

Description

`OS_ARM_DisableISR()` is used to disable interrupt acceptance of a specific interrupt source.

Prototype

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

Parameters

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

Additional Information

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

6.3.4 OS_ARM_ISRSetPrio()

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.
`Prio`	The priority which should be set for the specific interrupt.

Return Value

Previously assigned priority for the addressed interrupt source.

Additional Information

This function sets the priority of an interrupt channel by programming the interrupt controller. Refer to CPU-specific manuals about allowed priority levels.

Example

```
// Install UART interrupt handler
OS_ARM_InstallISRHandler(UART_ID, &COM_ISR);          // UART interrupt vector
OS_ARM_ISRSetPrio(UART_ID, UART_PRIO);                // UART interrupt priotity
OS_ARM_EnableISR(UART_ID);                            // Enable UART interrupt
```

6.3.5 OS_ARM_ClearPendingFlag()

Description

`OS_ARM_ClearPendingFlag()` is used to clear an interrupt pending flag

Prototype

```
void OS_ARM_ClearPendingFlag(int ISRIndex);
```

Parameters

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

Additional Information

This function just clears the interrupt pending flag inside the interrupt controller. It does not clear the interrupt pending flag in any peripheral.

6.3.6 OS_ARM_IsPending()

Description

`OS_ARM_IsPending()` is used to check if an interrupt is pending

Prototype

```
unsigned int OS_ARM_IsPending(int ISRIndex);
```

Parameters

Parameter	Description
`ISRIndex`	Index of the interrupt source which should be checked

Return value

= 0	Interrupt is not pending.
=! 0	Interrupt is pending.

Additional Information

This function just checks the interrupt pending flag inside the interrupt controller. It does not check the interrupt pending flag in any peripheral.

6.4 Fast Interrupt (FIQ)

FIQ cannot be used with embOS functions, but is reserved for high speed user functions.

Note the following:

- FIQ is never disabled by embOS.
- Never call any embOS function from an FIQ handler.
- Do not assign any embOS interrupt handler to FIQ.

Note

When you decide to use FIQ, ensure that an interrupt vector for FIQ handling is included in your application.

Chapter 7

VFP and NEON support

7.1 Vector Floating Point and NEON support

Some ARM MCUs come with integrated vectored floating point unit VFP and NEON unit. When activating the VFP or NEON support in the project options, the compiler and linker will add efficient code which uses the VFP when floating point operations are used or NEON instructions where possible in the application.

With embOS, the VFP and NEON registers have to be saved and restored when task switches are performed. For efficiency reasons, embOS does not save and restore the VFP and NEON registers for every task automatically. The context switching time and stack load are therefore not affected when the VFP/NEON unit is not used or needed. Saving and restoring the VFP/NEON registers can be enabled for every task individually by extending the task context of the tasks, where VFP or NEON is used.

7.1.1 OS_TASK_SetContextExtensionVFP()

Description

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

Prototype

```
void OS_TASK_SetContextExtensionVFP(void);
```

Additional information

`OS_TASK_SetContextExtensionVFP()` extends the task context to save and restore the VFP registers during context switches. There is no need to extend the task context for every task. Only those tasks using the VFP for calculation have to be extended.

After using this function, any further task context extensions cannot be added by calling `OS_TASK_SetContextExtension()`, but can be added using `OS_TASK_AddContextExtension()` instead.

7.1.2 OS_TASK_SetContextExtensionNEON()

Description

`OS_TASK_SetContextExtensionNEON()` has to be called as first function in a task, when the NEON unit is used in the task and the NEON regsisters have to be added to the task context.

Prototype

```
void OS_TASK_SetContextExtensionNEON(void);
```

Additional information

`OS_TASK_SetContextExtensionNEON()` extends the task context to save and restore the NEON registers during context switches. There is no need to extend the task context for every task. Only those tasks using the NEON for calculation have to be extended.

After using this function, any further task context extensions cannot be added by calling `OS_TASK_SetContextExtension()`, but can be added using `OS_TASK_AddContextExtension()` instead.

7.1.3 Using VFP/NEON in interrupt service routines

Using the VFP/NEON in interrupt service routines requires additional functions to save and restore the VFP/NEON registers.

As the compiler might not add additional code to save and restore the VFP/NEON registers on entry and exit of interrupt service routines, it is the users responsibility to save the VFP/NEON registers on entry of an interrupt service routine when the VFP or NEON is used in the ISR.

embOS delivers functions to save and restore the VFP or NEON context in an interrupt service routine.

7.1.3.1 OS_VFP_Save() / OS_NEON_Save()

Description

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

Prototype

```
void OS_VFP_Save (void);

void OS_NEON_Save(void);
```

Additional information

`OS_VFP_Save()` / `OS_NEON_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()`/`OS_NEON_Save()` function, the interrupt service routine may use the VFP or NEON unit for calculation without destroying the saved content of the VFP/NEON registers.

To restore the registers, the ISR has to call `OS_VFP_Restore()`/`OS_NEON_Restore()` at the end.

The function may be used in any ISR regardless the priority. It is not restricted to low priority interrupt functions.

7.1.3.2 OS_VFP_Restore() / OS_NEON_Restore()

Description

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

Prototype

```
void OS_VFP_Restore (void);

void OS_NEON_Restore(void);
```

Additional information

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

Chapter 8

Technical data

This chapter lists technical data of embOS used with ARM CPUs.

8.1 Memory requirements

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

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

embOS resource	RAM [bytes]
Task control block	48
Software timer	32
Mutex	32
Semaphore	16
Mailbox	32
Queue	40
Task event	0
Event object	24