embOS

Real-Time Operating System

CPU & Compiler specifics for S12Z using S12lisa compiler

Document: UM01071
Software Version: 5.02a
Revision: 0
Date: August 21, 2018

SEGGER

A product of SEGGER Microcontroller GmbH

www.segger.com

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: August 21, 2018

Software	Revision	Date	By	Description
5.02a	0	180821	TS	Initial version.

About this document

Assumptions

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

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

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 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

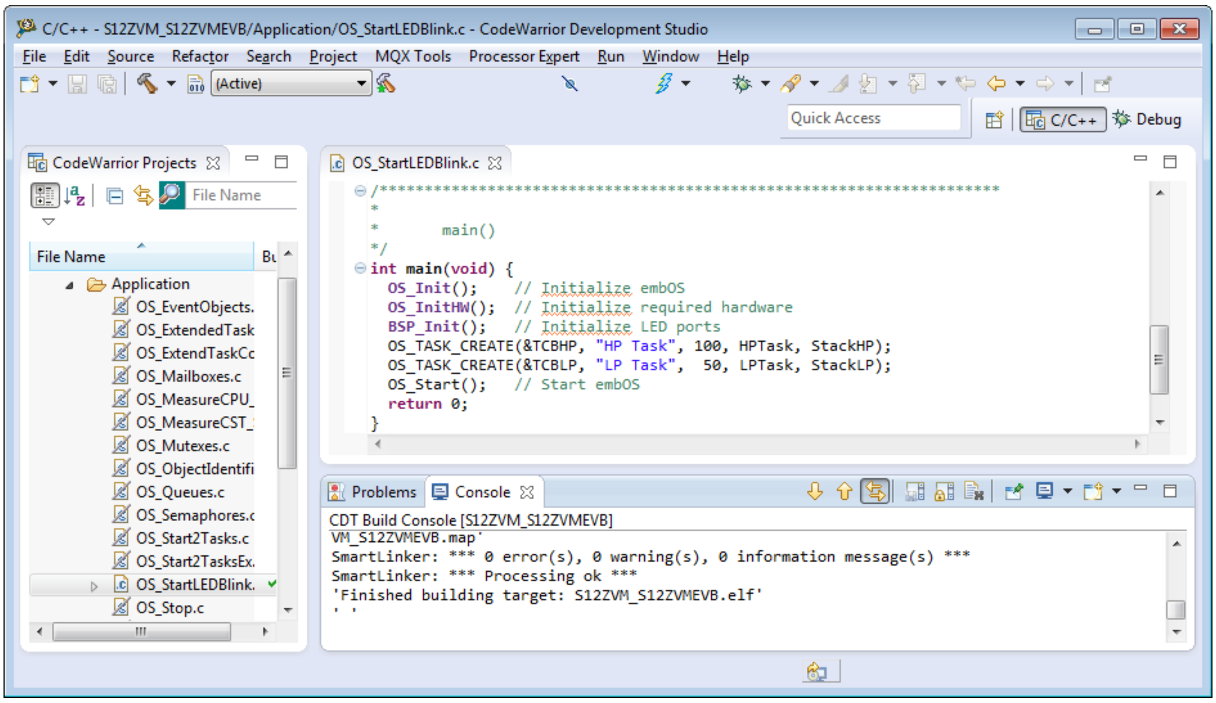
After installation of embOS you can create your first multitasking application. You have received several ready to go sample start workspaces and projects and every other files needed in the subfolder `Start`. It is a good idea to use one of them as a starting point for all of your applications. The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



```
/*********************************************************************
*
*       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;
}
```

CDT Build Console [S12ZVM_S12ZVMEVB]

```
VM_S12ZVMEVB.map'
SmartLinker: *** 0 error(s), 0 warning(s), 0 information message(s) ***
SmartLinker: *** Processing ok ***
'Finished building target: S12ZVM_S12ZVMEVB.elf'
' '
```

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.

```c
/*********************************************************************
*                 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 ****************************/
```

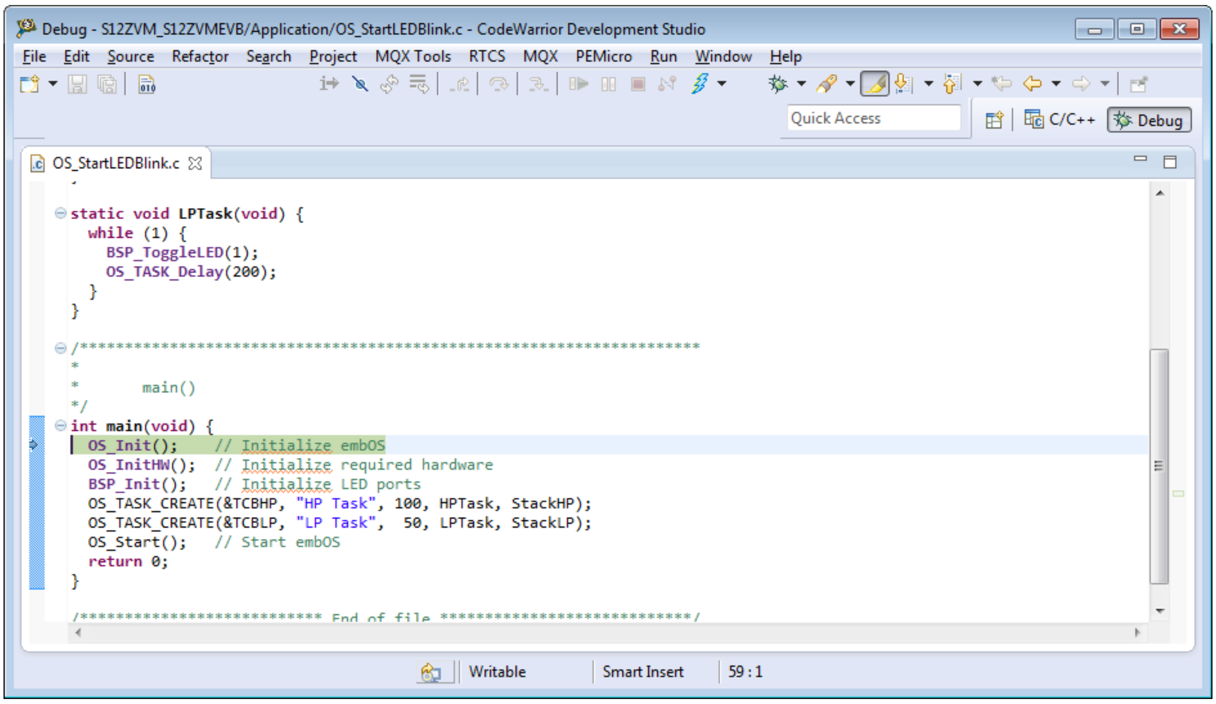
1.4 Stepping through the sample application

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.

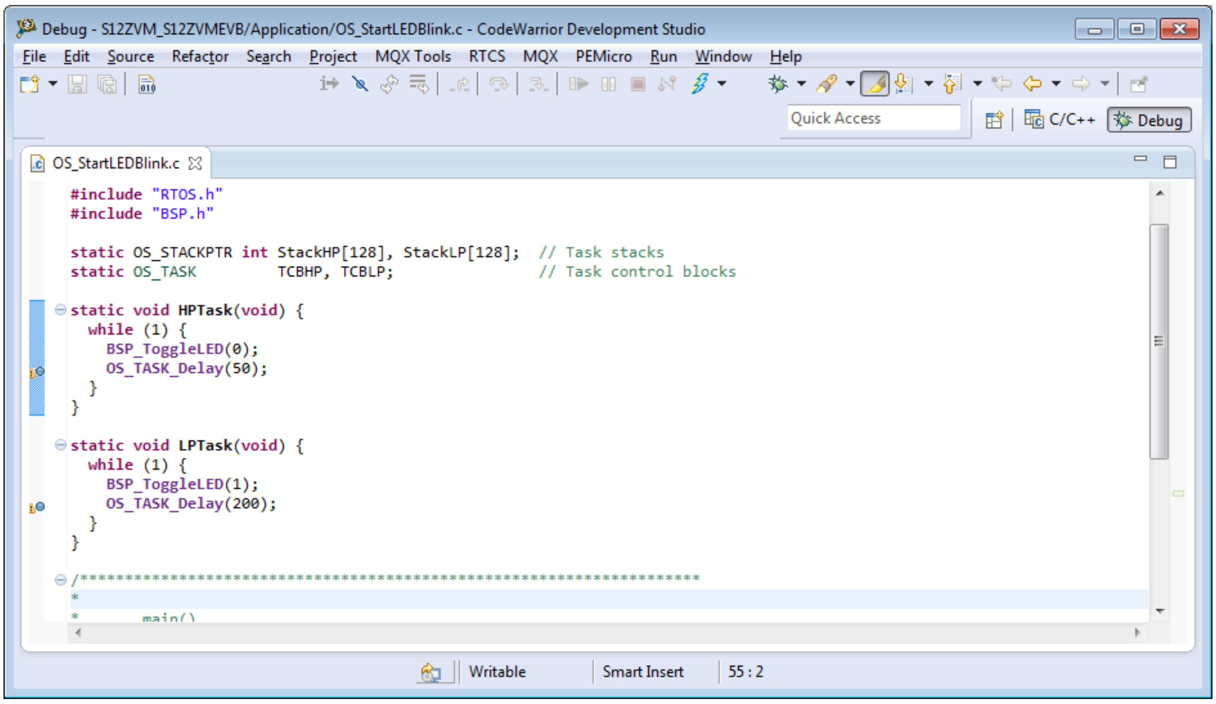
`OS_InitHW()` is part of `RTOSInit.c` and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the system tick interrupt for embOS. Step through it to see what is done.

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


Debug - S12ZVM_S12ZVMEVB/Application/OS_StartLEDBlink.c - CodeWarrior Development Studio
File Edit Source Refactor Search Project MQX Tools RTCS MQX PEMicro Run Window Help
Quick Access
C/C++
Debug
OS_StartLEDBlink.c
Writable
Smart Insert
59 : 1

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

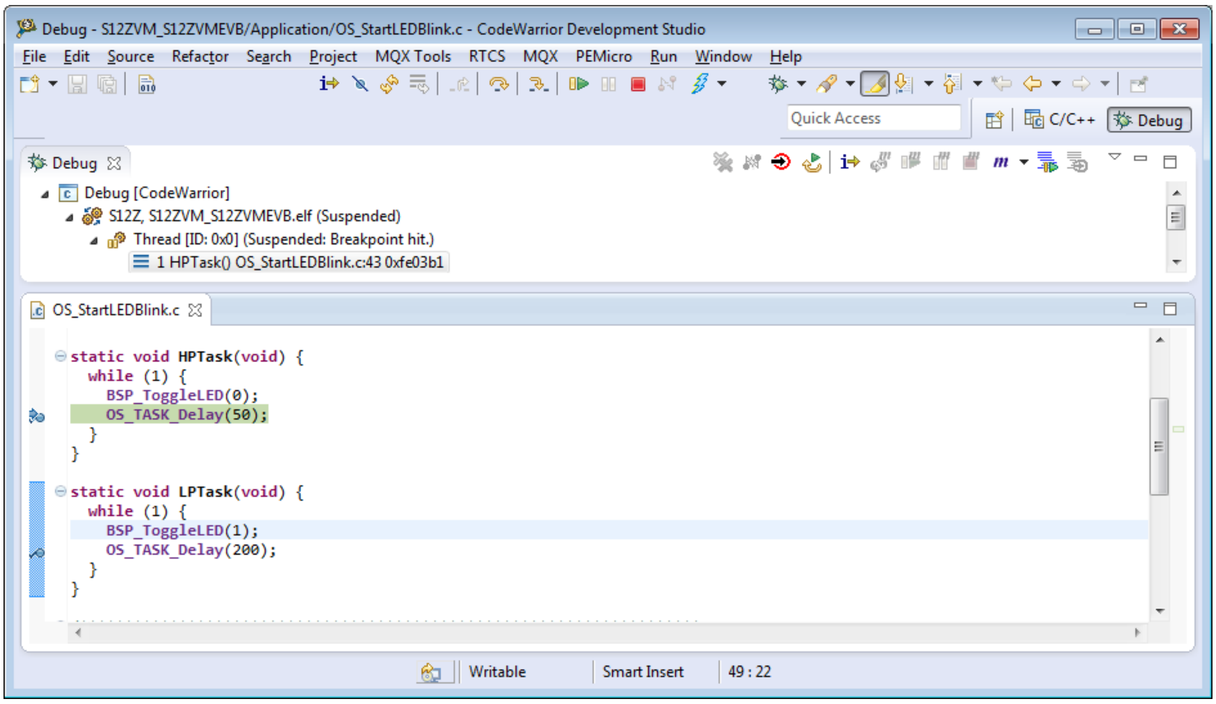
Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.


Debug - S12ZVM_S12ZVMEVB/Application/OS_StartLEDBlink.c - CodeWarrior Development Studio
File Edit Source Refactor Search Project MQX Tools RTCS MQX PEMicro Run Window Help
Quick Access
C/C++
Debug
OS_StartLEDBlink.c
Writable
Smart Insert
55 : 2

```
#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()
```

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.


Debug - S12ZVM_S12ZVMEVB/Application/OS_StartLEDBlink.c - CodeWarrior Development Studio
File Edit Source Refactor Search Project MQX Tools RTCS MQX PEMicro Run Window Help
Quick Access
C/C++
Debug

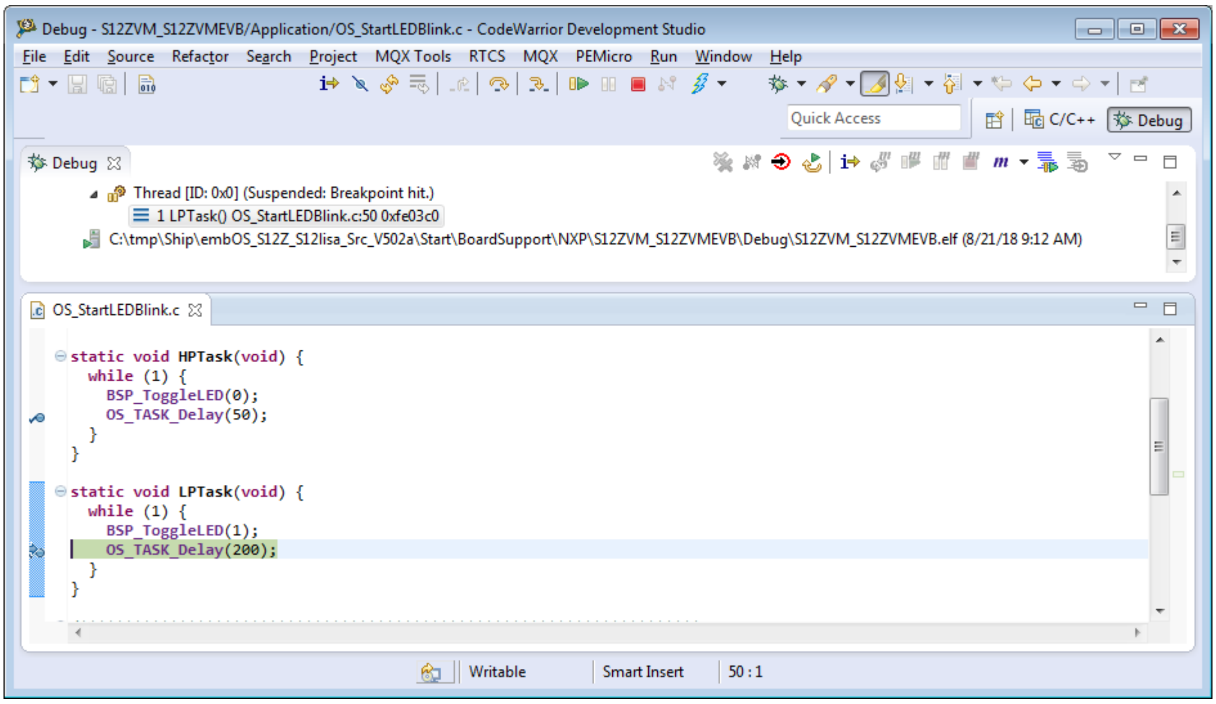
Debug [CodeWarrior]
S12Z, S12ZVM_S12ZVMEVB.elf (Suspended)
Thread [ID: 0x0] (Suspended: Breakpoint hit.)
1 HPTask() OS_StartLEDBlink.c:43 0xfe03b1

OS_StartLEDBlink.c

```
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);
  }
}
```

Writable Smart Insert 49 : 22

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


Debug - S12ZVM_S12ZVMEVB/Application/OS_StartLEDBlink.c - CodeWarrior Development Studio
File Edit Source Refactor Search Project MQX Tools RTCS MQX PEMicro Run Window Help
Quick Access
C/C++
Debug

Thread [ID: 0x0] (Suspended: Breakpoint hit.)
1 LPTask() OS_StartLEDBlink.c:50 0xfe03c0
C:\tmp\Ship\embOS_S12Z_S12lisa_Src_V502a\Start\BoardSupport\NXP\S12ZVM_S12ZVMEVB\Debug\S12ZVM_S12ZVMEVB.elf (8/21/18 9:12 AM)

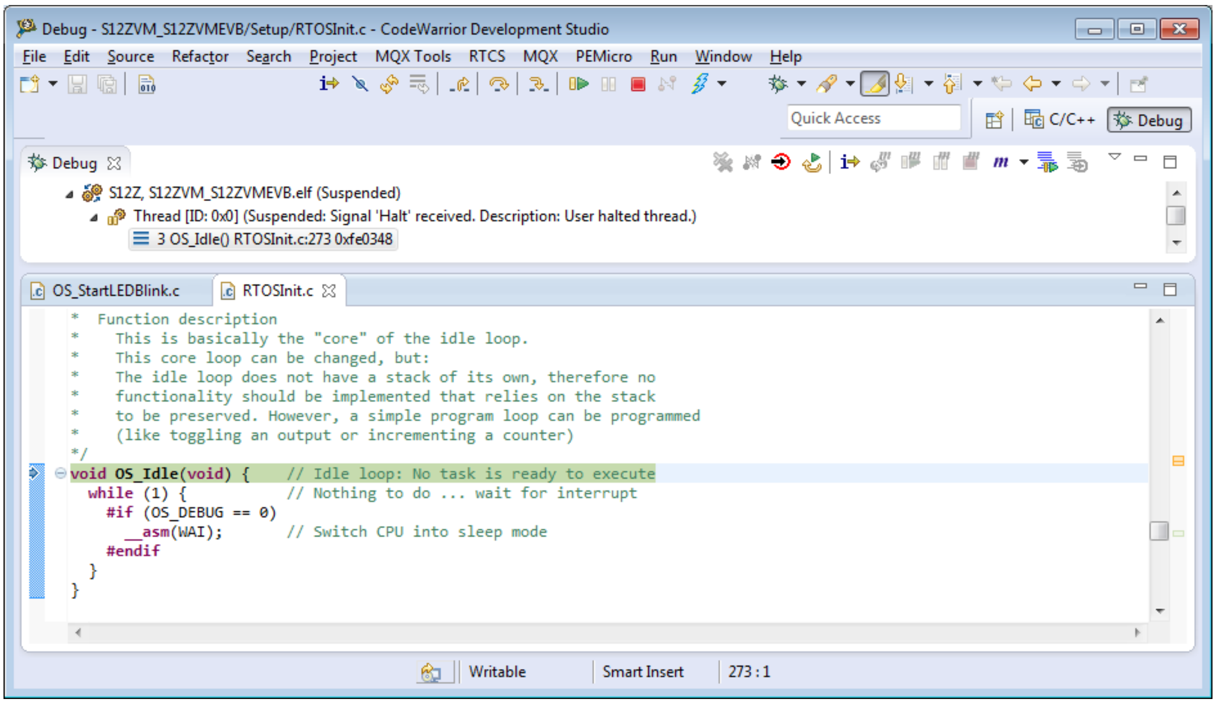
OS_StartLEDBlink.c

```
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);
  }
}
```

Writable Smart Insert 50 : 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()`.

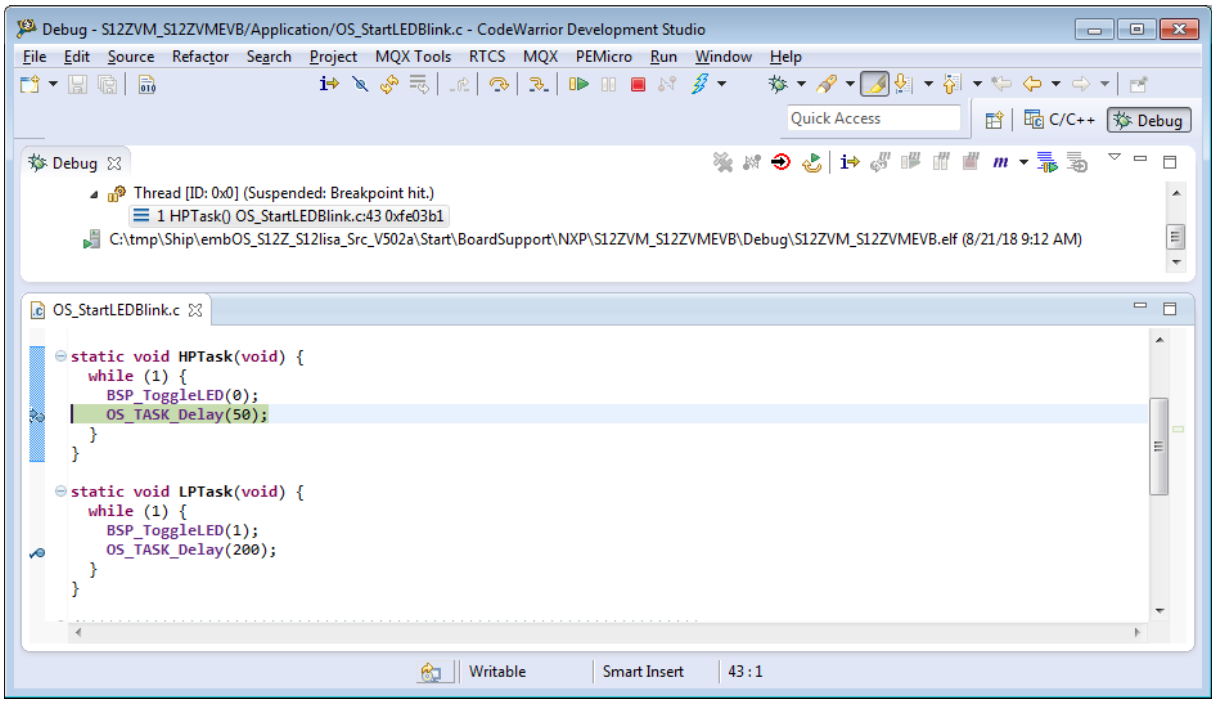

Debug - S12ZVM_S12ZVMEVB/Setup/RTOSInit.c - CodeWarrior Development Studio
File Edit Source Refactor Search Project MQX Tools RTCS MQX PEMicro Run Window Help
Quick Access
C/C++
Debug
Debug
OS_StartLEDBlink.c
RTOSInit.c
Writable
Smart Insert
273 : 1

```
S12Z, S12ZVM_S12ZVMEVB.elf (Suspended)
  Thread [ID: 0x0] (Suspended: Signal 'Halt' received. Description: User halted thread.)
    3 OS_Idle() RTOSInit.c:273 0xfe0348
```

```
*  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
    #if (OS_DEBUG == 0)
      __asm(WAI);        // Switch CPU into sleep mode
    #endif
  }
}
```

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.


Debug - S12ZVM_S12ZVMEVB/Application/OS_StartLEDBlink.c - CodeWarrior Development Studio
File Edit Source Refactor Search Project MQX Tools RTCS MQX PEMicro Run Window Help
Quick Access
C/C++
Debug
Debug
OS_StartLEDBlink.c
Writable
Smart Insert
43 : 1

```
  Thread [ID: 0x0] (Suspended: Breakpoint hit.)
    1 HPTask() OS_StartLEDBlink.c:43 0xfe03b1
C:\tmp\Ship\embOS_S12Z_S12lisa_Src_V502a\Start\BoardSupport\NXP\S12ZVM_S12ZVMEVB\Debug\S12ZVM_S12ZVMEVB.elf (8/21/18 9:12 AM)
```

```
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);
  }
}
```

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:

`os<Core><MemModel>_<Libmode>.lib`

Parameter	Meaning	Values
`Core`	Specifies the core	s12z: S12Z core
`MemModel`	Memory Model	s : small m : medium l : large
`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

`oss12zl_DP.lib` is the library for a project using S12Z core, large memory model with debug and profiling support.

Chapter 4

CPU and compiler specifics

4.1 Standard system libraries

embOS for S12Z and S12lisa compiler may be used with standard system libraries for most of all projects. Heap management and file operation functions of standard system libraries are not reentrant and can therefore not be used with embOS, if non thread safe functions are used from different tasks. For heap management, embOS delivers its own thread safe functions which may be used. These functions are described in embOS CPU independent manual.

Chapter 5

Stacks

This chapter describes how embOS uses the different stacks of the S12Z 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 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 S12Z core, the minimum basic task stack size is about 64 bytes. Because any function call uses some amount of stack and every exception also pushes some bytes onto the current stack, the task stack size has to be large enough to handle exceptions too. We recommend at least 256 bytes stack as a start.

5.2 System stack

The minimum system stack size required by embOS is about 128 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 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 STACKSIZE define in your *.prm linker file. We recommend a minimum stack size of 256 bytes for the system stack.

5.3 Interrupt stack

The S12Z core has no separate interrupt stack pointer. Interrupts are executed on the current stack which could be task stack or system stack.

Chapter 6

Interrupts

6.1 What happens when an interrupt occurs?

- The CPU-core receives an interrupt request form the interrupt controller.
- As soon as the interrupts are enabled, the interrupt is accepted and executed.
- The CPU pushes temporary registers and the return address onto the current stack.
- The CPU jumps to the vector address from the vector table
- The interrupt handler is processed.
- The interrupt handler ends with a return from interrupt
- 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 S12Z core 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 interrupt void SysTick_Handler(void) {
  OS_INT_Enter();    // Inform embOS that interrupt code is running
  OS_TICK_Handle();
  OS_INT_Leave();    // Inform embOS that interrupt handler is left
}
```

6.3 Interrupt vector table

The interrupt vector table is located in a C source file. All interrupt handler function addresses have to be inserted in the vector table.

6.4 Interrupt-stack switching

Interrupt stack switching is currently not supported. Interrupts run on the current stack which could be the CSTACK or a task stack.

6.5 Zero latency interrupts

Zero interrupt latency is supported with embOS for S12Z. Instead of disabling interrupts when embOS does atomic operations, the interrupt level of the CPU is set to 4. Therefore all interrupt priorities higher than 4 can still be processed. You must not execute any embOS function from within a zero latency interrupt function.

6.5.1 OS_INT_SetPriorityThreshold()

The interrupt priority limit for fast interrupts is set to 4 by default. This means, all interrupts with higher priority from 4 up to the maximum CPU specific priority will never be disabled by embOS. Description `OS_INT_SetPriorityThreshold()` is used to set the interrupt priority limit between zero latency interrupts and lower priority embOS interrupts.

Prototype

void `OS_INT_SetPriorityThreshold(unsigned` int Priority)

Parameter

Parameter	Description
`Priority`	The highest value useable as priority for embOS interrupts. All interrupts with higher priority are never disabled by embOS. Valid range: 1 ≤ Priority ≤ 7

Additional information

To disable zero latency interrupts at all, the priority limit may be set to the highest interrupt priority supported by the CPU, which is 7. To modify the default priority limit, `OS_INT_SetPriorityThreshold()` should be called before embOS was started. In the default projects, `OS_INT_SetPriorityThreshold()` is not called. The start projects use the default zero latency interrupt priority limit. Any interrupts running above the zero latency interrupt priority limit must not call any embOS function.

Chapter 7

Technical data

This chapter lists technical data of embOS used with S12Z CPUs.

7.1 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	16
Software timer	11
Mutex	10
Semaphore	5
Mailbox	15
Queue	17
Task event	0
Event object	6