embOS

Real-Time Operating System

CPU & Compiler specifics for ColdFire using CodeWarrior MCU

Document: UM010074
Software Version: 5.10.2.0
Revision: 0
Date: September 2, 2020

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

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

Copyright notice

Trademarks

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

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

Contact address

SEGGER Microcontroller GmbH

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: September 2, 2020

Software	Revision	Date	By	Description
5.10.2.0	0	200902	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 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

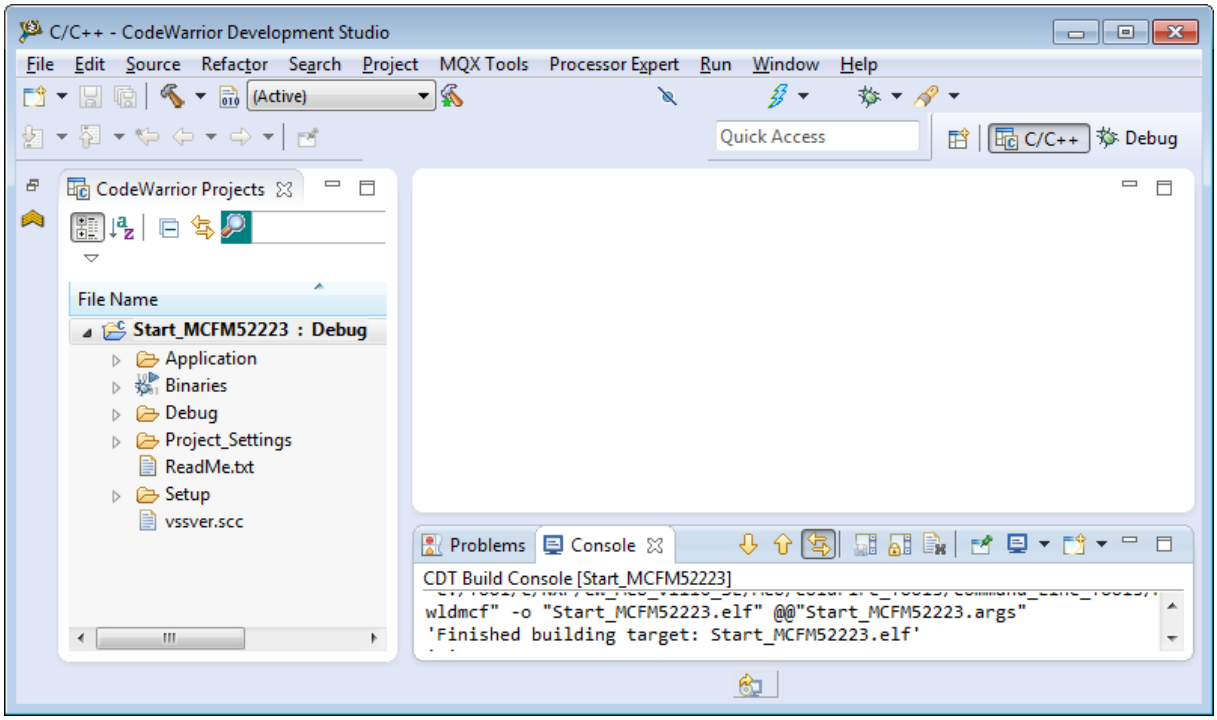
After installation of embOS you can create your first multitasking application. You have received several ready to go sample start workspaces and projects and every other files needed in the subfolder `Start`. It is a good idea to use one of them as a starting point for all of your applications. The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



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

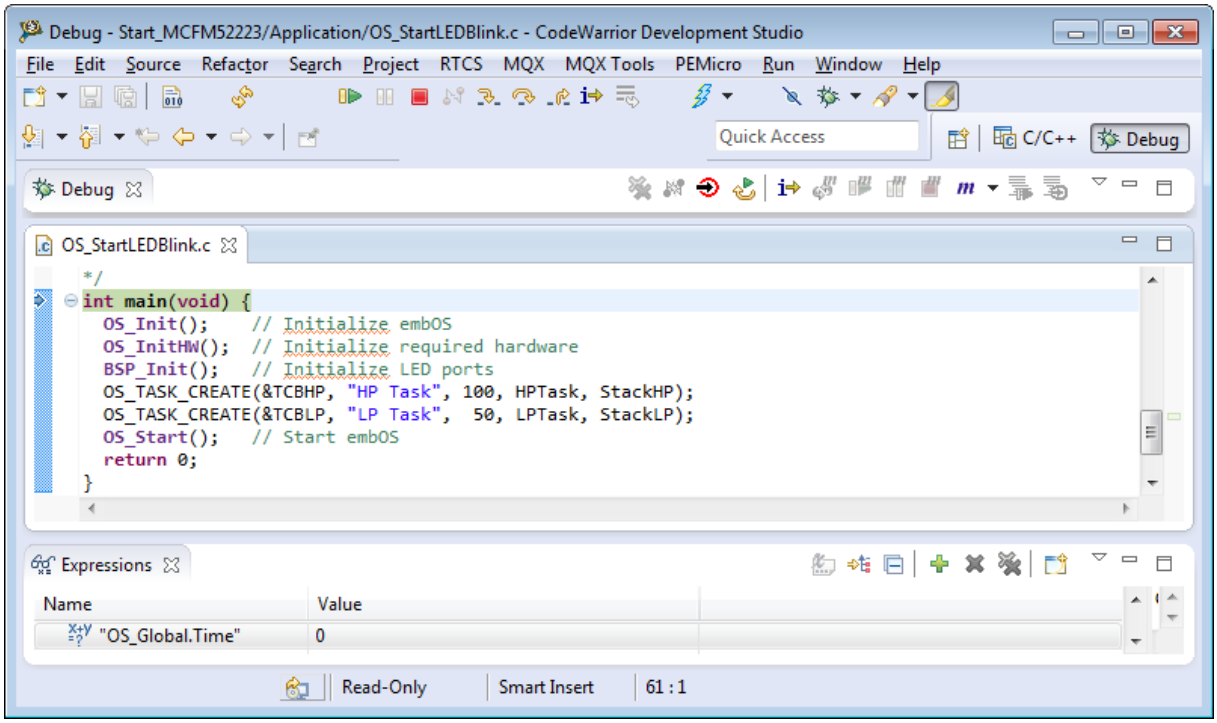

1.4 Stepping through the sample application

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.

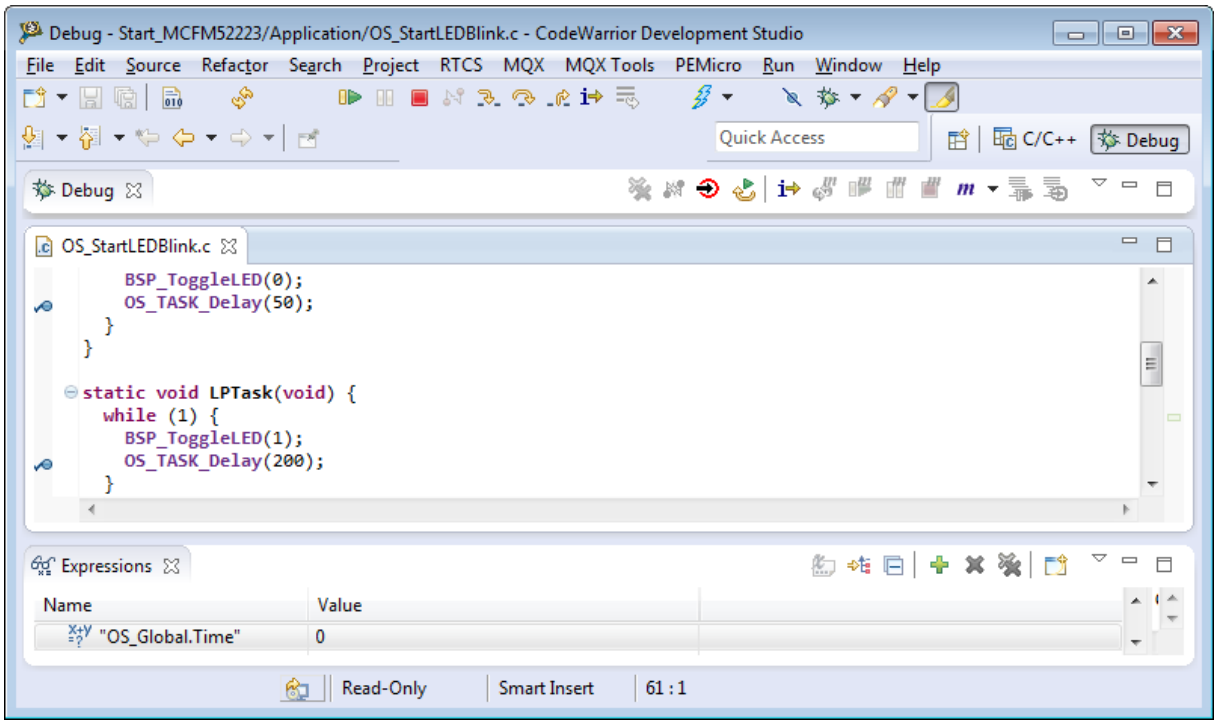
`OS_Init()` is part of the embOS library and written in assembler; you can there fore only step into it in disassembly mode. It initializes the relevant OS variables.

`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.

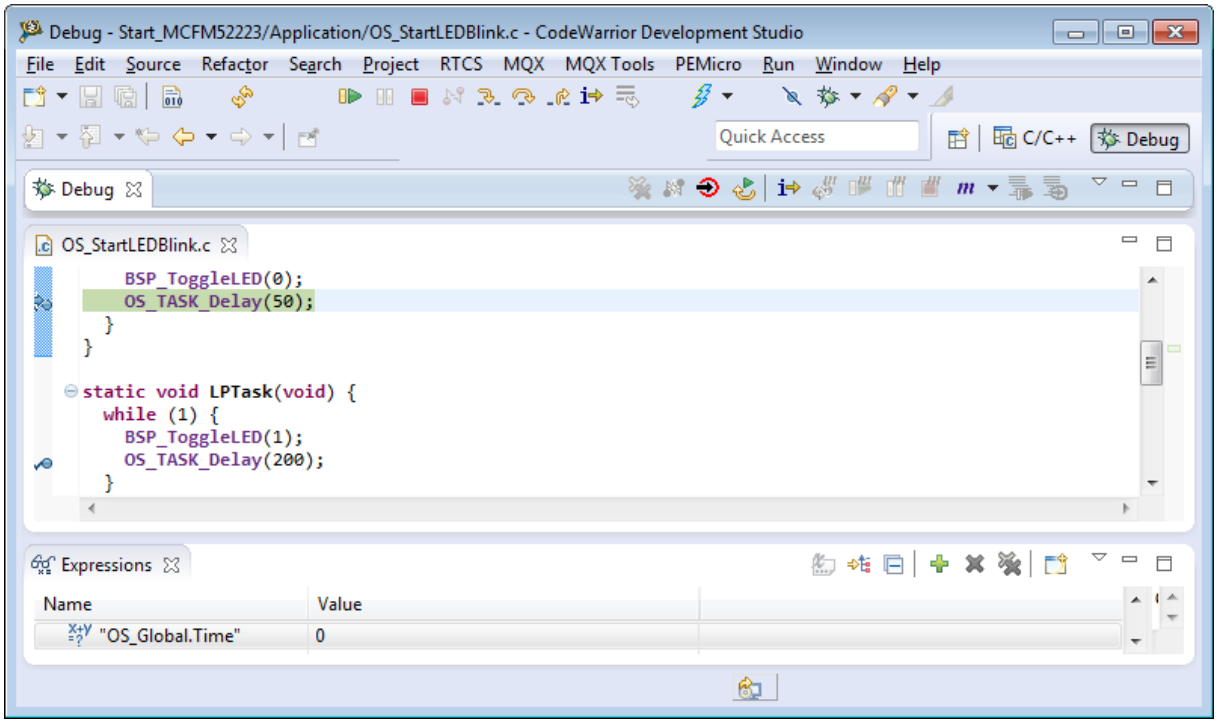


Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.

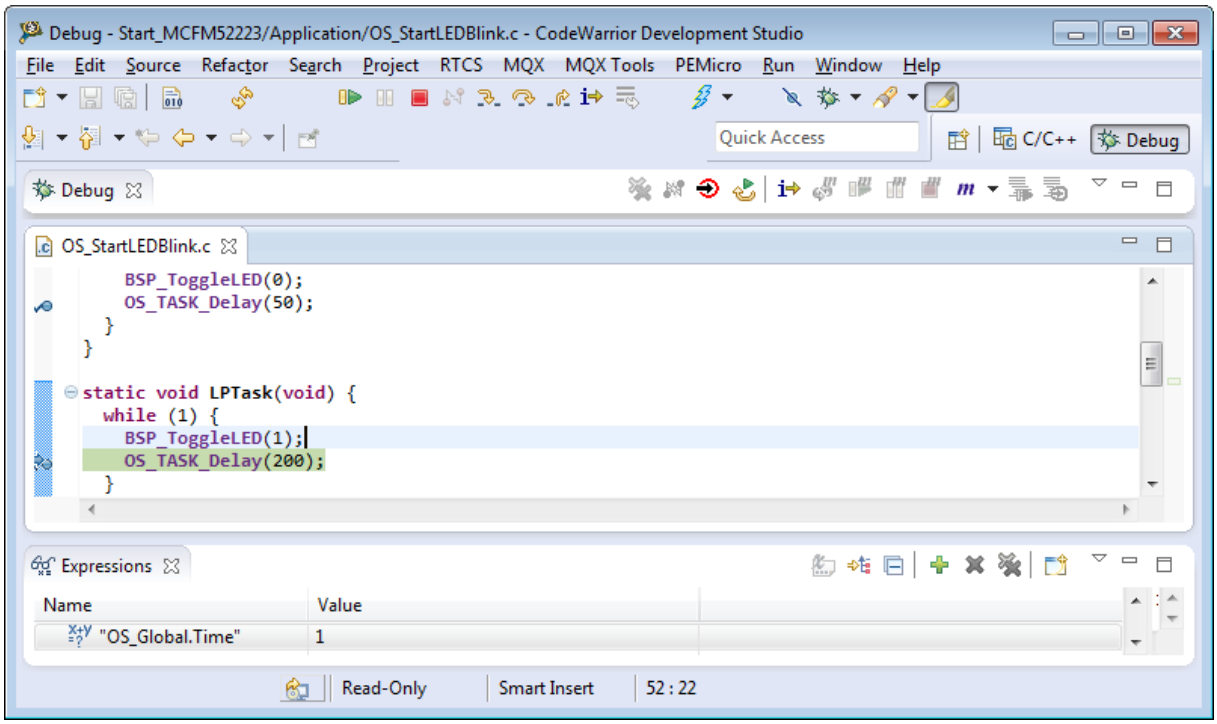


As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only.

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

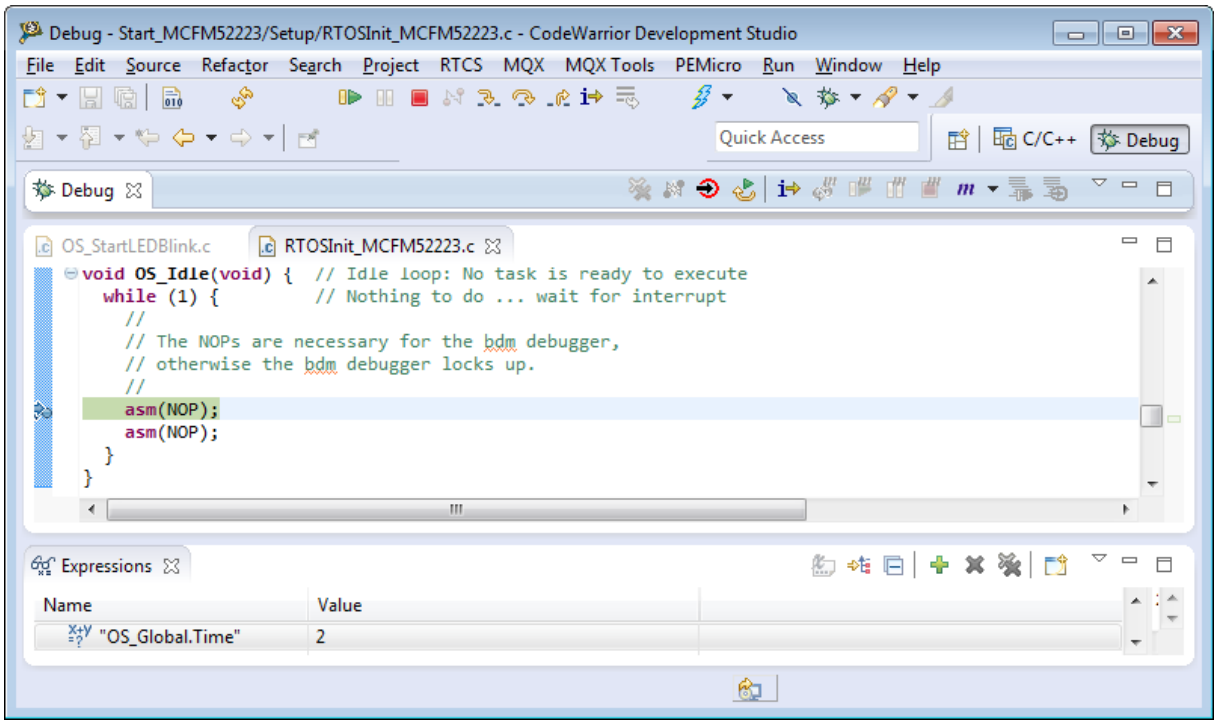


If you continue stepping, you will arrive at the task that has lower priority:



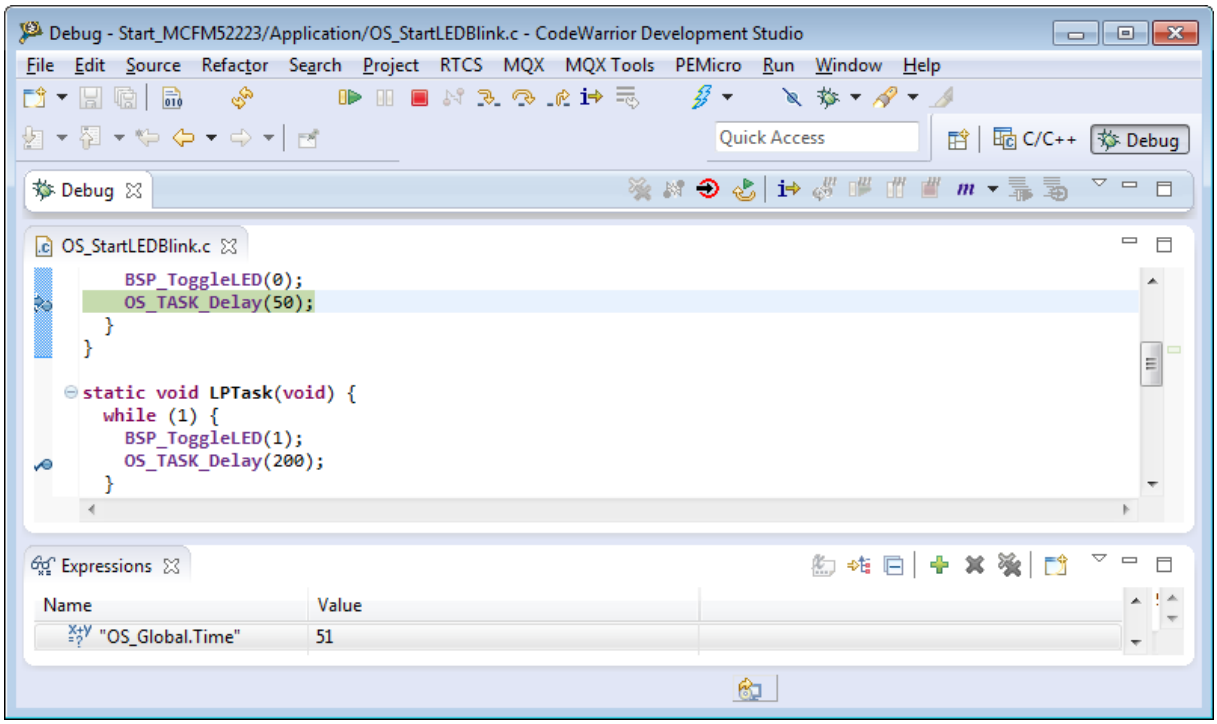
Continue to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

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 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 the directory .\Start\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 .\Start\BoardSupport\<Manufacturer>\<MCU> subfolder. It contains hardware-dependent initialization code for embOS. It initializes the system timer interrupt but can also initialize or set up the interrupt controller, clocks and PLLs, the memory protection unit and its translation table, caches and so on.
- **OS_Error.c** from one target specific subfolder .\Start\BoardSupport \<Manufacturer>\<MCU>. The error handler is used only if a debug library is used in your project.
- One **embOS library** from the subfolder .\Start\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 always 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 project 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 .\Start\BoardSupport directory. 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(), the interrupt service routines for the embOS system tick timer 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. Please contact SEGGER if you like to use other compiler features and we will perform all embOS quality test with your desired set of compiler options.

The libraries are named as follows:

`osCF_<ABI>_<CodeModel><DataModel>_<LibMode>.a`

Parameter	Meaning	Values
`ABI`	Application binary interface	StdABI : Standard ABI
`CodeModel`	Specifies the code model	f : Far code model
`DataModel`	Specifies the data model	f : Far data model
`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

`osCF_StdABI_ff_DP.lib` is the library for a project using standard ABI, far code and data model and with debug and profiling support.

Chapter 4

CPU and compiler specifics

4.1 Standard system libraries

embOS for ColdFire and CodeWarrior MCU 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 ColdFire 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 ColdFire CPUs, this minimum basic task stack size is about 56 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 512 bytes stack as a start.

5.2 System stack

The embOS system executes in supervisor mode, the scheduler also executes in supervisor mode. The minimum system stack size required by embOS is about 136 bytes (debug & stack check & profiling build). However, since the system stack is also used by the application before the start of multitasking (the call to `OS_Start()`), and because software-timers and C-level interrupt handlers also use the system-stack, the actual stack requirements depend on the application.

The size of the system stack can be changed by modifying the linker file. We recommend a minimum stack size of 1024 bytes.

5.3 Interrupt stack

If a hardware exception occurs, the ColdFire core use the current stack which can either be the task stack or the system stack.

Chapter 6

Interrupts

The NXP Coldfire core comes with an built in vectored interrupt controller.

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
- The CPU enters supervisor mode and fetches an 8 bit vector from the interrupt controller
- The CPU pushes the status register, the vector number and the return address onto the current stack.
- The CPU jumps to the vector address
- The interrupt handler function is processed.
- The interrupt handler ends with a "return from interrupt"
- The CPU switches back to the mode which was active before the exception was called.
- The CPU restores the status register and return address from the stack and continues the interrupted function.

6.2 Interrupt vector table

After Reset, the NXP Coldfire CPU uses an initial interrupt vector table which is located in ROM at address `0x00`. It contains the initial stack address, the reset vector and the vectors for all exceptions and interrupts. The interrupt vector table is located in the CPU specific assembler startup file which is delivered with embOS. Initially, all exception vectors address dummy exception handler functions located in the startup file. The peripheral interrupt handler vectors point to an embOS first level interrupt handler which is part of the embOS library. Only interrupt handler function addresses for high priority (zero latency) exceptions may be inserted in the vector table. All low priority interrupts have to call one of the embOS first level interrupt handler `OS_ISR_Handler()` or `OS_ISR_HandlerNestable()`. The interrupt vectors for peripheral interrupts are held in a separate interrupt vector table in RAM or ROM. These vectors are initialized and enabled or disabled by embOS functions during runtime. The variable vector table is declared in the CPU specific RTOSInit.c source file.

6.3 First level interrupt handler OS_ISR_Handler() and OS_ISR_HandlerNestable()

The first level interrupt handler `OS_ISR_Handler()`/`OS_ISR_HandlerNestable()` are the default interrupt handler for all peripheral interrupts. They are inserted in the vector table in the CPU specific startup code for all exceptions and peripheral interrupts. The embOS first level interrupt handler must be called for every interrupt running on low priority. The first level interrupt handler `OS_ISR_Handler()` sets the interrupt priority level for Coldfire to the zero latency interrupt priority limit, thus disabling all low priority interrupts to avoid interrupt nesting. The first level interrupt handler `OS_ISR_HandlerNestable()` is an alternate interrupt handler which allows nested interrupts. It may be used for peripheral interrupts when interrupt nesting shall be allowed for the specific interrupt.

6.4 Second level interrupt handler OS_irq_handler()

The second level interrupt handler `OS_irq_handler()` is called from the first level interrupt handlers and is part of the CPU specific RTOSInit.c source file. `OS_irq_handler()` uses the interrupt vector number passed as parameter to call the corresponding interrupt handler function from the vector table. The interrupt vector table is declared in the RTOSInit.c source file and contains the vectors for all peripherals.

6.5 Application level interrupt handler in C

Interrupt handlers for Coldfire using embOS are written as normal C-functions which do not take parameters and do not return any value. The addresses of all peripheral interrupt

handler functions called by the embOS interrupt handler have to be inserted in the peripheral interrupt vector table located in the CPU specific RTOSInit.c file.

Example

Simple interrupt routine:

```
void SysTick_Handler(void) {
  PIT0_PCSR |= PIT_PCSR_PIF;
  OS_TICK_Handle();
}
```

6.6 Zero latency interrupts

Instead of disabling interrupts when embOS does atomic operations, the interrupt level of the CPU is set to the zero latency interrupt priority limit, which is initially set to 5 for Coldfire V2 cores. Therefore all interrupt priorities higher than the zero latency interrupt priority limit can still be processed. You must not execute any embOS API function from within a zero latency interrupt function. The zero latency interrupt priority limit may be modified during runtime by calling the embOS function `OS_INT_SetPriorityLimit()`.

6.7 Interrupt priorities

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

- Any interrupt handler using embOS API functions has to run with an interrupt priorities from 0 up to the zero latency interrupt priority limit.
- Any zero latency interrupt (running at priorities above the zero latency interrupt priority limit) must not call any embOS API function.

6.8 OS_INT_SetPriorityLimit()

Description

`OS_INT_SetPriorityLimit()` is used to set the interrupt priority limit between zero latency interrupts and lower priority embOS interrupts.

The interrupt priority limit for fast interrupts is set to 5 by default. This means, all interrupts with higher priority from 6 up to the maximum CPU specific priority are never be disabled by embOS.

Prototype

```
void OS_INT_SetPriorityLimit(unsigned int Priority);
```

Parameters

Parameter	Description
Priority	The highest value useable as priority for embOS interrupts. 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 interrupt priority supported by the CPU, which is 7 for Coldfire CPU. To modify the default priority limit, `OS_INT_SetPriorityLimit()` be called before embOS was started. In the default start projects, `OS_INT_SetPriorityLimit()` The start projects use the default Fast interrupt

priority limit. Any interrupts running above the Fast interrupt priority limit must embOS function.

Example

```
int main(void) [
  OS_Init();
  OS_InitHW();
  OS_INT_SetPriorityLimit(6);
  OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
  OS_Start();
  return 0;
}
```

6.9 OS_EnableISR()

Description

OS_EnableISR() is used to install a specific interrupt vector when CPUs with interrupt controller are used.

Prototype

```
OS_ISR_HANDLER* OS_EnableISR (unsigned int    ISRIndex,
                              OS_ISR_HANDLER* pISRHandler,
                              unsigned int    Prio,
                              unsigned int    SubPrio);
```

Parameters

Parameter	Description
ISRIndex	Index of the interrupt source which should be enabled.
pISRHandler	Address of the interrupt handler function.
Priority	Interrupt priority
SubPriority	Interrupt sub-priority

Return value

The address of the previous installed interrupt which was installed at the addressed vector number before.

Additional information

This function installs the interrupt vector, modifies the priority and enable the interrupt in the interrupt controller. When the interrupt handler table is located in ROM, the vector can not be modified, the function sets the priority only and enables it.

> **Note**
>
> The priority and sub-priority must be unique for every ISR. The embOS debug code calls OS_Error(OS_ERR_ISR_VECTOR) when this is violated.

Example

```
void OS_InitHW(void) {
  OS_EnableISR(OS_ID_SYSTICK, SysTick_Handler, 1, 0);
}
```


6.10 OS_DisableISR()

Description

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

Prototype

```
void OS_DisableISR(unsigned int ISRIndex);
```

Parameters

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

Additional information

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

```
void Task(void) {
  OS_DisableISR((OS_ID_UART);
}
```


Chapter 7

Technical data

This chapter lists technical data of embOS used with ColdFire 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	36
Software timer	16
Mutex	16
Semaphore	8
Mailbox	24
Queue	32
Task event	0
Event object	12

