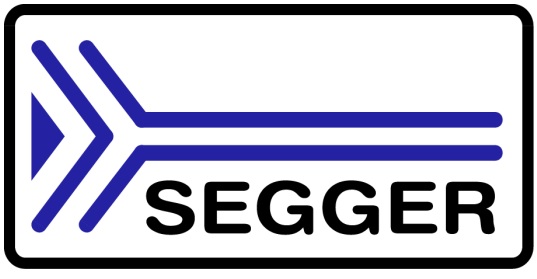
embOS

Real-Time
Operating System

CPU & Compiler
specifics for PowerPC
using
CodeWarrior for MCU

Document: UM01054
Software version 4.04a
Revision: 0
Date: December 3, 2014



A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Disclaimer

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

Copyright notice

Trademarks

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

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

Contact address

SEGGER Microcontroller GmbH & Co. KG

In den Weiden 11
D-40721 Hilden

Germany

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

Manual versions

This manual describes the current software version. If any error occurs, inform us and we will try to assist you as soon as possible.
Contact us for further information on topics or routines not yet specified.

Print date: December 3, 2014

Software	Revision	Date	By	Description
4.04a	0	141203	TS	New software version.
3.88h	0	140210	TS	First version.

About this document

Assumptions

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

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

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

How to use this manual

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

Typographic conventions for syntax

This manual uses the following typographic conventions:

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

Table 2.1: Typographic conventions

SEGGER Microcontroller GmbH & Co. KG develops and distributes software development tools and ANSI C software components (middleware) for embedded systems in several industries such as telecom, medical technology, consumer electronics, automotive industry and industrial automation.

SEGGER's intention is to cut software development time for embedded applications by offering compact flexible and easy to use middleware, allowing developers to concentrate on their application.

Our most popular products are emWin, a universal graphic software package for embedded applications, and embOS, a small yet efficient real-time kernel. emWin, written entirely in ANSI C, can easily be used on any CPU and most any display. It is complemented by the available PC tools: Bitmap Converter, Font Converter, Simulator and Viewer. embOS supports most 8/16/32-bit CPUs. Its small memory footprint makes it suitable for single-chip applications.

Apart from its main focus on software tools, SEGGER develops and produces programming tools for flash micro controllers, as well as J-Link, a JTAG emulator to assist in development, debugging and production, which has rapidly become the industry standard for debug access to ARM cores.

Corporate Office:
http://www.segger.com

United States Office:
http://www.segger-us.com

EMBEDDED SOFTWARE (Middleware)

emWin
Graphics software and GUI
emWin is designed to provide an efficient, processor- and display controller-independent graphical user interface (GUI) for any application that operates with a graphical display.

embOS
Real Time Operating System
embOS is an RTOS designed to offer the benefits of a complete multitasking system for hard real time applications with minimal resources.

embOS/IP
TCP/IP stack
embOS/IP a high-performance TCP/IP stack that has been optimized for speed, versatility and a small memory footprint.

emFile
File system
emFile is an embedded file system with FAT12, FAT16 and FAT32 support. Various Device drivers, e.g. for NAND and NOR flashes, SD/MMC and CompactFlash cards, are available.

USB-Stack
USB device/host stack
A USB stack designed to work on any embedded system with a USB controller. Bulk communication and most standard device classes are supported.

SEGGER TOOLS

Flasher
Flash programmer
Flash Programming tool primarily for micro controllers.

J-Link
JTAG emulator for ARM cores
USB driven JTAG interface for ARM cores.

J-Trace
JTAG emulator with trace
USB driven JTAG interface for ARM cores with Trace memory. supporting the ARM ETM (Embedded Trace Macrocell).

J-Link / J-Trace Related Software
Add-on software to be used with SEGGER's industry standard JTAG emulator, this includes flash programming software and flash breakpoints.

Table of Contents

Chapter 1

Using embOS for PowerPC

This chapter describes how to start with embOS for PowerPC cores and the Freescale Codewarrior compiler for Embedded PowerPC. You should follow these steps to become familiar with embOS.

1.1 Installation

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

To install it, proceed as follows:

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

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

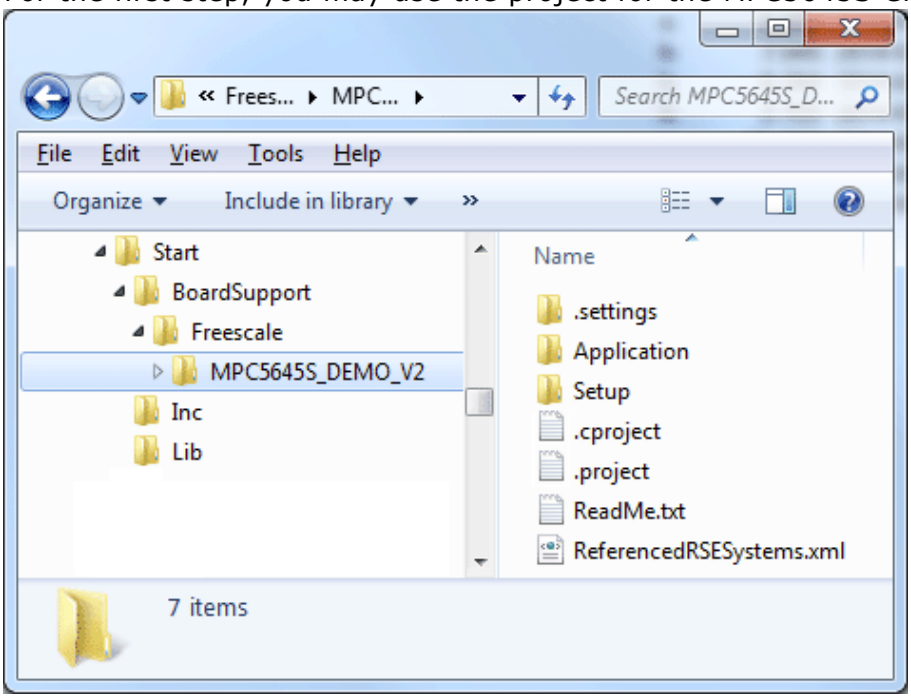
You should do this even if you do not intend to use the project manager for your application development to become familiar with embOS.

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

1.2 First steps

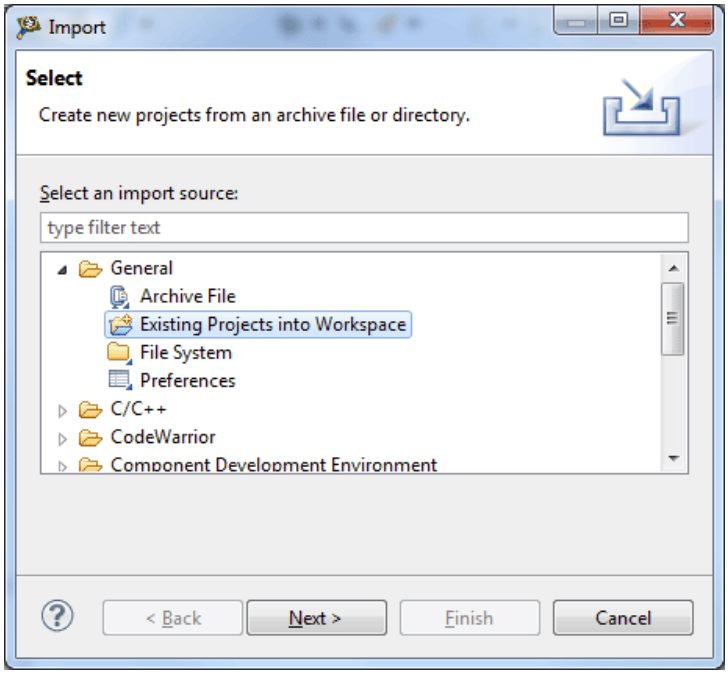
After installation of embOS you can create your first multitasking application. You received several ready to go sample start projects and every other files needed in the subfolder **Start**. It is a good idea to use one of them as a starting point for all of your applications. The subfolder **BoardSupport**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

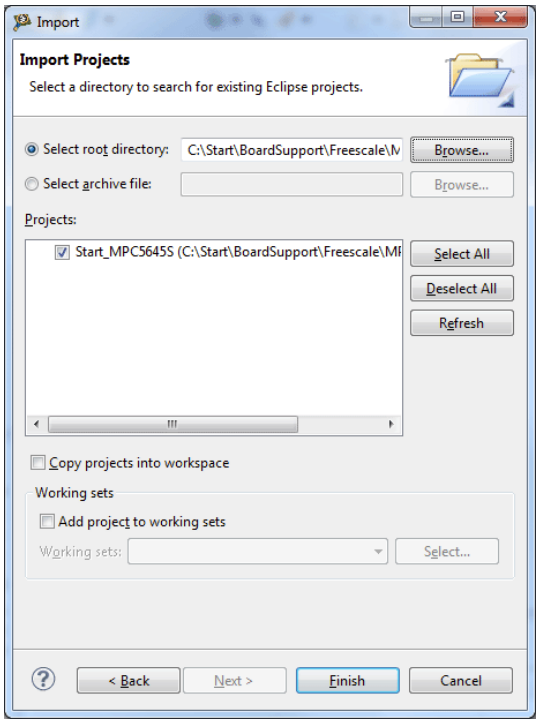
For the first step, you may use the project for the MPC5645S CPU:



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

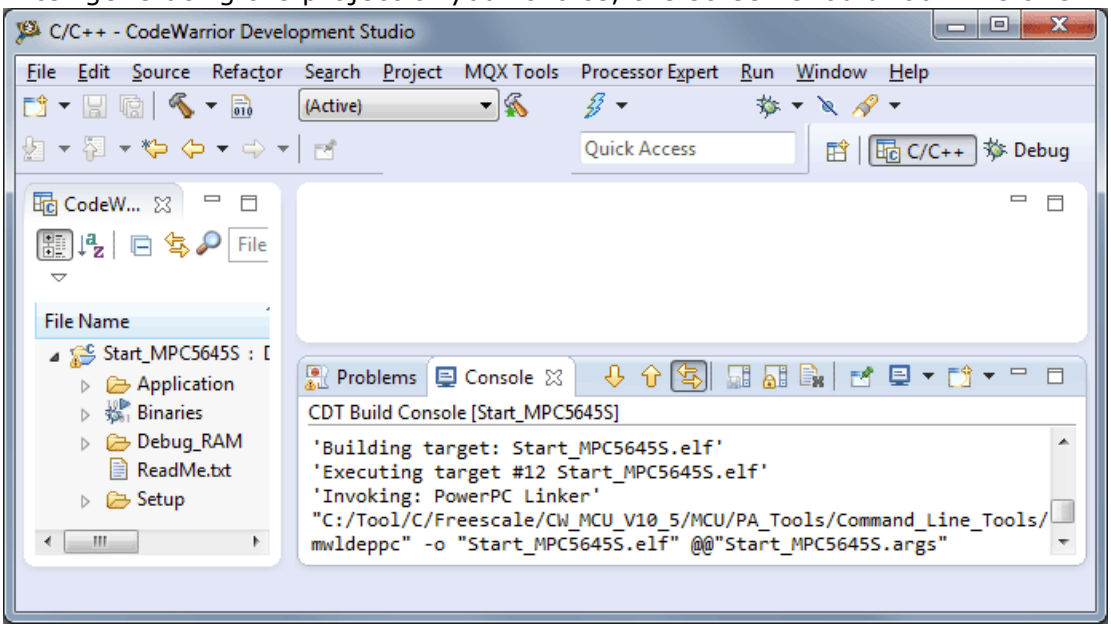
- Create a work directory for your application, for example `c:\work`.
- Copy the whole folder **Start** which is part of your embOS distribution into your work directory.
- Clear the read-only attribute of all files in the new **Start** folder.
- Create a new workspace at the location of your choice and import the project from **Start\BoardSupport\Freescale\MPC5645S_DEMO_V2**:





- Build the project. It should be build without any error or warning messages.

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



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

1.3 The example application Start_2Tasks.c

The following is a printout of the example application `Start_2Tasks.c`. It is a good starting point for your application. (Note that the file actually shipped with your port of embOS may look slightly different from this one.)

What happens is easy to see:

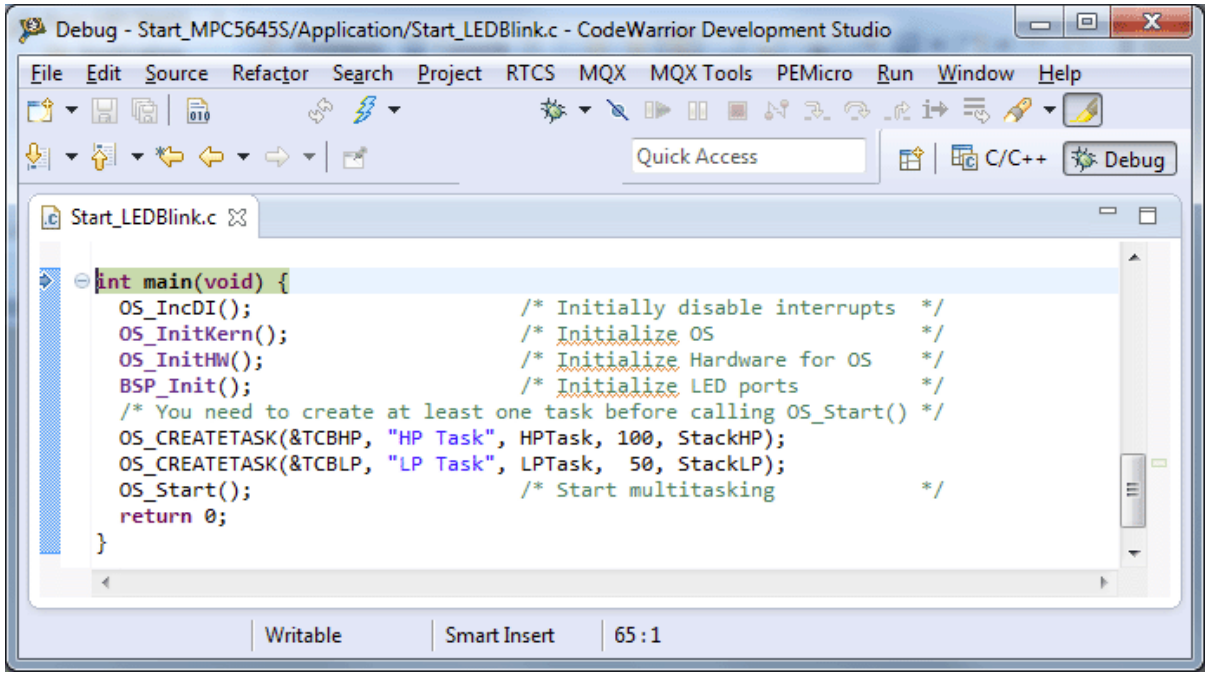
After initialization of embOS; two tasks are created and started.
The two tasks are activated and execute until they run into the delay, then suspend for the specified time and continue execution.

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

#include "RTOS.h"

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

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

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

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

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

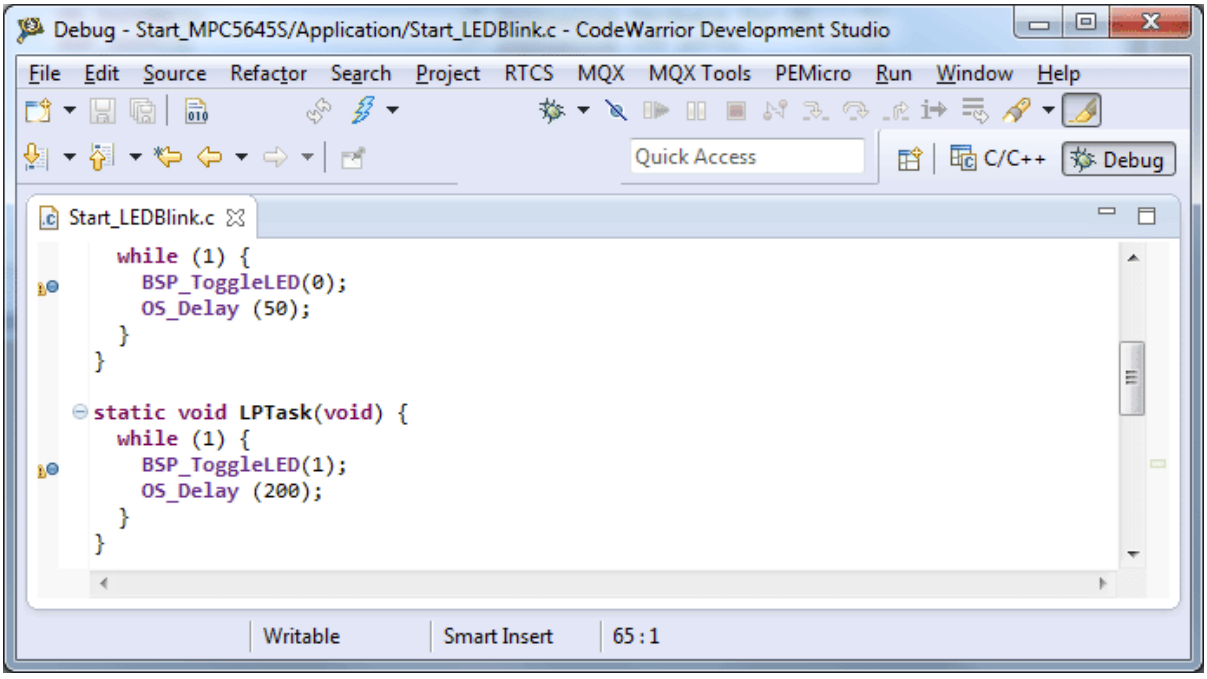
1.4 Stepping through the sample application

When starting the debugger, you will see the `main` function (see example screenshot below). The `main` function appears as long as the debugger option **Run to main** is selected, which it is by default. Now you can step through the program. `OS_IncDI()` initially disables interrupts.
`OS_InitKern()` is part of the embOS library and written in assembler; you can therefore only step into it in disassembly mode. It initializes the relevant OS variables. Because of the previous call of `OS_IncDI()`, interrupts are not enabled during execution of `OS_InitKern()`.
`OS_InitHW()` is part of `RTOSInit_*.c` and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the timer-tick-interrupt for embOS. Step through it to see what is done.

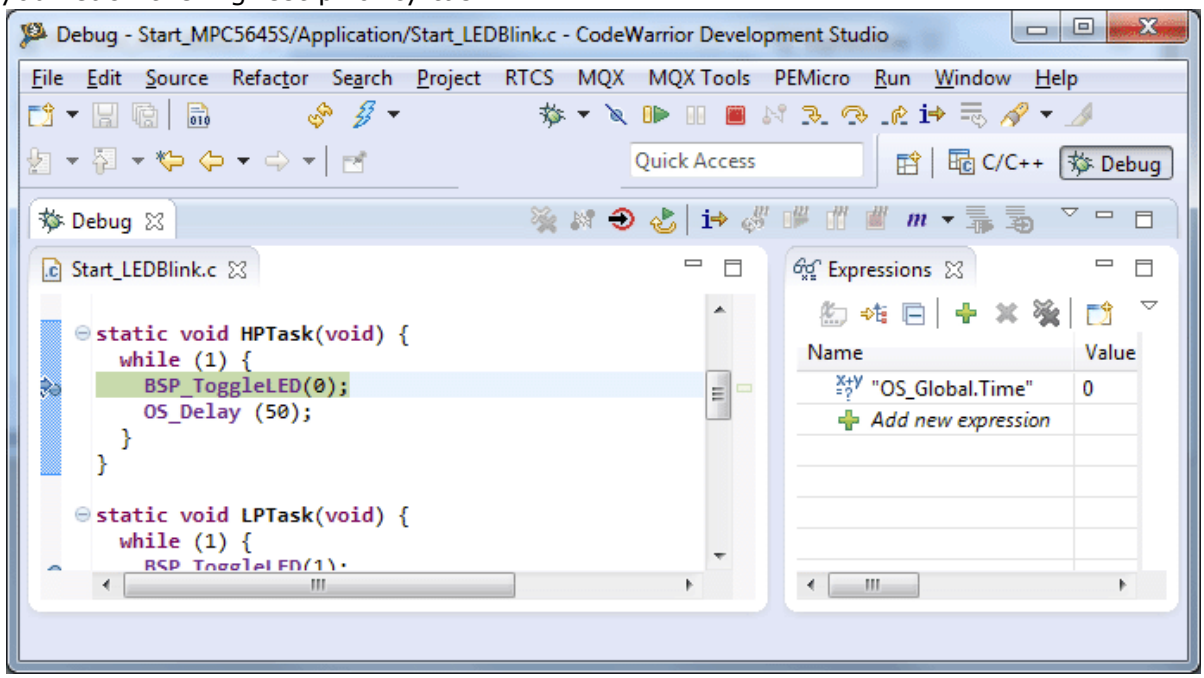
`OS_Start()` should be the last line in main, because it starts multitasking and does not return.



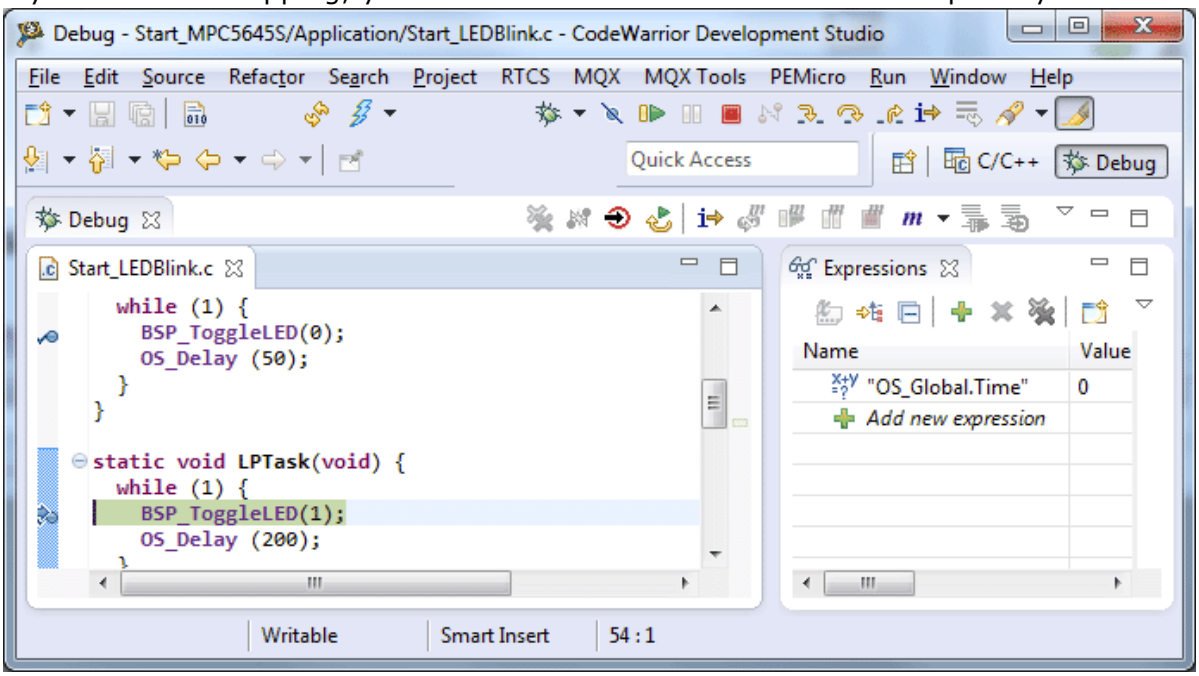
Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.



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.

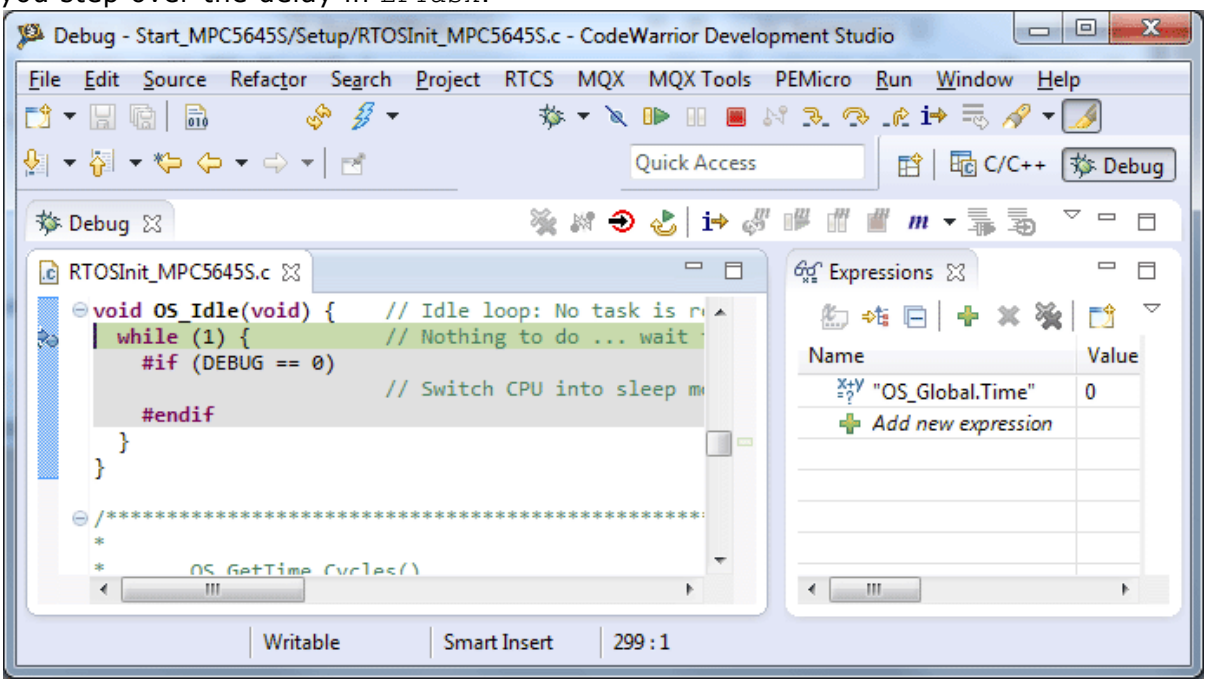


If you continue stepping, you will arrive in the task that has lower priority:



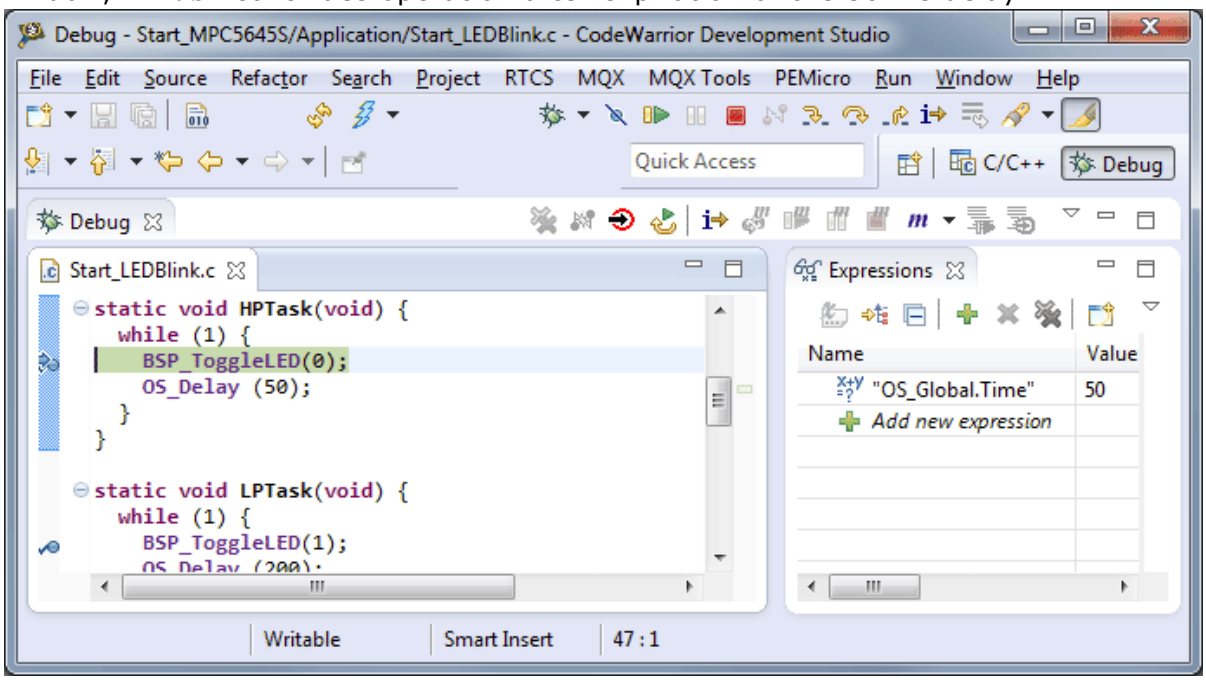
Continue to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

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



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

As can be seen by the value of embOS timer variable OS_Time, shown in the Watch window, `HPTask` continues operation after expiration of the 50 ms delay.



Chapter 2

Build your own application

This chapter provides all information to setup your own embOS project.

2.1 Introduction

To build your own application, you should always start with one of the supplied sample workspaces and projects. Therefore, select an embOS workspace as described in First steps on page 9 and modify the project to fit your needs. Using a sample project as starting point has the advantage that all necessary files are included and all settings for the project are already done.

2.2 Required files for an embOS for PowerPC

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

- `RTOS.h` from subfolder `Inc\`.
  This header file declares all embOS API functions and data types and has to be included in any source file using embOS functions.
- RTOSInit_*.c from one target specific **BoardSupport\<Manufacturer>\<MCU>** subfolder.
  It contains hardware-dependent initialization code for embOS. It initializes the system timer, timer interrupt and optional communication for embOSView via UART.
- One embOS library from the subfolder `Lib\`.
- OS_Error.c from one target specific subfolder **BoardSupport\<Manufacturer>\<MCU>**.
  The error handler is used if any library other than Release build library is used in your project.
- Additional low level init code may be required according to CPU.

When you decide to write your own startup code or use a `__low_level_init()` function, ensure that non-initialized variables are initialized with zero, according to C standard. This is required for some embOS internal variables.
Your `main()` function has to initialize embOS by a call of `OS_InitKern()` and `OS_InitHW()` prior any other embOS embOS functions are called.
You should then modify or replace the `Start_2Task.c` source file in the subfolder `Application\`.

2.3 Change library mode

For your application you 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, add it in the project settings. Only add one embOS library at the same time.
- Check and set the appropriate `OS_LIBMODE_*` define as preprocessor option and/or modify the OS_Config.h file accordingly.

Chapter 3

PowerPC specifics

3.1 CPU modes

embOS for PowerPC supports all memory and code model combinations that the Freescale CodeWarrior for MCU compiler supports.

3.2 Available libraries

embOS for PowerPC and Freescale CodeWarrior for MCU compiler comes with 7 different libraries, one for each library mode. The library names follow the naming convention for the EWL system libraries from CodeWarrior for MCU compiler.

3.2.1 Naming conventions for prebuilt libraries compatible to CodeWarrior for MCU compiler

embOS for PowerPC and CodeWarrior for MCU is shipped with different prebuilt libraries with different combinations of the following features:

- CPU Core - Core
- Instruction set - `ISA`
- Library mode - `LibMode`

The libraries are named as follows:

`libos_<Core>_<ISA>_<Libmode>.a`

Parameter	Meaning	Values
`Core`	Specifies the CPU core	E200z4: Always e200z4 core
`ISA`	Specifies the instruction set	VLE: Always variable length encoding
`LibMode`	Specifies the library mode	XR: Extreme Release
		R: Release
		S: Stack check
		SP: Stack check + profiling
		D: Debug
		DP: Debug + profiling
		DT: Debug + profiling + trace

Example

libos_E200z4_VLE_DP.a is the library for a project using a e200z4 core, VLE mode, with debug and profiling support.

Chapter 4

Compiler specifics

4.1 Standard system libraries

embOS for PowerPC and CodeWarrior for MCU may be used with CodeWarrior standard libraries.

4.2 Embedded Floating Point Unit EFPU and Signal Processing Unit SPE support

The PowerPC e200z4 core comes with an integrated Emdedded Floating Point Unit EFPU and Signal Processing Unit SPE.
When selecting the CPU and activating the EFPU/SPE support in the project options, the startup code enables the EFPU/SPE units.
With embOS, the EFPU/SPE registers have to be saved and restored when preemptive or cooperative task switches are performed.
For efficiency reasons, embOS does not save and restore the EFPU/SPE registers for every task automatically. The context switching time and stack load are therefore not affected when the EFPU/SPE unit is not used or needed.
Saving and restoring the EFPU/SPE registers can be enabled for every task individually by extending the task context of the tasks which need and use the EFPU/SPE.

4.2.1 OS_ExtendTaskContext_EFPU_SPE()

Description

`OS_ExtendTaskContext_EFPU_SPE()` has to be called as first function in a task, when the EFPU/SPE is used in the task and the EFPU/SPE registers have to be added to the task context.

Prototype

`void` OS_ExtendTaskContext_EFPU_SPE(`void`)

Return value

None.

Additional Information

OS_ExtendTaskContext_EFPU_SPE() extends the task context to save and restore the EFPU/SPE registers during context switches.
Additional task context extension for a task by calling `OS_ExtendTaskContext()` is not allowed and will call the embOS error handler `OS_Error()` in debug builds of embOS.
There is no need to extend the task context for every task. Only those tasks using the EFPU/SPE for calculation have to be extended.

4.2.2 Using the EFPU/SPE in interrupt service routines

Using the EFPU/SPE in interrupt service routines requires additional functions to save and restore the EFPU/SPE registers.
embOS delivers two functions to save and restore the EFPU/SPE context in an interrupt service routine.

4.2.2.1 OS_EFPU_SPE_Save()

Description

`OS_EFPU_SPE_Save()` has to be called as first function in an interrupt service routine, when the EFPU/SPE is used in the interrupt service routine. The function saves the temporary EFPU/SPE registers on the stack.

Prototype

`void` OS_EFPU_SPE_Save(`void`)

Return value

None.

Additional Information

OS_EFPU_SPE_Save() declares a local variable which reserves space for all temporary registers and stores the registers in the variable.
After calling the OS_EFPU_SPE_Save() function, the interrupt service routine may use the EFPU/SPE for calculation without destroying the saved content of the EFPU/SPE registers.
To restore the registers, the ISR has to call OS_EFPU_SPE_Restore() at the end.

4.2.2.2 OS_EFPU_SPE_Restore()

Description

`OS_EFPU_SPE_Restore()` has to be called as last function in an interrupt service routine, when the EFPU/SPE registers were saved by a call of OS_EFPU_SPE_Save() at the beginning of the ISR. The function restores the temporary EFPU/SPE registers from the stack.

Prototype

`void` OS_EFPU_SPE_Restore(`void`)

Return value

None.

Additional Information

OS_EFPU_SPE_Restore() restores the temporary EFPU/SPE registers which were saved by a previous call of OS_EFPU_SPE_Save().
It has to be used together with OS_EFPU_SPE_Save() and should be the last function called in the ISR.

Example of a interrupt service routine using EFPU/SPE

```
void ADC_ISR_Handler(void) {
  OS_EFPU_SPE_Save();      // Save EFPU/SPE registers
  DoSomeFloatOperation();
  OS_EFPU_SPE_Restore();  // Restore EFPU/SPE registers.
}
```


Chapter 5

Stacks

5.1 Task stack for PowerPC

Each task uses its individual stack. The stack poiter 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 PowerPC CPUs, this minimum basic task stack size is about 90 bytes. Because any function call uses some amount of stack and every exception also pushes at least 80 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 PowerPC

The minimum system stack size required by embOS is about 144 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 project settings.
We recommend a minimum stack size of 512 bytes for the C stack.

5.3 Interrupt stack for PowerPC

The PowerPC CPU has no separate interrupt stack. The interrupt runs on the current stack which could be task stack or C 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 stores the machine status register and the return address into the register SSR0 and SSR1.
- The CPU jumps to the embOS low level interrupt handler IRQ_Handler().
- IRQ_Handler() saves all scratch registers.
- IRQ_Handler() calls the embOS high level interrupt handler OS_irq_handler().
- OS_irq_handler() reads the interrupt vector address from the interrupt controller and calls the application handler function.
- The low level interrupt handler restores all scratch register. It ends with a "rfi" instruction and restores the machine status register and the jumps back to the instruction where the interrupt occurred.

6.2 Defining interrupt handlers in C

Interrupt handlers called from the embOS interrupt handler in RTOSInit*.c are just normal C-functions which do not take parameters and do not return any value.

The default C interrupt handler OS_irq_handler() in RTOSInit*.c first calls OS_Enterinterrupt() or OS_EnterNestableInterrupt() to inform embOS that interrupt code is running. Then this handler examines the source of interrupt and calls the related interrupt handler function.

Finally, the default interrupt handler OS_irq_handler() in RTOSInit*.c calls OS_LeaveInterrupt() or OS_LeaveNestableInterrupt() and returns to the lowe level interrupt handler IRQ_Handler().

Depending on the interrupting source, it may be required to reset the interrupt pending condition of the related peripherals.

Example

Simple interrupt routine:

```
static void OS_ISR_Tick (void) {
  PIT0_TFLG = 0x01u; // Clear PIT0 flag
  OS_TICK_Handle();
}
```


6.2.1 OS_PPC_InstallISRHandler(): Install an interrupt handler

Description

OS_PPC_InstallISRHandler() is used to setup an interrupt handler function for a specific interrupt source.

Prototype

```
OS_ISR_HANDLER* OS_PPC_InstallISRHandler (OS_U32 ISRIndex,
                                          OS_ISR_HANDLER* pISRHandler,
                                         OS_U8 Prio);
```

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.
pISRHandler	Address of the interrupt handler function.
Prio	The priority which should be set for the specific interrupt. Prio ranges from 0 (lowest priority) to 15 (highest priority)

Table 6.1: OS_PPC_InstallISRHandler parameter list

Return value

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

Additional Information

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

6.3 Interrupt vector table

embOS for PowerPC uses the INTC software vector mode and a vector table in RAM. All entries are initialized with a default handler function. The application has to call OS_PPC_InstallISRHandler() to install an interrupt handler for a peripheral interrupt.

6.4 Interrupt priorities

The interrupt controller supports interrupt priority. The application has to call OS_PPC_InstallISRHandler() to setup a priority for a peripheral interrupt.

6.5 Interrupt nesting

The PowerPC CPU uses a priority controlled interrupt controller which allows nesting of interrupts. Any interrupt with a higher priority may interrupt an interrupt handler running on a lower priority. Nesting can only be enabled globally for all interrupts. To do set the define *ALLOW_NESTED_INTERRUPTS* as a project option or change this define in the according RTOSInit.c to "1".

6.6 Zero latency interrupts

Zero latency interrupts are not supported by embOS for PowerPC.

Chapter 7

STOP / WAIT Mode

7.1 Introduction

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

Chapter 8

Technical data

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 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	44
Resource semaphore	16
Counting semaphore	8
Mailbox	24
Software timer	20

Chapter 9

Files shipped with embOS

List of files shipped with embOS

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

Any additional files shipped serve as example.

Index

