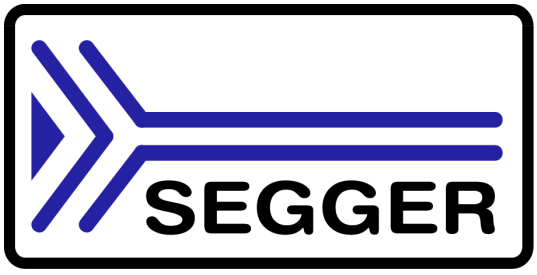
embOS

Real-Time
Operating System

CPU & Compiler
specifics for Freescale
Coldfire V1 using
Freescale CodeWarrior

Document: UM01006
Software version 4.16
Revision: 0
Date: February 22, 2016



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: February 22, 2016

Software	Revision	Date	By	Description
4.16	0	160222	MC	Initial FrameMaker Version.

About this document

Assumptions

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

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

- DOS command line

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

How to use this manual

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

Typographic conventions for syntax

This manual uses the following typographic conventions:

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

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 4

Using embOS

This chapter describes how to start with and use embOS. You should follow these steps to become familiar with embOS.

4.1 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*.

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.

4.2 First steps

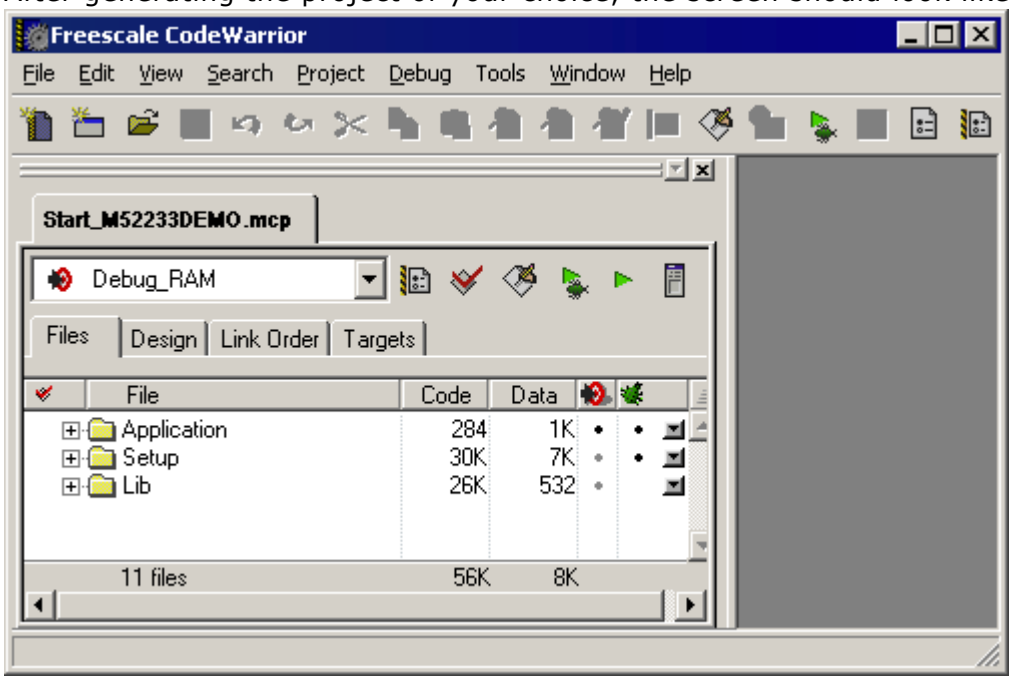
After installation of embOS you can create your first multitasking application. You have received several ready to go sample start workspaces and projects and every other files needed in the subfolder **Start**. It is a good idea to use one of them as a starting point for all of your applications. The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

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\<DeviceManufactor>\<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.

4.3 The example application OS_StartLEDBlink.c

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

What happens is easy to see:

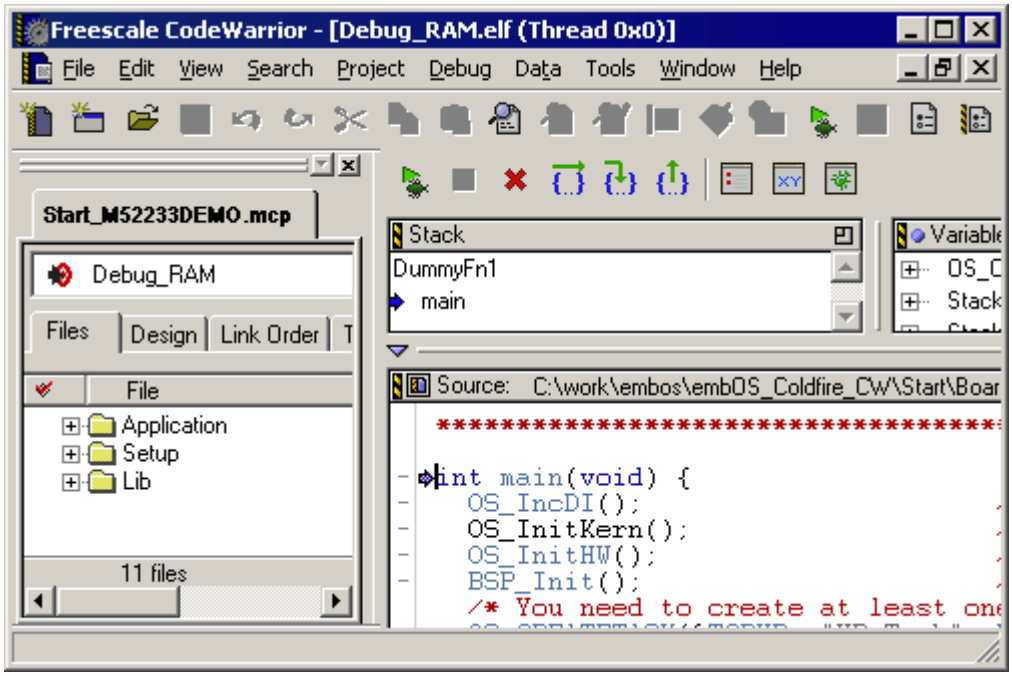
After initialization of embOS; two tasks are created and started.
The two tasks are activated and execute until they run into the delay, then suspend for the specified time and continue execution.

```
/*********************************************************************
*                SEGGER Microcontroller GmbH & Co. KG                *
*                        The Embedded Experts                        *
**********************************************************************
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).
--------- END-OF-HEADER ------------------------------------------------
*/

#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_Delay (50);
  }
}

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

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

/****** End Of File *************************************************/
```

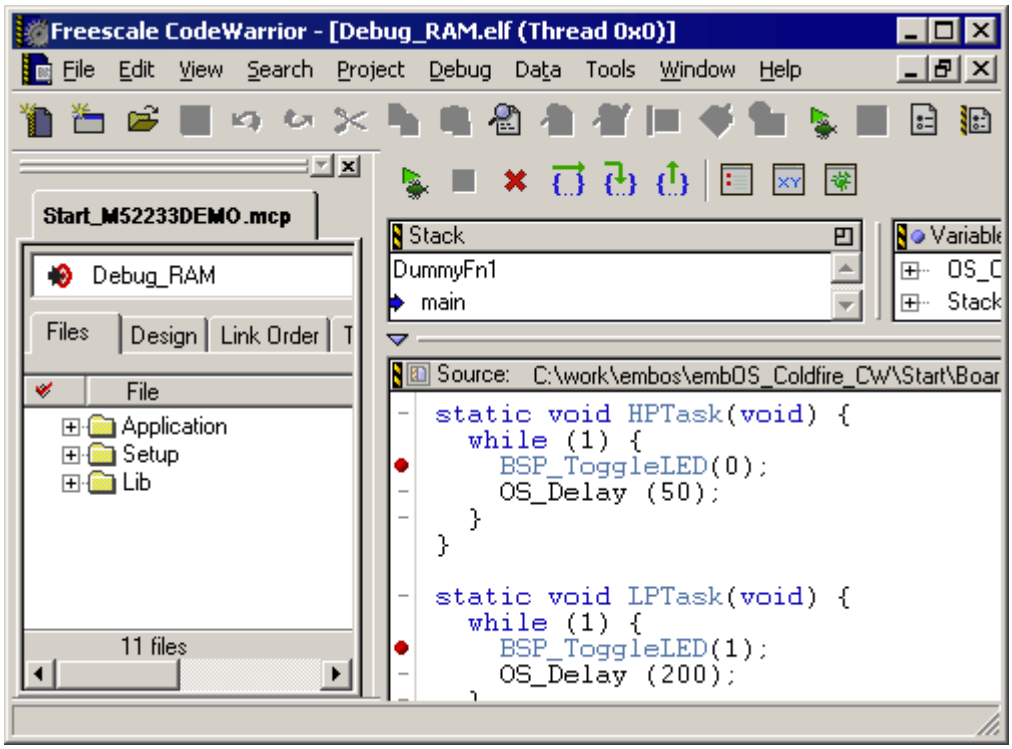

4.4 Stepping through the sample application

When starting the debugger, you will see the `main()` function (see example screenshot below). The `main()` function appears as long as project option **Run to main** is selected, which it is enabled 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 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.

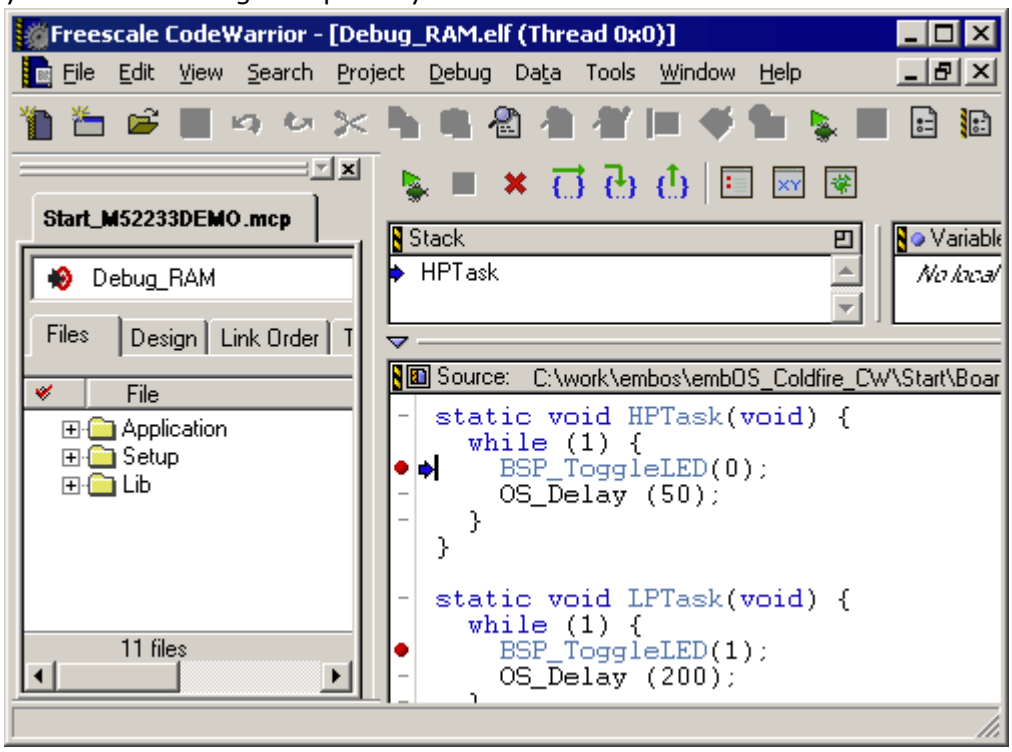


Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.

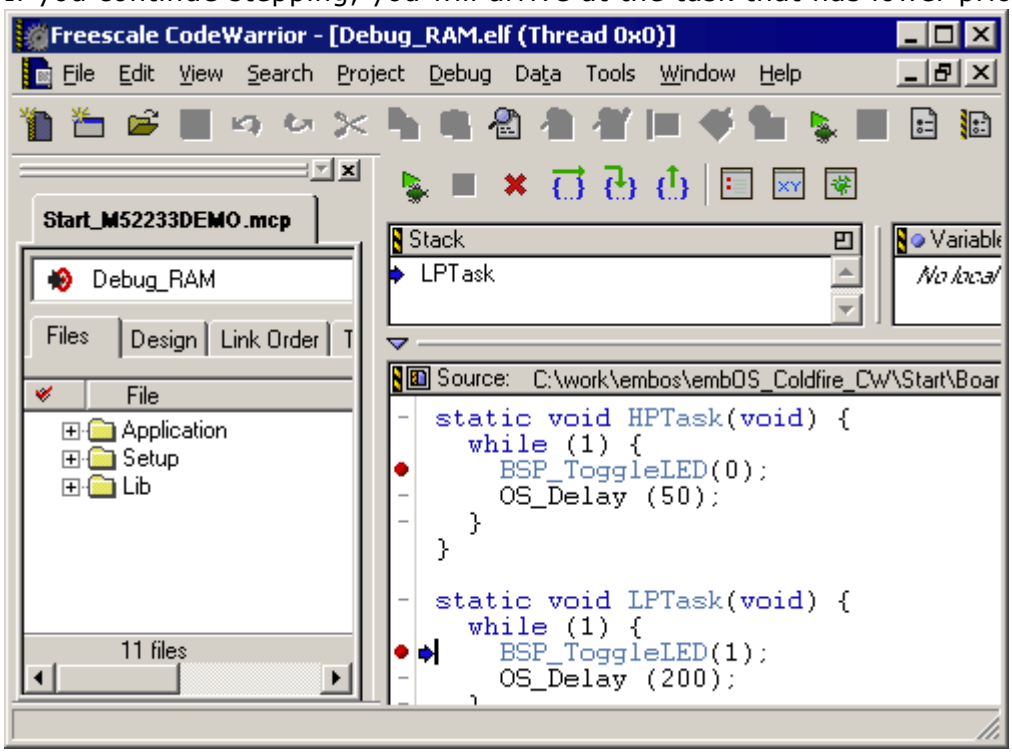


As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only.

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

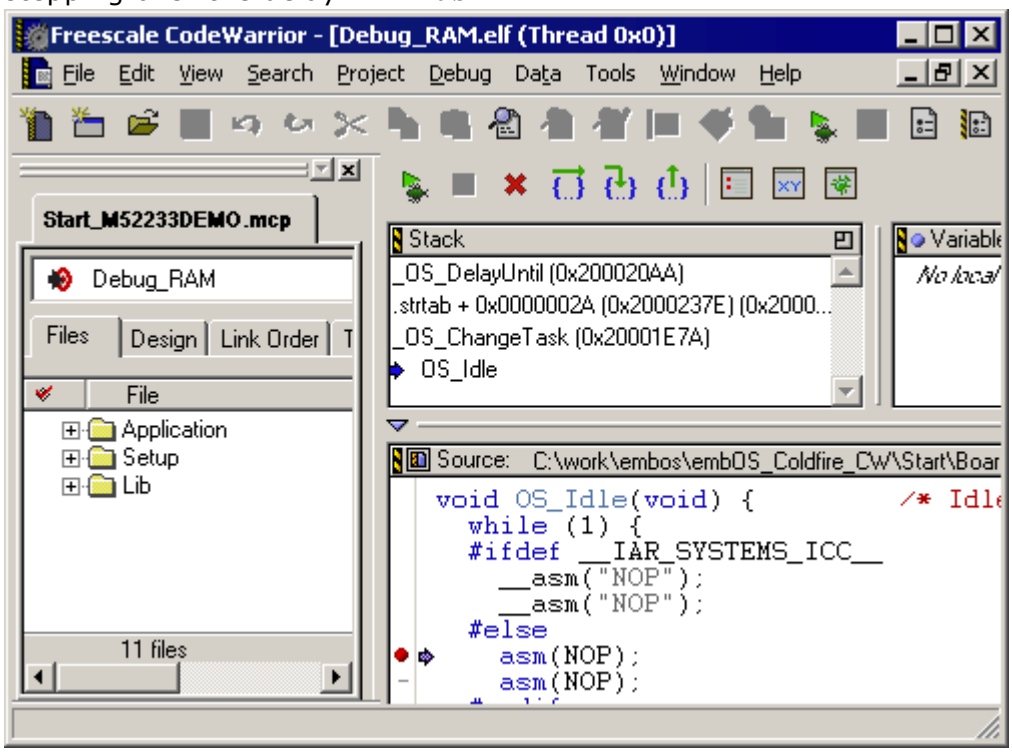


If you continue stepping, you will arrive at the task that has lower priority:



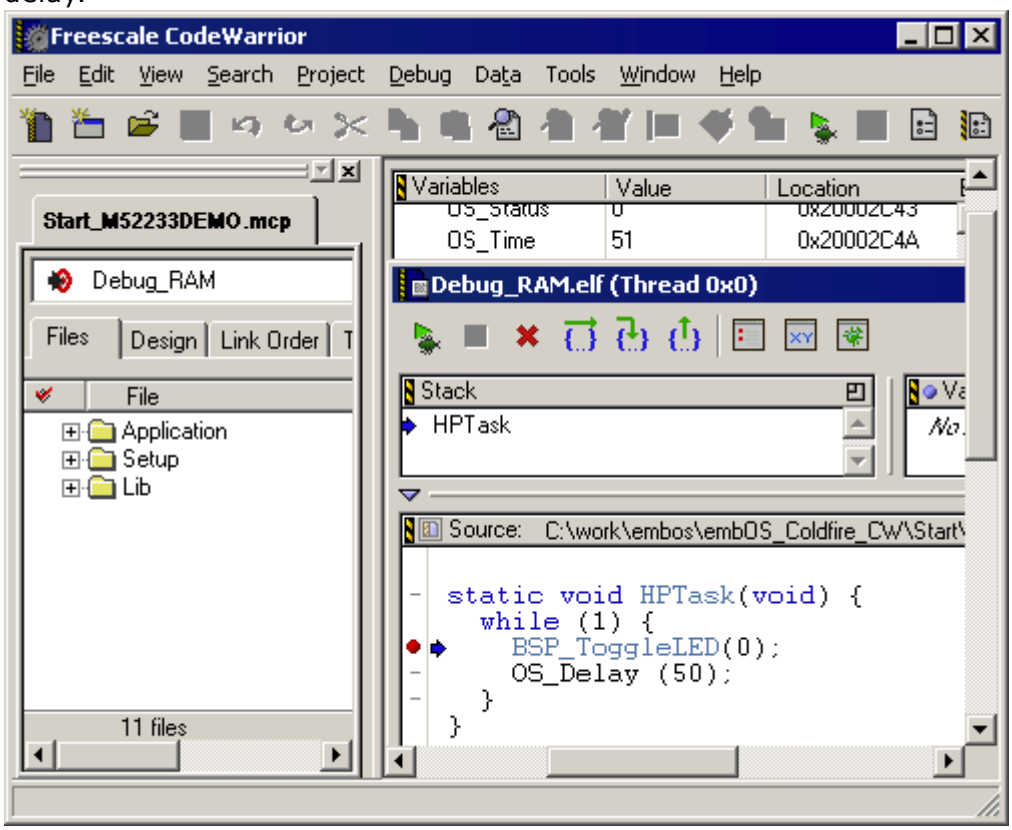
Continue to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

You will arrive there when you step into the `OS_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 5

Build your own application

This chapter provides all information to set up your own embOS project.

5.1 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* 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.

5.2 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.
- 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 debug library is used in your project.
- 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 a call of `OS_InitKern()` and `OS_InitHW()` prior any other embOS functions are called.
You should then modify or replace the `OS_StartLEDBlink.c` source file in the subfolder **Application**.

5.3 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.

5.4 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 6

Libraries

This chapter includes CPU-specific information such as CPU-modes and available libraries.

6.1 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_4i_CF_<ABI>_<CodeModel><DataModel>_<LibMode>.lib`

Parameter	Meaning	Values
ABI	Specifies the application binary interface	<nothing>: Code generation with the compact ABI RegABI: Code generation with the register ABI StdABI: Code generation with the standrd ABI
CodeModel	Specifies the code model	n: near code model f: far code model
DataModel	Specifies the data model	n: near 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
		dt: Debug + profiling + trace

Example

los_4i_CF_StdABI_ff_DP.lib is the library for a project using Standard ABI, far code model, far data model and embOS with debug and profiling support.

Chapter 7

CPU and compiler specifics

7.1 Standard system libraries

embOS for Freescale Coldfire V1 and *Freescale* Codewarrior may be used with any of the standard *Freescale* Codewarrior system libraries for most of all projects.

7.1.1 Heap management, dynamic memory allocation

Heap management and file operation functions of standard system libraries are not reentrant and can therefore not be used with embOS, if these non thread safe functions are used from different tasks.

For heap management, embOS delivers its own thread safe functions whichmay be used. These functions are described in the generic, CPU independentembOS manual.

When using dynamic memory allocation in an embOS project, the `sbrk()` function which comes with embOS has to be used.

The `sbrk()` function from the run-time libraries can not be used because it may compare the pointer into the heap against the current stack pointer of the CPU to decide whether free space is available. As embOS tasks run on its own stack which might be located before the start address of the heap, this function will fail.

Adding the source file sbrk.c found in the CPU specific `SETUP` folder automatically replaces the `sbrk()` function from the run-time library.

7.1.2 Defining the heap memory

The `sbrk()` function which comes with embOS examines the heap limit between two linker generated symbols:

`__HEAP_START` has to be located at the start of the heap.

`__HEAP_END` is the last address of the heap area.

These two symbols have to be defined in the linker description file. The total amount of the heap are all bytes located between `__HEAP_START` and `__HEAP_END`. The heap therefore has a fixed size which may be modified by a constant in the linker description file.

Chapter 8

Stacks

8.1 Task stack for Freescale Coldfire V1

All embOS tasks execute in *supervisor mode* using the supervisor stack pointer. The stack-size required is the sum of the stack-size of all routines plus 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 embOS-routines.

For the Freescale Coldfire CPU, this minimum task stack size is about 72 bytes. But because any function call uses some amount of stack and every exception also pushes at least 12 bytes onto the current stack, the task stack size has to be large enough to handle all nested exceptions too. We recommend at least 256 bytes stack as a start.

8.2 System stack for Freescale Coldfire V1

The embOS system executes in *supervisor mode*, the scheduler also executes in *supervisor mode*. The minimum system stack size required by embOS is about 176 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 "C"-level interrupt handlers also use the system-stack, the actual stack requirements depend on the application. We recommend a stack size of 512 bytes.

For stack checking, embOS needs to know the size and the location of the stack.

The stack has to be declared in the linker file, using the symbols `___SP_END` and `___SP_INIT`.

8.3 Interrupt stack for Freescale Coldfire V1

If a normal hardware exception occurs, the Coldfire V1 CPU uses the supervisor stack as interrupt stack. Take care that the supervisor stack is large enough to handle all nested interrupts.

Chapter 9

Interrupts

The Freescale Coldfire V1 core comes with one or more built in vectored interrupt controllers which support up to 64 separate interrupt sources each. The real number of interrupt sources depends on the specific target CPU.

9.1 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 executed
- 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 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.

9.2 Defining interrupt handlers in C

Interrupt handlers for Coldfire V1 used with embOS are written as normal "C"-functions which do not take parameters and do not return any value.

All these interrupt handler are called from the high level interrupt service routine `OS_irq_handler()` in the CPU specific RTOSInit file.

Interrupts are nestable per default using the priority controlled interrupt handling of the Coldfire vectored interrupt controller.

If you wish to use non nestable interrupts, you may define the compile time switch `ALLOW_NESTED_INTERRUPTS` in the CPU specific RTOSInit file to 0.

Example

Simple interrupt routine:

```
void OS_ISR_TickHandler(void) {
  TPM1C0SC &= ~TPM1C0SC_CH0F;  // Reset compare interrupt request flag
  OS_HandleTick();
}
```

9.3 Interrupt vector table

After Reset, the Freescale Coldfire V1 CPU uses an initial interrupt vector table which is located in ROM at address 0x00. It contains the initial address for the main stack pointer and addresses for all exceptions. The interrupt vector table is located in the C file exceptions.c in the CPU specific subfolder. For all interrupts a separate interrupt handler table in RAM is used. This allows to enable dynamic interrupt handler installation with `OS_EnableISR()/OS_DisableISR()`.

9.4 Zero latency interrupts

9.4.1 Zero latency interrupts with Coldfire

Since Coldfire V1 has fixed interrupt priorities only, embOS has to disable interrupts by setting the CPUs interrupt priority to the highest priority of 7. Therefore zero latency interrupts are not supported.

9.5 Interrupt handling with vectored interrupt controller

For Coldfire V1, which has a built in vectored interrupt controller, embOS delivers additional functions to install and setup interrupt handler functions. To handle interrupts with the vectored interrupt controller, embOS offers the following functions:

9.5.1 OS_EnableISR(): Enable specific interrupt

Description

`OS_EnableISR()` is used to install a specific interrupt vector when Coldfire V1 CPUs with interrupt controller are used.

Prototype

```
OS_ISR_HANDLER* OS_EnableISR(int ISRIndex, OS_ISR_HANDLER* pISRHandler)
```

Parameter	Description
ISRIndex	Index of the interrupt source which should be enabled.
pISRHandler	Address of the interrupt vector function.

Return value

`OS_ISR_HANDLER*`: the address of the previous installed interrupt function which was installed at the addressed vector number before.

9.5.2 OS_DisableISR(): Disable specific interrupt

Description

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

Prototype

```
void OS_DisableISR(int ISRIndex)
```

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

Return value

None.

Additional Information

This function replaces the installed interrupt handler in the vector table by a default interrupt handler function. It does not disable the interrupt of any peripherals. This has to be done elsewhere.

9.5.3 High priority non maskable exceptions

High priority non maskable exceptions with non configurable priority like Reset, NMI and HardFault can not be used with embOS functions. These exceptions are never disabled by embOS.

Never call any embOS function from an exception handler of one of these exceptions.

Chapter 10

Technical data

This chapter lists technical data of embOS used with Coldfire V1 CPUs.

10.1 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	32
Resource semaphore	16
Counting semaphore	8
Mailbox	24
Software timer	20

Index

