embOS

Real Time Operating System

CPU & Compiler specifics for

Freescale Coldfire core

using IAR workbench

Document Rev. 2

SEGGER

A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Contents

1. About this document

This guide describes how to use ***embOS*** Real Time Operating System for the Freescale Coldfire series of microcontroller using *IAR Workbench*.

1.1. How to use this manual

This manual describes all CPU and compiler specifics of ***embOS*** using Freescale Coldfire based controllers with *IAR Workbench*. Before actually using ***embOS***, you should read or at least glance through this manual in order to become familiar with the software.
Chapter 2 gives you a step-by-step introduction, how to install and use ***embOS*** for Freescale Coldfire using *IAR Workbench.* If you have no experience using ***embOS***, you should follow this introduction, because it is the easiest way to learn how to use ***embOS*** in your application.
Most of the other chapters in this document are intended to provide you with important detailed information about functionality and fine-tuning of ***embOS*** for the Freescale Coldfire based controllers using *IAR Workbench*.

2. Using *embOS* with IAR Workbench

The following chapter describes how to start with and use ***embOS*** for Freescale Coldfire and *IAR Workbench*. You should follow these steps to become familiar with ***embOS*** for Freescale Coldfire and *IAR Workbench*

2.1. Installation

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

In order 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, please keep all files in their respective sub directories. Make sure the files are not read only after copying.
If you received a zip-file, please extract it to any folder of your choice, preserving the directory structure of the zip-file.

Assuming that you are using *IAR Workbench* project manager to develop your application, no further installation steps are required. You will find a prepared sample start application, which you should use and modify to write your application. So follow the instructions of the next chapter 'First steps'.

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

If for some reason you will not work with the project manager, 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 *IAR Workbench* project manager, it may be used without the project manager using batch files or a make utility without any problem.

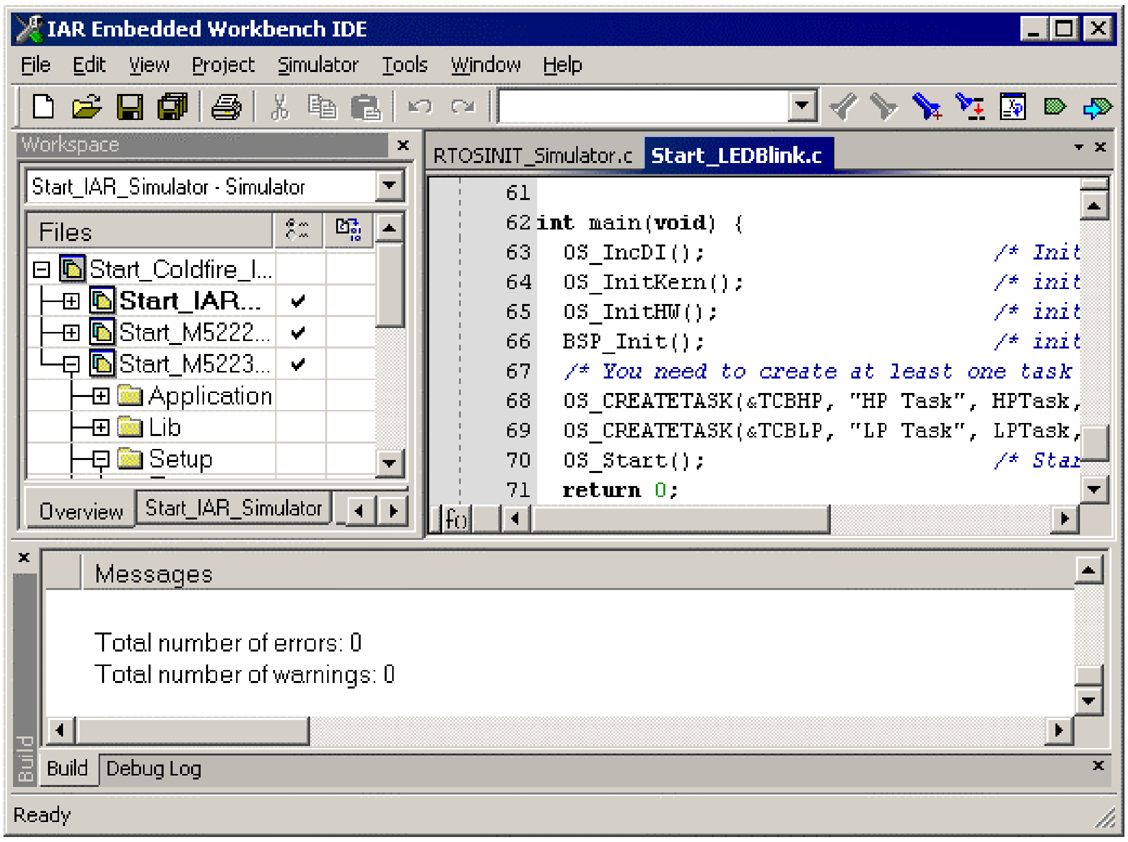
2.2. First steps

After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received ready to go sample start workspaces and projects and it is a good idea to use one of these as a starting point of all your applications.
Your ***embOS*** distribution contains one folder "Start" which contains the sample start workspaces and projects and every additional files used to build your application.

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 a sample workspace start\start_*.eww with *IAR Workbench* project manager (e.g. by double clicking it). We used Start_M52233DEMO.eww for our documentation.
- Build the start project

Your screen should look like follows:



```
int main(void) {
  OS_IncDI();                                    /* Init
  OS_InitKern();                                 /* init
  OS_InitHW();                                   /* init
  BSP_Init();                                    /* init
  /* You need to create at least one task
  OS_CREATETASK(&TCBHP, "HP Task", HPTask,
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,
  OS_Start();                                    /* Star
  return 0;
```

```
Total number of errors: 0
Total number of warnings: 0
```

For latest information you should refer to the ReadMe.txt files in the start and CPU folders..

2.3. The sample application Start_LEDBlink.c

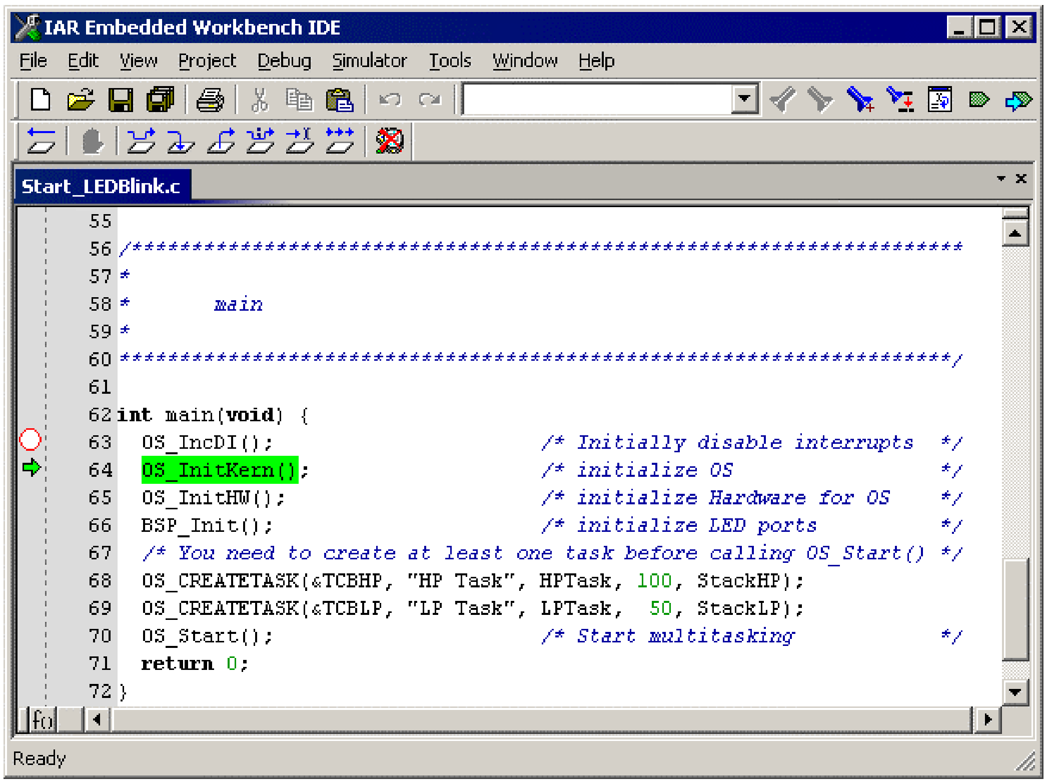
The following is a printout of the sample application Start_LEDBlink.c. It is a good starting-point for your application. (Please note that the file actually shipped with your port of ***embOS*** may look slightly different from this one)
What happens is easy to see:
After initialization of ***embOS;*** 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.

```
----------------------------------------------------------------------
File    : Start_LEDBlink.c
Purpose : Sample program for OS running on EVAL-boards with LEDs
--------- END-OF-HEADER ---------------------------------------------*/


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

OS_STACKPTR int StackHP[128], StackLP[128];          /* Task stacks */
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;
}
```

2.4. Stepping through the sample application using C-Spy

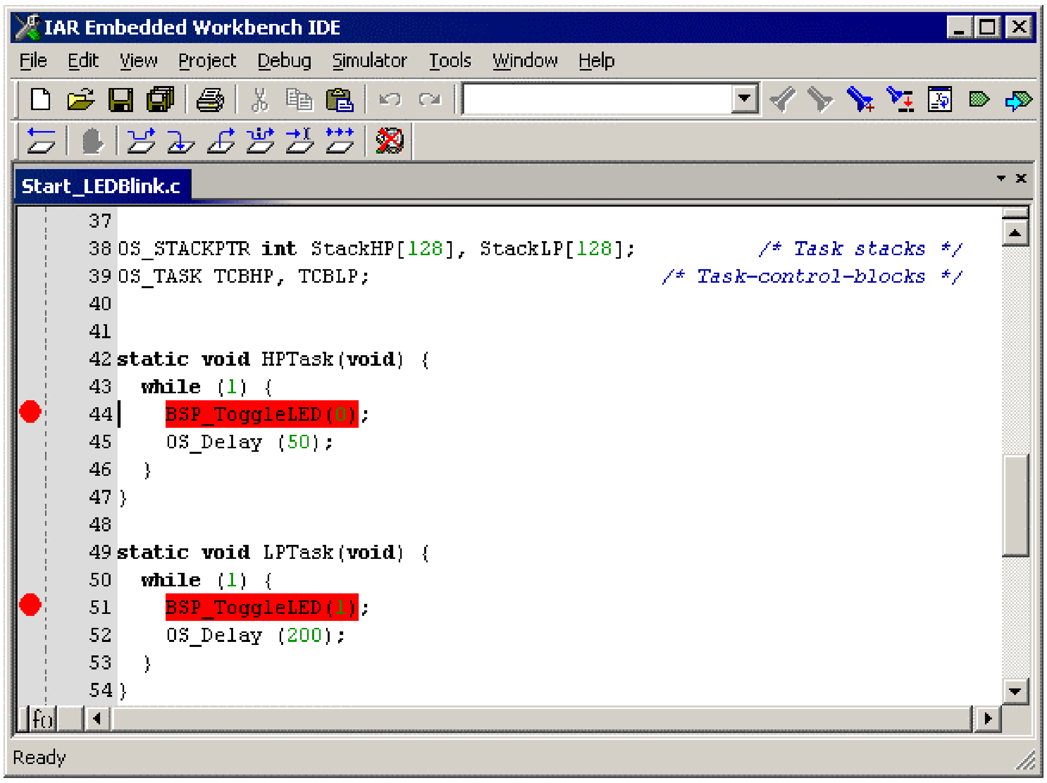
When starting the debugger, you will usually see the main function (very similar to the screenshot below). In some debuggers, you may look at the startup code and have to set a breakpoint at main. Now you can step through the program.
`OS_IncDI()` initially disables interrupts.
`OS_InitKern()` is part of the ***embOS*** library; 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, since it starts multitasking and does not return.


IAR Embedded Workbench IDE
File Edit View Project Debug Simulator Tools Window Help
Start_LEDBlink.c
Ready

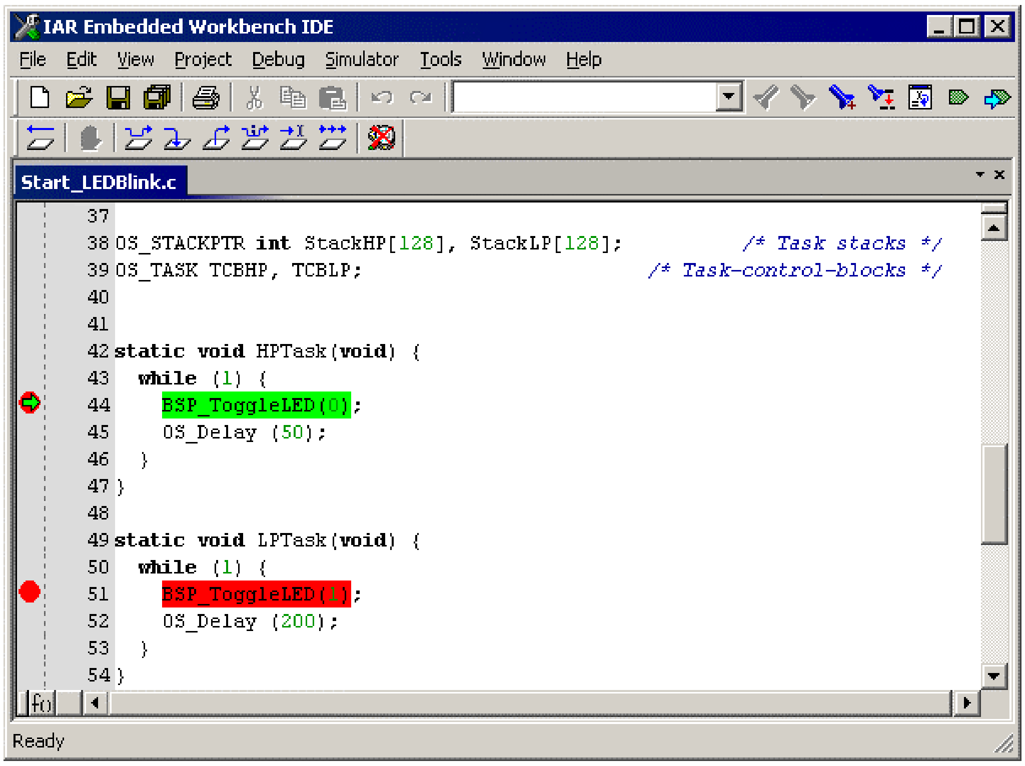
```
/***********************************************************************
*
*       main
*
***********************************************************************/

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;
}
```

Before you continue stepping, you should set two break points in the two tasks as shown below:


IAR Embedded Workbench IDE
File Edit View Project Debug Simulator Tools Window Help
Start_LEDBlink.c
Ready

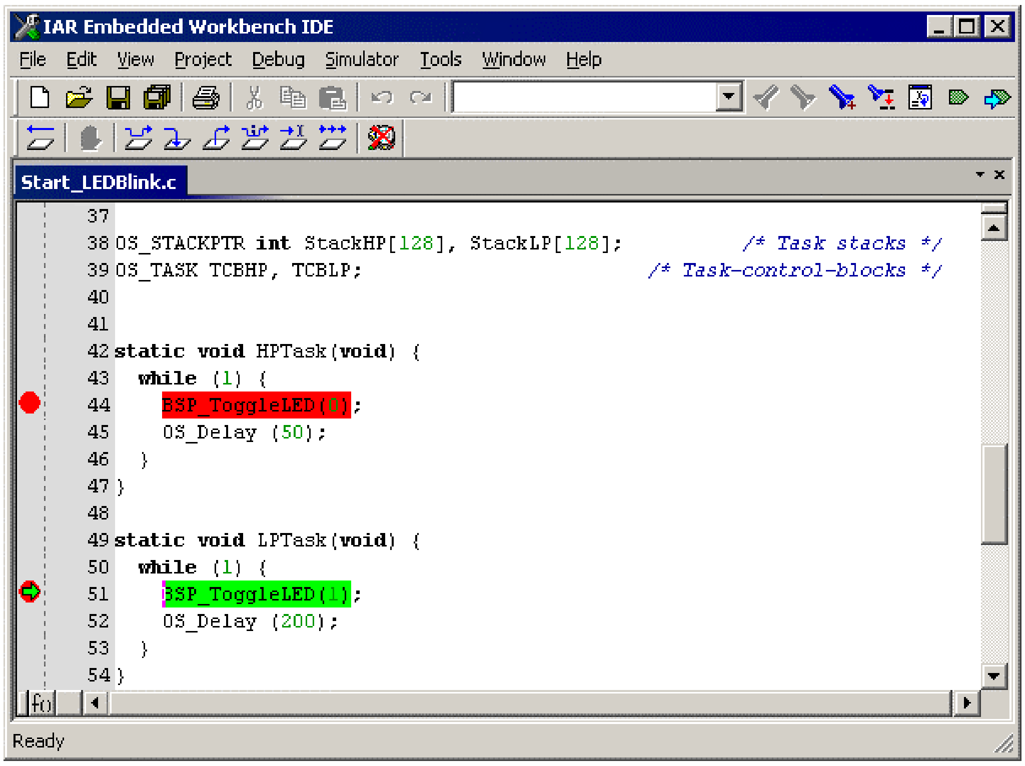
```

OS_STACKPTR int StackHP[128], StackLP[128];          /* Task stacks */
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);
  }
}
```

As `OS_Start()` is part of the ***embOS*** library, you can not step through it. You may press GO to reach the highest priority task.


IAR Embedded Workbench IDE
File Edit View Project Debug Simulator Tools Window Help
Start_LEDBlink.c
Ready

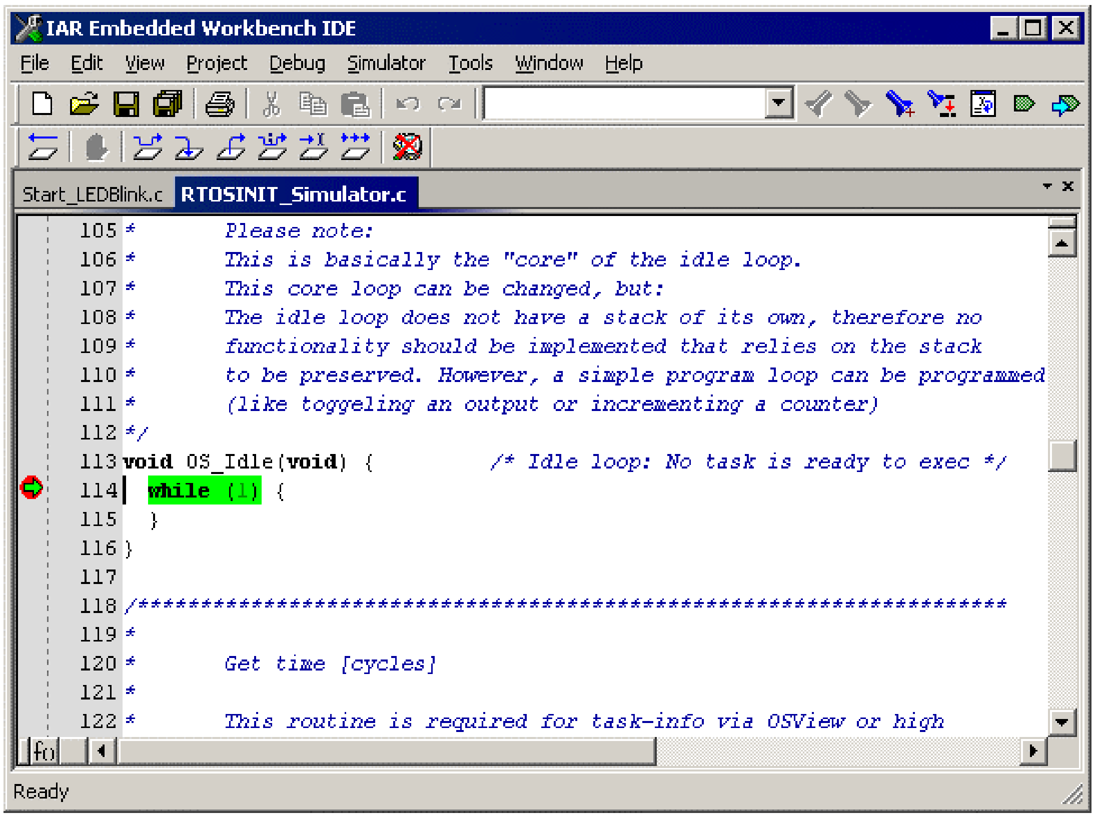
```

OS_STACKPTR int StackHP[128], StackLP[128];          /* Task stacks */
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);
  }
}
```

If you continue by pressing GO, you will arrive in the task with lower priority:


IAR Embedded Workbench IDE
File Edit View Project Debug Simulator Tools Window Help
Start_LEDBlink.c
Ready

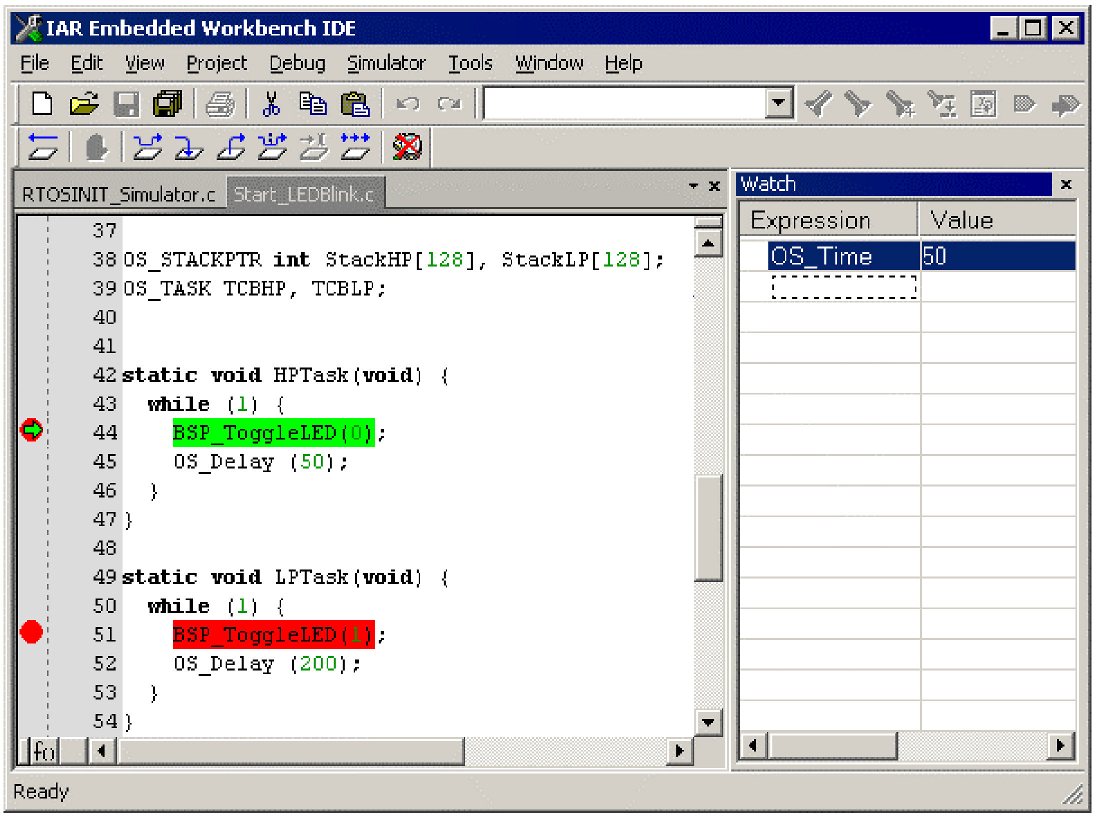
```

OS_STACKPTR int StackHP[128], StackLP[128];          /* Task stacks */
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);
  }
}
```

Continuing 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 set a breakpoint there before you continue:


IAR Embedded Workbench IDE
File Edit View Project Debug Simulator Tools Window Help
Start_LEDBlink.c RTOSINIT_Simulator.c
Ready

```
*       Please note:
*       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 toggeling an output or incrementing a counter)
*/
void OS_Idle(void) {          /* Idle loop: No task is ready to exec */
  while (1) {
  }
}

/*********************************************************************
*
*       Get time [cycles]
*
*       This routine is required for task-info via OSView or high
```

If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay. Press GO to enter the highest priority task again.
As can be seen by the value of **embOS** timer variable `OS_Time`, shown in the watch window, Task0 continues operation after expiration of the 50 ms delay.


IAR Embedded Workbench IDE
File Edit View Project Debug Simulator Tools Window Help
RTOSINIT_Simulator.c Start_LEDBlink.c
Watch
Ready

```

OS_STACKPTR int StackHP[128], StackLP[128];
OS_TASK TCBHP, TCBLP;


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

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

Expression	Value
OS_Time	50

3. Build your own application

To build your own application, you should always start with a copy of the sample start workspace and project. Therefore copy the entire folder “Start” from your ***embOS*** distribution into a working folder of your choice and then modify the start project there. This has the advantage, that all necessary files are included and all settings for the project are already done.

3.1. Required files for an *embOS* application

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 sub folder 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 CPU subfolder.
  It contains hardware dependent initialization code for ***embOS*** timer and optional UART for embOSView.
- One ***embOS*** **library** from the Lib\ subfolder
- **cstart*.s68** from the CPU specific Setup\ subfolder.
  It contains the interrupt vector table and default exception handler.
- **OS_Error.c** from subfolder Setup\ 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, please ensure that non initialized variables are initialized with zero, according to “C” standard. This is required for some ***embOS*** internal variables.
Also ensure, that main() is called with the CPU running in supervisor mode.
Your main() function has to initialize ***embOS*** by call of `OS_InitKern()` and `OS_InitHW()` prior any other ***embOS*** functions except `OS_IncDI()` are called.
You should then modify or replace the main.c source file in the subfolder src\.

3.2. Change library mode

For your application you may wish to choose an other 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 library is already contained in your project, just select the appropriate configuration or enable the library and disable others.
- To add a library, you may add a new embOSLib group to your project and add the new library to the new group. Exclude all other library groups from build, delete unused embOSLib groups or remove them from the configuration. Alternatively you may add the library to the predefined “Lib” group and exclude all other libraries from build.
- Check and set the appropriate OS_LIBMODE_* define as preprocessor option, or modify the OS_Config.h file.

3.3. Select an other CPU

embOS for Freescale Coldfire and IAR Workbench compiler contains CPU specific code for various Coldfire CPUs and starter kits. The sample start workspaces contain a project for a specific eval boards or starter kits.
Check whether your CPU and starter board is supported by ***embOS***. CPU specific functions are located in the subfolders of the start project folder.
To select a CPU which is already supported, just select the appropriate project from the start workspace.
If your CPU is currently not supported, examine all RTOSInit files in the CPU specific subfolders and select one which almost fits your CPU. You may have to modify OS_InitHW(), OS_COM_Init() and communication routines to embOS-View.

4. Freescale Coldfire specifics

4.1. CPU modes

embOS supports all code models and the far data memory model that IAR's C-compiler supports.
The near data memory model can be supported by recompiling the sources with appropriate compiler settings.

4.2. Available libraries

embOS for Freescale Coldfire for IAR Workbench is shipped with 56 different libraries for different CPU and code model options.
The naming convention of the ***embOS*** libraries follows the naming convention of the IAR runtime libraries.
The libraries are named as follows:

oscf<ISA><Code_Model><Data_Model><Lib_Config>_<LibMode>.r79

Parameter	Meaning	Values
ISA	Specifies the instruction set variant	a: ISA A (Coldfire V2) ap: ISA A+ (Coldfire V2) b: ISA B (V3, Future extension) c: ISA C (Coldfire V1)
Code_Model	Specifies the code model	n: near code model f: far code model
Data_Model	Specifies the data model	n: near data model f: far data model
Lib_Config	Specifies the runtime library variant	n: normal f: full (not needed for embOS)
LibMode	Library mode	xr: Release w/o round robin and task names
		r: Release
		ss: Stack check
		d: Debug
		sp: Stack check + profiling
		dp: Debug + profiling
		dt: Debug + trace

Example:

oscfapnfn_dp.r68 is the library for a project using ISA A+ instruction set, near code model, far data model, normal runtime libraries, with debug and profiling support.

5. Compiler specifics

5.1. Standard system libraries

embOS for Freescale Coldfire and IAR Workbench may be used with standard IAR Workbench 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.

6. Stacks

6.1. Task stack for Freescale Coldfire

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 32 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.

6.2. System stack for Freescale Coldfire

The ***embOS*** system executes in *supervisor mode*, the scheduler executes also in *supervisor mode*. The minimum system stack size required by ***embOS*** is about 136 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.

6.3. Interrupt stack for Freescale Coldfire

If a normal hardware exception occurs, the Coldfire CPU uses the current stack as interrupt stack. Be aware that your stacks are big enough to have place for the interrupt stack data.

7. Interrupts

The Freescale Coldfire core comes with an built in vectored interrupt controller which supports up to 57 separate interrupt sources. The real number of interrupt sources depends on the specific target CPU.

7.1. 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.

7.2. Interrupt vector table

After Reset, the Freescale 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 exceptions or fast interrupts 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 source file.

7.3. First level interrupt handler OS_ISR_Handler()

The first level interrupt handler `OS_ISR_Handler()` is the default interrupt handler for all peripheral interrupts and exceptions. It is inserted in the vector table in the CPU specific startup code for all exceptions and peripheral interrupts.
This first level interrupt handler acts as follows:

- Push temporary registers onto the stack
- Call `OS_EnterInterrupt()` to inform ***embOS*** that interrupt code is executed and disable low priority interrupts.
- Call the second level interrupt handler `OS_irq_Handler()` and pass the interrupt vector number as parameter.

- Call `OS_LeaveInterrupt()` when the second level interrupt returns to inform ***embOS*** that the interrupt is left, prepare a task switch if requested.
- Restores the saved registers.
- Execute a return from interrupt, restoring the previous processor state, the interrupted function continues, or the activated task starts execution.

Calling an ***embOS*** first level interrupt handler for every interrupt running on low priority is required for stable operation.
The first level interrupt handler `OS_ISR_Handler()`sets the interrupt priority level for Coldfire to the *Fast interrupt priority* level, thus disabling all low priority interrupts to avoid interrupt nesting.

7.4. First level interrupt handler OS_ISR_HandlerNestable()

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.
This first level interrupt handler acts as follows:

- Push temporary registers onto the stack.
- Call `OS_EnterNestableInterrupt()` to inform ***embOS*** that interrupt code is running and enable nested interrupts.
- Call the second level interrupt handler `OS_irq_Handler()` and pass the interrupt vector number as parameter.
- Call OS_LeaveNestableInterrupt() when the second level handler returns to inform embOS that the interrupt function is left, prepare a task switch, if requested
- Restores the saved registers
- Execute a return from interrupt, restoring the previous processor state, the interrupted function continues, or the activated task starts execution.

Calling an ***embOS*** first level interrupt handler for every interrupt running on low priority is required for stable operation.
The first level interrupt handler `OS_ISR_HandlerNestable()` does not modify the interrupt priority level of the Coldfire CPU, thus allows nested interrupts.

7.5. Second level interrupt handler OS_irq_handler()

The second level interrupt handler `OS_irq_handler()` is called from the first level interrupt handler and is part of the CPU specific RTOSInit source file and is therefore part of the application and may be modified if required.
The second level interrupt 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 source file and contains the vectors for all peripherals.

7.6. 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 file.

<u>Example of a peripheral interrupt handler</u>

"Simple" interrupt-routine

```
static void _ISR_TickHandler(void) {
  PIT0_PCSR |= PIT_PCSR_PIF;        /* reset Timer interrupt flag */
  OS_HandleTick();
}
```

7.7. Fast interrupts with Freescale Coldfire CPUs

Instead of disabling interrupts when ***embOS*** does atomic operations, the interrupt level of the CPU is set to the "*Fast interrupt priority limit*", which is set to 7 initially for Coldfire V1 cores and is 5 for other Coldfire cores. Therefore all interrupt priorities higher than the "*Fast interrupt priority limit*" can still be processed. Please note, that higher priority number defines a higher priority. All interrupts with a priority level above the *Fast Interrupt priority level* are never disabled.
These interrupts are named *Fast interrupts*. You must not execute any ***embOS*** function from within a *fast interrupt* function.
The *Fast interrupt priority limit* may be modified during runtime by calling the ***embOS*** function `OS_SetFastIntPriorityLimit()`.

7.8. Interrupt priorities

With introduction of *Fast 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 *Fast interrupt priority limit*.
- Any *Fast interrupt* (running at priorities above the Fast interrupt priority limit must not call any ***embOS*** API function.

7.9. OS_SetFastIntPriorityLimit(): Set the interrupt priority limit for fast 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***.

Description

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

Prototype

```
void OS_SetFastIntPriorityLimit(unsigned int Priority)
```

Parameter	Meaning
`Priority`	The highest value useable as priority for ***embOS*** interrupts. Interrupts with higher priority are never disabled by ***embOS***. Valid range: 1 <= `Priority` <= 7

Return value

NONE.

Add. 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 for Coldfire CPU.
To modify the default priority limit, `OS_SetFastIntPriorityLimit()` should be called before ***embOS*** was started.
In the default start projects, `OS_SetFastIntPriorityLimit()` is not called. The start projects use the default Fast interrupt priority limit.
Any interrupts running above the Fast interrupt priority limit must not call any ***embOS*** function.

7.10. Interrupt handling with vectored interrupt controller

For Coldfire, 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:

7.10.1. OS_EnableISR(): Install an interrupt handler

Description

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

Prototype

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

Parameter	Meaning
`ISRIndex`	Index of the interrupt source, normally the interrupt vector number.
`pISRHandler`	Address of the interrupt handler function.
`Prio`	Interrupt priority, NOT for Coldfire V1
`SubPrio`	Interrupt sub priority, NOT for Coldfire V1

Return value

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

Add. information

This function installs the interrupt vector and modify the priority and automatically enable the interrupt in the interrupt controller by setting the interrupt mask.
When the interrupt handler table is located in ROM, the vector can not be modified, the function sets the priority only.
Coldfire V1 cores do not support variable interrupt priorities. The two parameter Prio and SubPrio do not exist in ***embOS*** for Coldfire V1 cores.

7.10.2. 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	Meaning
`ISRIndex`	Index of the interrupt source which should be enabled.

Return value

NONE.

Add. 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.

7.11. 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.

8. STOP / WAIT Mode

In case your controller supports some kind of power saving mode, it should be possible to use it also with ***embOS***, as long as the system timer keeps working and timer interrupts are processed. To enter that mode, you usually have to implement some special sequence in function OS_Idle(), which you can find in ***embOS*** module RTOSINIT.c.

9. Technical data

9.1. 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 kernel itself has a minimum ROM size requirement of about 1.700 bytes.
In the table below, you can find minimum RAM size for ***embOS*** resources. Please note, that sizes depend on selected ***embOS*** library mode; table below is for a release build.

embOS resource	RAM [bytes]
Task control block	32
Resource semaphore	8
Counting semaphore	4
Mailbox	20
Software timer	20

10. 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	`Start.*`	Sample workspace and project files for IAR Workbench.
START\INC	`RTOS.H`	Include file for ***embOS***, to be included in every "C"-file using ***embOS*** -functions
START\LIB	`os*.r79`	***embOS*** libraries
START\SRC	`main.c`	Sample frame program to serve as a start
START\SRC	`OS_Error.c`	***embOS*** runtime error handler used in stack check or debug builds
START\CPU_*	`*.*`	CPU specific hardware routines for various CPUs and starter boards.

Any additional files shipped serve as example.

11. Index