

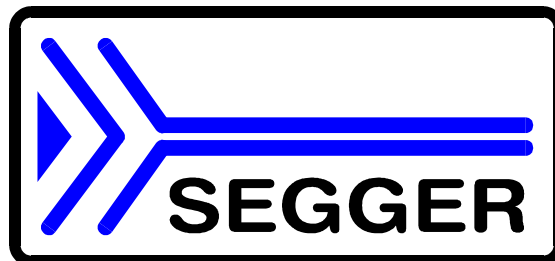
embOS

Zero latency
Real Time Operating System

CPU & Compiler specifics for
Fujitsu F²MC-16LX and FX series CPUs
Using Fujitsu Softune
workbench and compiler

Software version 3.86i
Document UM01034
Revision: 0

Date: May 10, 2012



A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Contents

Contents.....	3
1. About this document	4
1.1. How to use this manual.....	4
2. What is new?.....	4
2.1. Update / Upgrade information regarding interrupts.....	4
3. Using embOS with Softune workbench	5
3.1. Installation.....	5
3.2. First steps	6
3.3. The sample application Main.c	7
3.4. Stepping through the sample application Main.c using the Softune Simulator	7
4. Build your own application.....	11
4.1. Required files for an embOS application	11
4.2. Select a start project	11
4.3. Add your own code	11
4.4. Change library mode.....	11
4.5. Change memory model.....	12
4.6. Select an other CPU	12
4.7. Modify the startup-code	13
5. F ² MC-16LX compiler specifics	14
5.1. Memory models	14
5.2. Available libraries.....	14
6. Stacks	15
6.1. Task stack for F ² MC-16LX	15
6.2. F ² MC-16LX System (Interrupt) stack	15
6.3. User stack for F ² MC-16LX	16
6.4. Stack specifics of the Fujitsu F ² MC-16LX family.....	16
7. Dynamic memory, heap management	17
7.1. Heap memory definition and allocation.....	17
8. Interrupts	18
8.1. What happens when an interrupt occurs?	18
8.2. Zero latency, fast interrupts with F ² MC-16LX CPUs	18
8.3. Interrupt priorities with embOS for F16LX/FX CPUs.....	18
8.4. Defining interrupt handlers in "C"	19
8.5. OS_SetFastIntPriorityLimit()	20
8.6. Special considerations for the F ² MC-16LX / FX.....	20
9. STOP / WAIT Mode	21
10. Technical data.....	22
10.1. Memory requirements	22
11. Files shipped with embOS	22
12. Index	23

1. About this document

This guide describes how to use **embOS** with new zero latency interrupt handling for F²MC-16LX Real Time Operating System for the Fujitsu F²MC-16LX and F²MC-16FX series of microcontrollers using Softune workbench.

The stack specifics described in this manual refer to **embOS** version 3.22 or later which requires additional user stack defined in startup code.

1.1. How to use this manual

This manual describes all CPU and compiler specifics of **embOS** for F²MC-16LX and F²MC-16FX CPUs using Softune compiler. 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** using Softune 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 detailed information about functionality and fine-tuning of **embOS** for the F²MC-16LX using Softune compiler.

2. What is new?

- **OS heap management enabled**

Since version 3.84c of **embOS** for F²MC-16LX and FX CPUs, the thread safe heap management function `OS_malloc()`, `OS_free()` and `OS_realloc()` can be used.

The definition of the heap memory and the supporting `_sbrk()` function are delivered in the source file `sbrk.c`.

- **Zero latency, fast interrupts:**

Since version 3.22a.1 of **embOS** for F²MC-16LX and FX CPUs, interrupt handling inside **embOS** was modified. Instead of disabling interrupts when **embOS** does atomic operations, the interrupt level of the CPU is set to 2. Therefore all interrupts with level 1 or 0 can still be processed which results in zero latency.

Since version 3.60 of **embOS** for F²MC-16LX, the priority limit is not fixed and may be changed at runtime using `OS_SetFastIntPriorityLimit()`.

2.1. Update / Upgrade information regarding interrupts

When you update / upgrade from an **embOS** version prior 3.22a.1, you may have to change your interrupt handlers because of *Fast interrupt* support. All interrupt handlers using **embOS** functions have to run on priorities from 6 to 2.

Please read chapter “Interrupts” in this manual.

3. Using **embOS** with Softune workbench

3.1. 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 Softune workbench 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 workbench for your application development in order to become familiar with **embOS**.

If for some reason you will not work with the embedded workbench, 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 Softune workbench, it may be used without the workbench using batch files or a make utility without any problem.

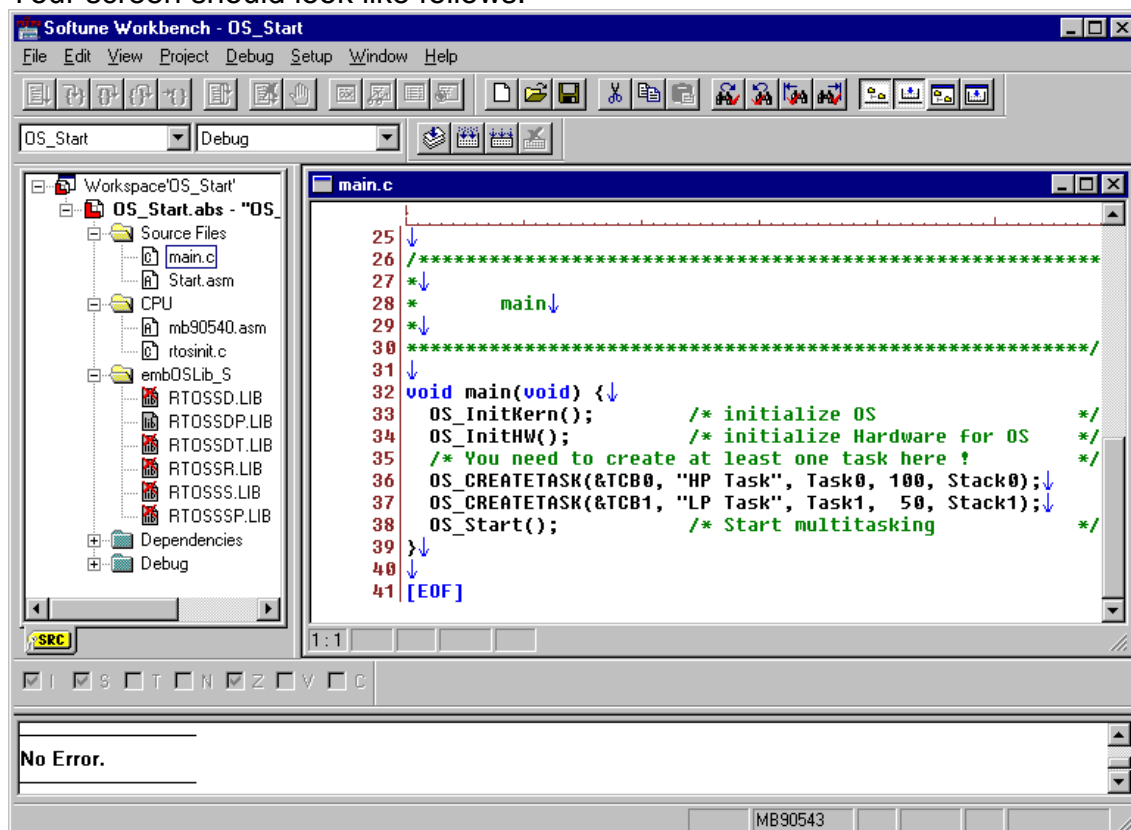
3.2. First steps

After installation of *embOS* (→ Installation) you are able to create your first multitasking application. You received a ready to go sample start project and it is a good idea to use this as a starting point of all your applications.

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 the sample start workspace \OS_Start.wsp with Softune workbench (e.g. by double clicking it)
- Build the start project

Your screen should look like follows:



3.3. The sample application Main.c

The following is a printout of the sample application main.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 2 tasks are activated and execute until they run into the delay, then suspend for the specified time and continue execution.

```

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

#include "RTOS.H"

OS_STACKPTR int Stack0[128], Stack1[128]; /* Task stacks */
OS_TASK TCB0, TCB1;                      /* Task-control-blocks */

void Task0(void) {
    while (1) {
        OS_Delay (10);
    }
}

void Task1(void) {
    while (1) {
        OS_Delay (50);
    }
}

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

int 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(&TCB0, "HP Task", Task0, 100, Stack0);
    OS_CREATETASK(&TCB1, "LP Task", Task1, 50, Stack1);
    OS_Start();           /* Start multitasking */
    return 0;
}

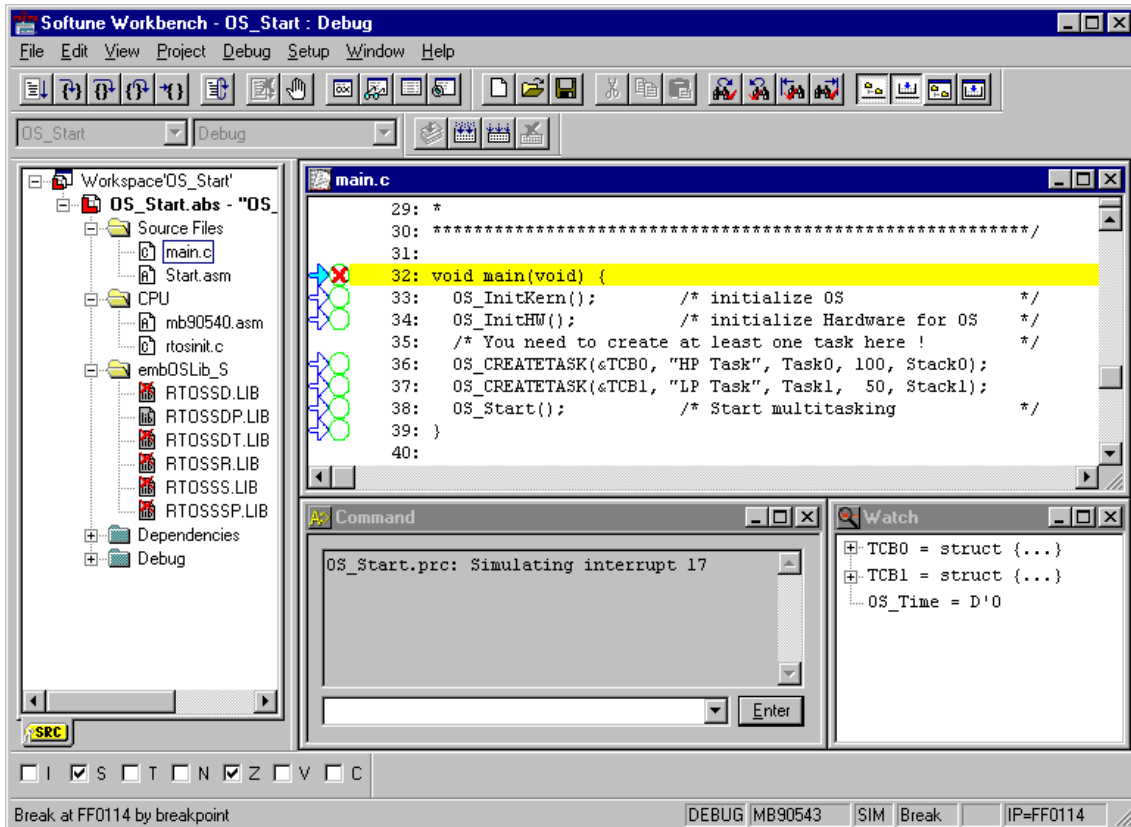
```

3.4. Stepping through the sample application Main.c using the Softune Simulator

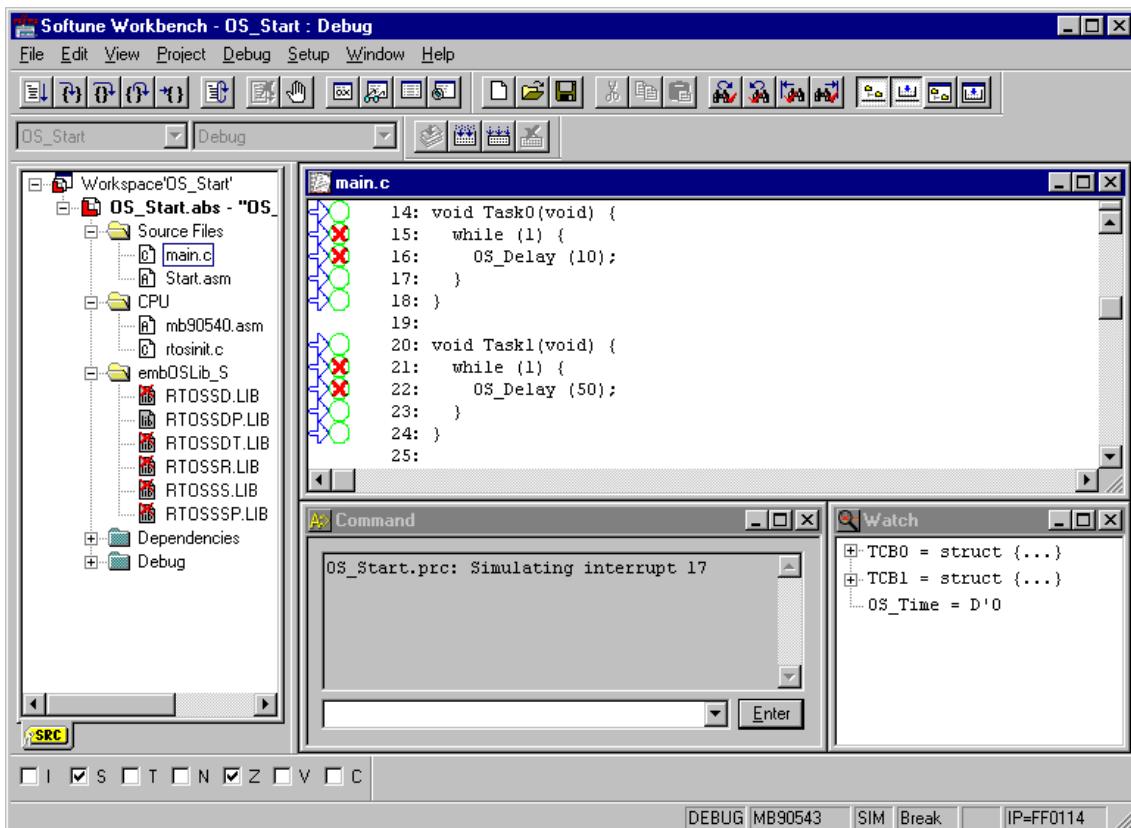
After compilation you should then start the simulator debugger from Softune workbench. This allows you to step through the program and see how the task switching between the two tasks work. The sample project is prepared, so that the timer interrupt, which is needed for delay, is simulated. This is done by the command file OS_Start.prc which is called after the simulator debugger is started.

After starting the simulator from menu 'Debug | Start Debug', the debugger command file automatically sets a breakpoint at main and starts execution.

The sample application runs its startup code and stops at main:



Before you start stepping through the program, you should set two additional breakpoints in the two tasks:

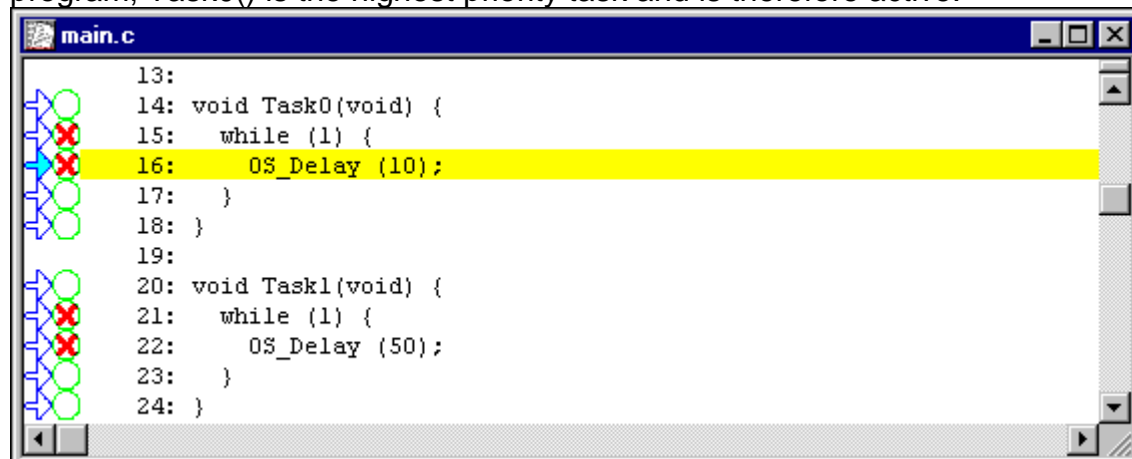


Now you can step through the program.

- OS_InitKern() is part of the **embOS** Library; you can therefore only step into it in disassembly mode. It initializes the relevant OS-Variables. It does not enable interrupts, when OS_IncDI() was called before.

- 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.

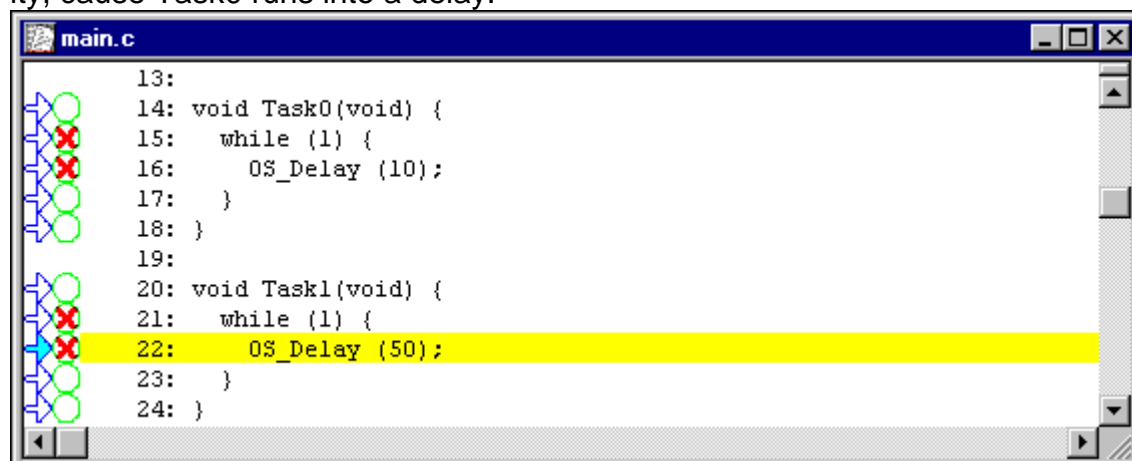
When you step into OS_Start(), the next line executed is already in the highest priority task created. (you may also use disassembly mode to get there of course, then stepping through the task switching process). In our small start program, Task0() is the highest priority task and is therefore active.



```

13:
14: void Task0(void) {
15:     while (1) {
16:         OS_Delay (10);
17:     }
18: }
19:
20: void Task1(void) {
21:     while (1) {
22:         OS_Delay (50);
23:     }
24: }
  
```

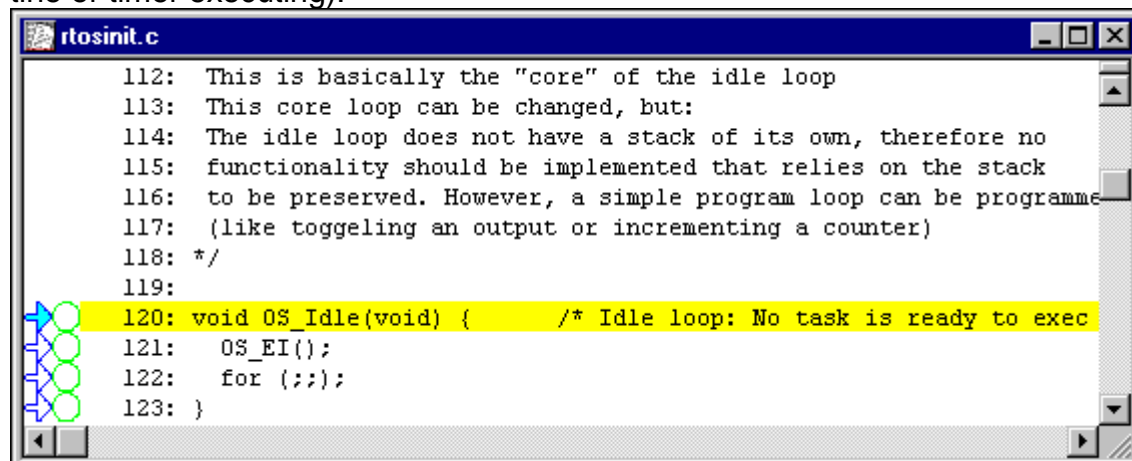
If you continue stepping, you will arrive in the task with the second highest priority, cause Task0 runs into a delay:



```

13:
14: void Task0(void) {
15:     while (1) {
16:         OS_Delay (10);
17:     }
18: }
19:
20: void Task1(void) {
21:     while (1) {
22:         OS_Delay (50);
23:     }
24: }
  
```

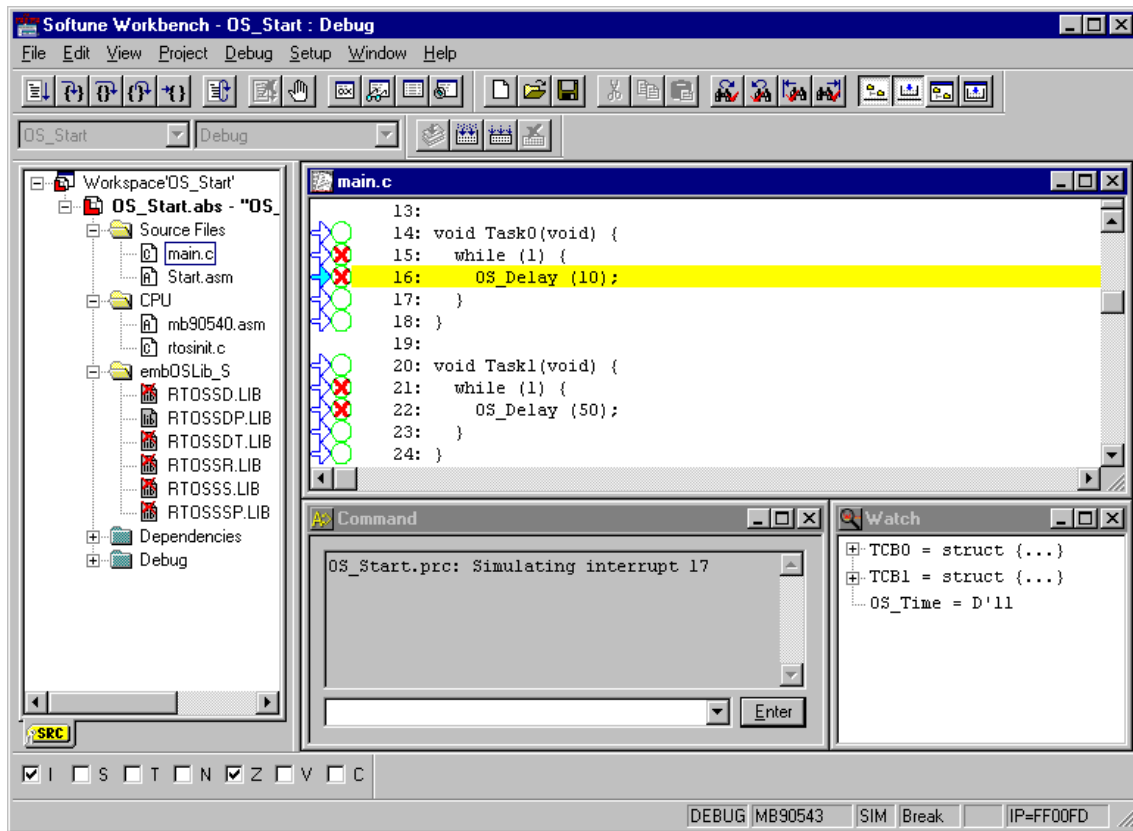
Continuing to step through the program, there is no other task ready for execution. *embOS* will therefore start the idle-loop, which is an endless loop which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).



```

112: This is basically the "core" of the idle loop
113: This core loop can be changed, but:
114: The idle loop does not have a stack of its own, therefore no
115: functionality should be implemented that relies on the stack
116: to be preserved. However, a simple program loop can be programmed
117: (like toggling an output or incrementing a counter)
118: */
119:
120: void OS_Idle(void) {      /* Idle loop: No task is ready to exec
121:     OS_EI();
122:     for (;;)
123: }
  
```

If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay. In the lower right corner you can see the system variable `OS_Time`, which shows how much time has expired in the target system.



4. Build your own application

To build your own application, you should start with the sample start workspace and start project. This has the advantage, that all necessary files are included and all settings for the project are already done.

4.1. 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 subfolder CPU\ or a CPU_* subfolder.
It contains the hardware dependent initialization code for the *embOS* timer and optional UART for *embOSView*.
- **OS_Error.c** from subfolder Src\
It contains the *embOS* error handler which is used in stack check or debug libraries. When a release version of libraries is used, this file is not required.
- **mb*.asm** from subfolder CPU\ or a CPU_* subfolder.
It contains special function register addresses used for the selected CPU.
- **start.asm** from subfolder Src\ for F16LX CPUs, or START_MB96F348.asm from the subfolder CPU_MB96F348 for F16FX CPUs.
It contains the startup code which runs and initializes the CPU after reset.
- One *embOS* library from the Lib\ subfolder

When you decide to write your own startup code, 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 your startup code exports `SSTACK_BASE`, `SSTACK_TOP`, `USTACK_BASE` and `USTACK_TOP` which are needed for stack checking functions of *embOS*.

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.

4.2. Select a start project

embOS comes with one start project for F16LX CPUs and one start project for F16FX CPUs which include different configurations for Release and Debug output. The start project for F16LX was built and tested with an MB90F543 CPU.

The start project for F16FX CPUs was built and tested with an MB96F348 CPU. For other CPUs there may be modifications necessary as described later.

4.3. Add your own code

For your own code, you may add a new group to the project. You should then modify or replace the `main.c` source file in the subfolder `src\`.

4.4. Change library mode

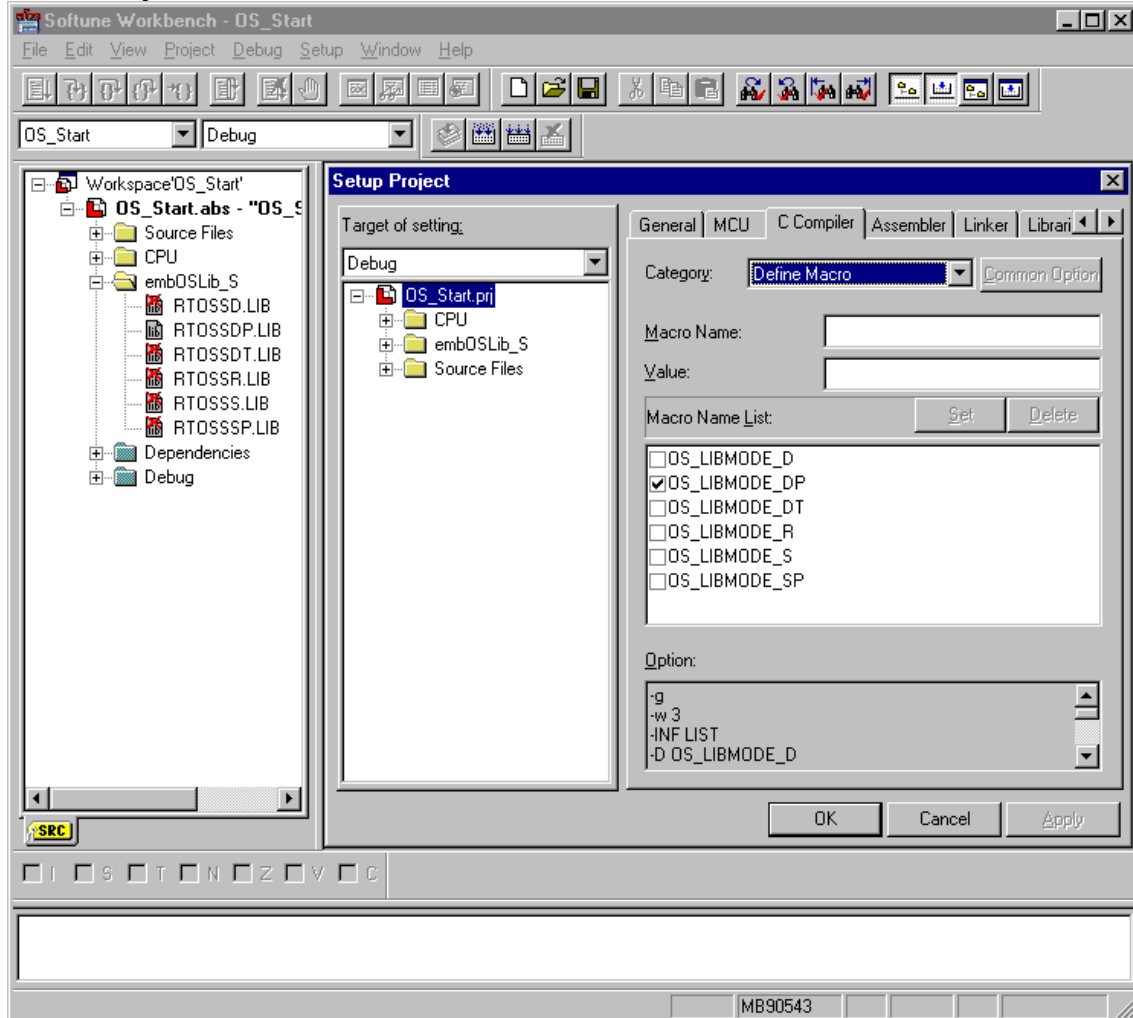
For your application you may wish to choose an other library. For debugging and program development you should use an *embOS* -debug library. For your

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:

- in the embOSLib group, exclude all libraries from build, except the one which should be used for your application.

Finally check project options about library mode setting according library mode used. Refer to chapter 4 about the library naming conventions to select the correct library.



4.5. Change memory model

The sample start project was built for small memory model. To change to an other memory model

- Select the desired memory model under “Project | Setup Project... | C Compiler | Category: Target Depend”.
- Include one or all **embOS** libraries which fit to the selected memory model. We recommend to add a group which contains all libraries of one memory model into your project.

4.6. Select an other CPU

- Select the desired CPU as project option under “Project | Setup Project... | MCU | Target MCU”.

- Copy all files from a CPU_* subdirectory into the CPU subdirectory of your start project folder. (For example: To switch MCU type to MB90F497, copy all files from subfolder CPU_90495 to subfolder CPU.

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(), communication routines for embOSView and also check the interrupt vector definitions in Rtosinit.c.

4.7. Modify the startup-code

When you decide to write your own startup code, or have to use a startup code not delivered with **embOS**, ensure that the stack segment end addresses are exported, as shown in the following excerpt of a startup file delivered with **embOS**:

```

;===== Modification for embOS, export symbols =====
                .EXPORT      SSTACK_BASE, SSTACK_TOP    ; required for embOS
                .EXPORT      USTACK_BASE, USTACK_TOP    ; required for embOS
;=====

```

These addresses are needed for the **embOS** stack check functions.

Also ensure that the CPU initializes both stack pointers and starts main() running up in user mode with the user stack selected.

The startup files coming from Fujitsu normally allow selection via a setting in the assembly startup file:

```

; ----- stack -----
#set          USRSTACK  0      ; use user stack, system stack for interrupts
#set          SYSSTACK  1      ; use system stack for all (program + inter)

#set          STACKUSE  USRSTACK ; USRSTACK, required for embOS since version 3.22
;
; ....
;
;=====
; Prepare stacks and set the default stack type
;=====
#macro SYSSTACKINI
    OR      CCR,#H'20                ; set System stack flag
    MOV     A,#BNKSYM SSTACK_TOP    ; System stack set
    MOV     SSB,A
    MOVW    A,#SSTACK_TOP
    MOVW    SP,A
#endm
#macro USRSTACKINI
    AND     CCR,#H'DF                ; User stack flag set
    MOV     A,#BNKSYM USTACK_TOP    ; User stack set
    MOV     USB,A
    MOVW    A,#USTACK_TOP
    MOVW    SP,A
#endm
#if STACKUSE == USRSTACK
    SYSSTACKINI
    USRSTACKINI                      ; finally user stack selected
#else
    USRSTACKINI
    SYSSTACKINI                      ; finally system stack selected
#endif

```

5. F²MC-16LX compiler specifics

5.1. Memory models

embOS for F²MC-16LX supports all memory models that Fujitsu's C-Compiler supports (SMALL, MEDIUM, COMPACT and LARGE).

5.2. Available libraries

embOS comes with all libraries for every combination of memory model and library mode. The files to use are:

Memorymodel	Library type	Library	define
SMALL	eXtreme Release	RTOSR	OS_LIBMODE_XR
SMALL	Release	RTOSR	OS_LIBMODE_R
SMALL	Stack-check	RTOSSS	OS_LIBMODE_S
SMALL	Stack-check + Profiling	RTOSSP	OS_LIBMODE_SP
SMALL	Debug	RTOSD	OS_LIBMODE_D
SMALL	Debug + Profiling	RTOSDP	OS_LIBMODE_DP
SMALL	Debug + Trace	RTOSDT	OS_LIBMODE_DT
MEDIUM	eXtreme Release	RTOSMR	OS_LIBMODE_XR
MEDIUM	Release	RTOSMR	OS_LIBMODE_R
MEDIUM	Stack-check	RTOSMS	OS_LIBMODE_S
MEDIUM	Stack-check + Profiling	RTOSMSP	OS_LIBMODE_SP
MEDIUM	Debug	RTOSMD	OS_LIBMODE_D
MEDIUM	Debug + Profiling	RTOSMDP	OS_LIBMODE_DP
MEDIUM	Debug + Trace	RTOSMDT	OS_LIBMODE_DT
COMPACT	eXtreme Release	RTOSCR	OS_LIBMODE_XR
COMPACT	Release	RTOSCR	OS_LIBMODE_R
COMPACT	Stack-check	RTOSCS	OS_LIBMODE_S
COMPACT	Stack-check + Profiling	RTOSCSP	OS_LIBMODE_SP
COMPACT	Debug	RTOSCD	OS_LIBMODE_D
COMPACT	Debug + Profiling	RTOSCDP	OS_LIBMODE_DP
COMPACT	Debug + Trace	RTOSCDT	OS_LIBMODE_DT
LARGE	eXtreme Release	RTOSLR	OS_LIBMODE_XR
LARGE	Release	RTOSLR	OS_LIBMODE_R
LARGE	Stack-check	RTOSLS	OS_LIBMODE_S
LARGE	Stack-check + Profiling	RTOSLSP	OS_LIBMODE_SP
LARGE	Debug	RTOSLD	OS_LIBMODE_D
LARGE	Debug + Profiling	RTOSLDP	OS_LIBMODE_DP
LARGE	Debug + Trace	RTOSLDT	OS_LIBMODE_DT

As can be seen from the table, the library names reflect the memory model and the library type.

When using Softune workbench, please check the following points:

- The memory model is set as Compiler option
- The **embOS** library which fits to the selected memory model, is part of your project (included in one group of your target) and is enabled for compilation.
- The appropriate define is set as compiler option for your project.

6. Stacks

6.1. Task stack for F²MC-16LX

The task stack-size required is the sum of the stack-size of all routines called by the tasks plus a basic stack size used to store the task context.

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 F²MC-16LX, this minimum stack size is about 50 bytes in the large memory model.

Every task has its individual stack which may be located in any RAM location. The task stack is addressed with the CPU's user stack pointer.

6.2. F²MC-16LX System (Interrupt) stack

The F²MC-16LX and F²MC-16FX CPUs have been designed with multitasking in mind; they have 2 stack-pointers, USP and SSP. The S-Flag selects the active stack-pointer. During execution of a task, the S-flag is cleared thereby selecting the user-stack-pointer. If an interrupt occurs, the F²MC-16LX sets the S-flag and switches to the system-stack-pointer automatically this way. The SSP is active when the interrupt function is entered and when the interrupt function is left.

embOS comes with a special interrupt handler which switches back to the default USER stack on entry and switches back to the interrupt stack when the interrupt handler is left.

This way, the interrupt does not use the stack of the task and the task stack-size does not have to be increased for interrupt-routines.

After reset, the F²MC-16LX CPU switches to its system stack. With *embOS*, since version 3.22, this stack has to be used for interrupts only. Therefore the startup code has to define a user stack and has to switch to user stack during startup.

As the interrupt stack is used for interrupt entry only, the required stack size depends on the interrupt nesting level only. For large code model and *embOS* debug libraries, every interrupt requires a maximum of 36 bytes on the interrupt stack. The total amount of stack used by interrupts can therefore be calculated by multiplying 36bytes by the maximum nesting level (depending on different interrupt priorities used).

The size of the interrupt stack is given as SSSIZE or STACK_SYS_SIZE in the startup files. Initially we define a stack size of 256 bytes and recommend a minimum of 128 bytes:

```

;===== Modification for embOS, USRSTACK required =====
SSSIZE      .EQU 256                ; <<< system stack size in words
#if STACKUSE == USRSTACK
USSIZE      .EQU 256                ; <<< user stack size, if used
#else
    #error "embOS requires STACKUSE set to USRSTACK"
#endif

```

6.3. User stack for F²MC-16LX

The F²MC-16LX and F²MC-16LX have 2 stack-pointers, USP and SSP. The S-Flag selects the active stack-pointer. During execution of a task, the S-flag is cleared thereby selecting the user-stack-pointer.

Since version 3.22, **embOS** also uses the user stack during startup, during the execution of main() and during execution of high level interrupt handler functions. Therefore a user stack has to be defined in the startup code as shown in the example above.

This default user stack is also used during execution of OS_Idle(), during internal scheduler operation and for **embOS** software timers.

The user stack size therefore depends on the library mode and the application. We initially set the user stack size to 256 bytes and recommend at least 128 bytes.

The size of the user stack is defined as USSIZE or STACK_USR_SIZE in the startup files.

When using your own startup, ensure that main() is called with user stack selected!

6.4. Stack specifics of the Fujitsu F²MC-16LX family

The Fujitsu F²MC-16LX family of microcontrollers can address 16 mega bytes of memory. Because it has 16-bit stack-pointers only, the stack is accessed by using additional bank registers. As the internal RAM of F²MC-16LX derivatives is relatively small, it is a good idea to use this RAM as interrupt stack and system stack.

Task stacks may reside in any RAM location, but have to reside in near memory, if small memory model is used.

For **embOS**, since version 3.22, it is necessary to setup your startup file to use USRSTACK as initial stack.

Normally you do not need a large user stack, because all tasks use their own individual stack. Refer to the START.asm which is shipped as sample.

7. Dynamic memory, heap management

The heap management functions of the standard system libraries are not thread safe, the heap can not be allocated from multiple tasks without any kind of mutual exclusion mechanism. Resource semaphores can be used to protect the heap against mutual preemptive access from different task.

embOS delivers the thread safe management functions `OS_malloc()`, `OS_free()` and `OS_realloc()` which handle the required heap locking automatically.

Since **embOS** version 3.84c of **embOS** for Fujitsu F2MC-16LX/FX, these functions can be used instead of the standard system library functions `malloc()`, `free()` and `realloc()`. The **embOS** heap management functions were disabled in previous versions of **embOS**.

Using the heap requires an additional helper function `_sbrk()` which is delivered with **embOS** in source form in the module `sbrk.c`.

7.1. Heap memory definition and allocation

The heap memory is defined as an array of characters in the source file `sbrk.c` which is delivered with **embOS**.

The size of the heap can be defined by the macro `HEAP_SIZE` either in the file itself, or as a preprocessor define in the project.

The file `sbrk.c` has to be included in every application which needs dynamic memory.

The `_sbrk()` function is called whenever the system library requests a new chunk from the heap, or when some portion at the end of the allocated memory is freed and given back to the heap.

```

/*****
 *
 *      SEGGER MICROCONTROLLER GmbH & Co KG
 *      Solutions for real time microcontroller applications
 *
 *****/
File   : sbrk.c
Purpose : Implementation of the _sbrk() function and heap memory
*/

/*****
 *
 *      Configuration
 *
 *****/
#ifndef  HEAP_SIZE
#define  HEAP_SIZE    1024
#endif

/*****
 *
 *      Local data
 *
 *****/
static long  brk_siz = 0;
static char  _heap[HEAP_SIZE];

/*****
 *
 *      sbrk()
 *
 *****/
extern char* sbrk(int size) {
    if (((brk_siz + size) > HEAP_SIZE) || ((brk_siz + size) < 0)) {
        return((char*)-1);
    }
    brk_siz += size;
    return(_heap + brk_siz - size);
}
/***** End Of File *****/

```

8. Interrupts

Interrupts are interruptions of a program caused by hardware. Normal interrupts are maskable and can occur at any time unless they are disabled with the CPU's disable-interrupt-instruction.

There are several good reasons for using interrupt-routines. They can respond very fast to external events like the status change on an input, the expiration of a hardware timer, reception or completion of transmission of a character via serial interface or other events.

8.1. What happens when an interrupt occurs?

- The CPU-core receives an interrupt request
- As soon as the interrupts are enabled and the processor interrupt priority level is below or equal to the priority level of the interrupting device, the interrupt is executed
- The CPU switches to the system stack
- The CPU saves PC, flags and bank registers on the stack
- The ILM is loaded with the priority of the interrupt thus blocking lower prioritized interrupts
- The CPU jumps to the address specified in the vector table for the interrupt service routine (ISR)
- ISR: save registers
- ISR: user-defined functionality
- ISR: restore registers
- ISR: Execute RETI command, restoring PC, Flags, bank registers and switching to the stack that was active before receiving the interrupt
- For details, please refer to the Fujitsu users manual.

IMPORTANT:

Fujitsu's F²MC-16LX and FX CPUs do not automatically disable interrupts, when an ISR is entered. Therefore nested interrupts are enabled per default.

8.2. Zero latency, fast interrupts with F²MC-16LX CPUs

Instead of disabling interrupts when **embOS** does atomic operations, the interrupt level of the CPU is set to an adjustable priority. All interrupts with higher priorities can still be processed. The default priority limit is set to 2, which means, all interrupts with priority 1 and 0 can still be processed, when interrupts are disabled by **embOS**.

These interrupts are named *Fast interrupts*. You must not execute any **embOS** function from within a *fast interrupt* function.

Please note, that the highest useable interrupt priority of interrupt handler calling **embOS** functions must not exceed the priority limit which can be adjusted during runtime using the function `OS_SetFastIntPriorityLimit()`.

8.3. Interrupt priorities with **embOS** for F16LX/FX CPUs

With introduction of *Fast interrupts*, interrupt priorities useable by the application are divided into two groups:

- Low priority interrupts with priorities from 6 to a user definable priority limit. These interrupts are called **embOS** interrupts.

- High priority interrupts with priorities above the user definable priority limit. These interrupts are called **Fast interrupts**.

Interrupt handler functions for both types have to follow the coding guidelines described in the following chapters.

The priority limit between *embOS* interrupts and Fast interrupts is predefined to a value of 2, allowing fast interrupts to run with high priorities of 0 and 1. The limit may be changed during system initialization by a call of `OS_SetFastIntPriorityLimit()`.

8.4. Defining interrupt handlers in "C"

Routines preceded by the keyword `__interrupt` save & restore the temporary registers and all registers they modify onto the stack and return with RETI.

The interrupt handler may be implemented in any source file.

The interrupt handler used by *embOS* are implemented in the CPU specific `RTOSInit_*.c` file.

Example of an *embOS* interrupt handler

embOS interrupt handler have to be used for interrupt sources running at all priorities up to the user definable interrupt priority level limit for fast interrupts.

```
__interrupt void OS_ISR_Tick (void) {
    OS_CallNestableISR(_ISR_Tick);
}
```

Any interrupt handler running at priorities from 1 to the selectable "Fast interrupt" priority limit has to be written according the code example above, regardless any other *embOS* API function is called.

The rules for an *embOS* interrupt handler are as follows:

- The *embOS* interrupt handler **must not define any local variables**.
- The *embOS* interrupt handler has to call `OS_CallISR()`, when interrupts should not be nested. It has to call `OS_CallNestableISR()`, when nesting should be allowed.
- **The interrupt handler must not perform any other operation, calculation or function call.** This has to be done by the local function called from `OS_CallISR()` or `OS_CallNestableISR()`.

Differences between `OS_CallISR()` and `OS_CallNestableISR()`

`OS_CallISR()` should be used as entry function in an *embOS* interrupt handler, when the corresponding interrupt should not be interrupted by another *embOS* interrupt. `OS_CallISR()` sets the interrupt priority of the CPU to the user definable "fast" interrupt priority level, thus locking any other *embOS* interrupt, Fast interrupts are not disabled.

`OS_CallNestableISR()` should be used as entry function in an *embOS* interrupt handler, when interruption by higher prioritized *embOS* interrupts should be allowed. `OS_CallNestableISR()` does not alter the interrupt priority of the CPU, thus keeping all interrupts with higher priority enabled.

Example of a *Fast interrupt* handler

Fast interrupt handler have to be used for interrupt sources running at priorities above the user definable interrupt priority limit.

They must not call any *embOS* function.

```
__interrupt void FastUserInterrupt (void) {
    unsigned long Count; // local variables are allowed
    Count = TPU_TCNT0;
    HandleCount(Count); // Any function call except embOS functions is allowed
}
```

The rules for a *Fast interrupt* handler are as follows:

- Local variables may be used.
- Other functions may be called.
- **embOS** functions must not be called, nor direct, neither indirect.
- The priority of the interrupt has to be above the user definable priority limit for fast interrupts.

8.5. OS_SetFastIntPriorityLimit()

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

Interrupts with higher priorities are never disabled by **embOS**.

The default interrupt priority limit for fast interrupts is set to 2, allowing interrupts with priorities of 6 to 2 be used with **embOS** functions. Interrupts with higher priorities are never disabled by **embOS** and must not call any **embOS** function. This priority limit may be changed by a call of OS_SetFastIntPriorityLimit().

Prototype

```
void OS_SetFastIntPriorityLimit (OS_UINT Prio);
```

Parameter	Meaning
Prio	The interrupt priority limit for fast interrupts. 0 <= Prio <= 5

Return value

Void.

Add. information

The Priority limit should not be set above 5, because 6 is the lowest interrupt priority used by **embOS** internal timer interrupt. Setting a priority limit of 6 would block all **embOS** interrupts.

The debug version of **embOS** checks whether the limit is above 5 and calls OS_Error(). The release version does not perform any checks.

8.6. Special considerations for the F²MC-16LX / FX

None.

9. STOP / WAIT Mode

Usage of the wait instruction is one possibility to save power consumption during idle times. If required, you may modify the `OS_Idle()` routine, which is part of the hardware dependent module.

The stop-mode works without a problem; however the real-time operating system is halted during the execution of the stop-instruction if the timer that the scheduler uses is supplied from the internal clock. With external clock, the scheduler keeps working.

10. Technical data

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 values in the table are for the default memory model.

Short description	ROM [byte]	RAM [byte]
Kernel	approx.1300	26
Event-management	< 200	---
Mailbox management	< 550	---
Single-byte mailbox management	< 300	---
Resource-semaphore management	< 250	---
Timer-management	< 250	---
Add. Task	---	18
Add. Semaphore	---	4
Add. Mailbox	---	12
Add. Timer	---	10
Power-management	---	---

11. 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	OS_Start.wsp	Sample workspace for Softune workbench
START	OS_Start.prj	Sample project for Softune workbench
START	OS_Start.dat	Additional settings for sample project
START	OS_Start.sup	Settings for Softune simulator
START	OS_Start.prc	Command file for Softune simulator, simulates timer interrupt
START\INC	RTOS.H	Include file for <i>embOS</i> , to be included in every "C"-file using <i>embOS</i> -functions
START\INC	_f16lxs.h	CPU identification. Translates Project CPU selection to CPU-Series definition.
START\LIB	RTOS*.LIB	<i>embOS</i> libraries
START\SRC	main.c	Sample frame program to serve as a start
START\SRC	sbrk.c	Heap memory definition and support.
START\SRC	START.ASM	Start up file, modified for <i>embOS</i>
START\CPU	mb90545.asm	Internal register defines of CPU
START\CPU	Rtosinit.C	CPU specific hardware routines used by <i>embOS</i> .
START\CPU_*	Rtosinit*.C mb90*.asm	CPU specific hardware routines for various CPUs.

12. Index

__interrupt.....	19	Interrupts	18	S	
_sbrk().....	17	L		sbrk.c	17
D		library mode	14	SSSIZE	15
Dynamic memory	17	M		SSTACK_BASE.....	11, 13
E		malloc()	17	SSTACK_TOP	11, 13
embOS interrupt.....	18	memory models	14	STACK_SYS_SIZE	15
F		memory requirements	22	STACK_USR_SIZE	16
Fast interrupt.....	18	O		Stacks	15
free().....	17	OS_CallISR()	19	Stop-mode	21
H		OS_CallNestableISR()	19	U	
Heap.....	17	OS_free().....	4, 17	USRSTACK	16
HEAP_SIZE.....	17	OS_InitHW().....	13	USSIZE	16
I		OS_malloc()	4, 17	USTACK_BASE	11, 13
Installation	5	OS_realloc()	4, 17	USTACK_TOP	11, 13
Interrupt priorities	18	OS_SetFastIntPriorityLimit()	20	W	
Interrupt, fast.....	18	R		Wait-mode	21
		realloc()	17	Z	
				Zero latency	18