embOS

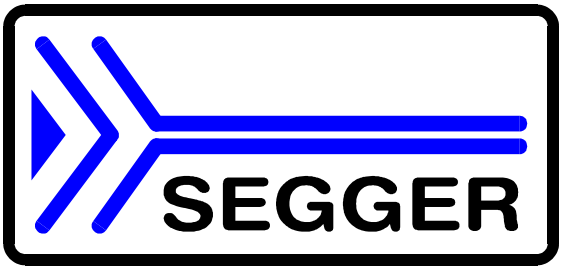
Zero latency
Real Time Operating System

CPU & Compiler specifics for

Fujitsu FR series CPUs

and Softune compiler V6

Document Rev. 3



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 Fujitsu FR series CPUs using Fujitsu compiler Fcc911s and Fujitsu Softune Workbench V6.

1.1. How to use this manual

This manual describes all CPU and compiler specifics for ***embOS*** for FR CPU series and Fujitsu Fcc911s compiler. Before actually using ***embOS***, you should read or at least glance through this manual in order to become familiar with the software.
Chapter 3 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, even if you do not plan to use Softune Workbench or Softune simulator, 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 FR CPU series using Fujitsu Fcc911s compiler.

2. What's new?

- **Zero latency, fast interrupts:**

Since version 3.22a of ***embOS*** for FR 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 20. Therefore all interrupts with level below twenty can still be processed which results in zero latency.

- **Selectable priority for Fast interrupts:**

Since version 3.60d of embOS for FR CPUs, the interrupt priority limit for fast interrupts can be modified during runtime.
Initially, the limit is set to 20, but may be modified by a call of the new function `OS_SetFastIntPriorityLimit()`.

2.1. Update / Upgrade information

When you update / upgrade from an ***embOS*** version prior 3.22a, 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 21 to 30.
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 prepared sample workspaces and start projects for different FR CPUs 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 Softune Workbench for your application development in order to become familiar with ***embOS.***

If for some reason you will not work with Softune Workbench, you should:
Copy either all or only the library-file that you need to your work-directory. Also copy all source and include files found in the start folder of your ***embOS*** distribution. 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.

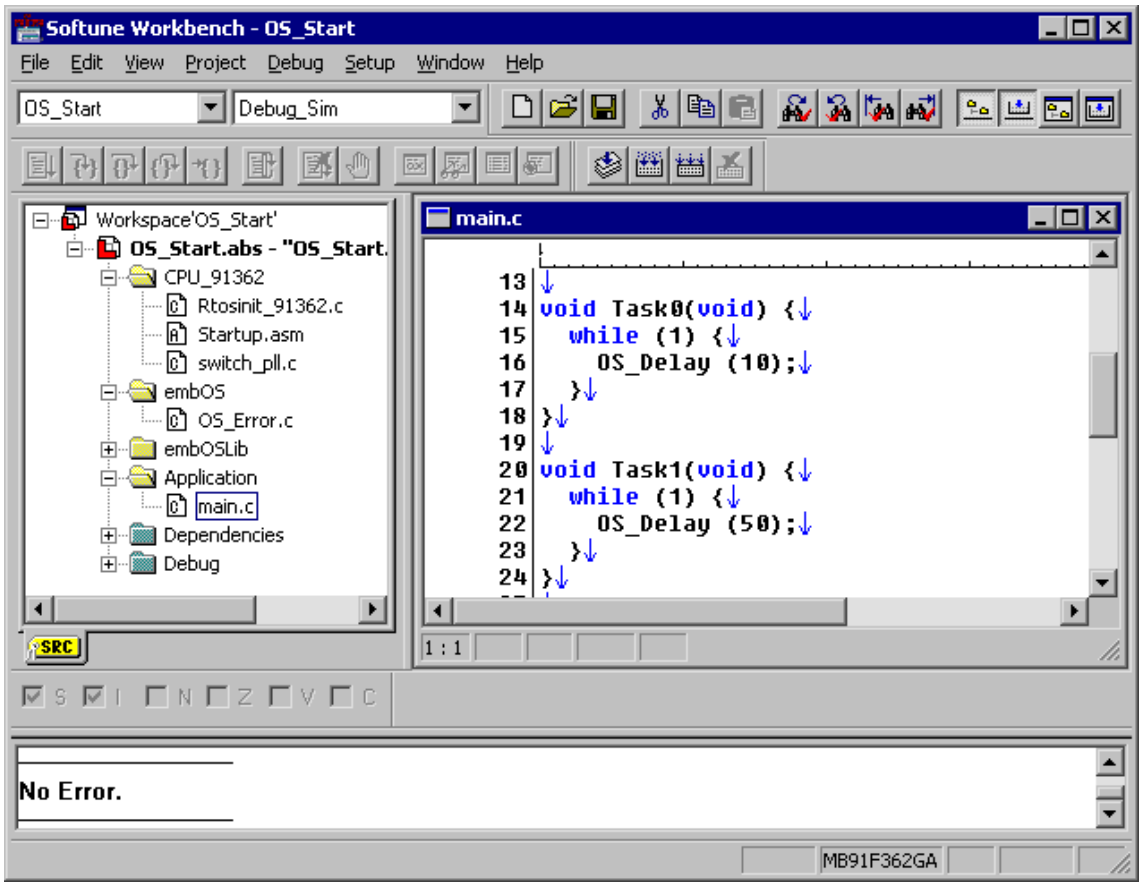
3.2. First steps

After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received ready to go sample workspaces and start projects for different CPUs. These workspaces are found in the CPU specific subfolders under “Start\BoardSupport\”. The following sample session is based on the startproject for an MB91362 CPU, located in the subdirectory “Start\BoradSupport\MB91362”. You may use an other workspace from an other subdirectory, if there is one available for your CPU.

The sample start project contains everything you need for the specific CPU.
As long as this CPU should be used, no further settings or modifications are required. When you would use an other CPU, you have to select an other start project, or have to modify the project as described later in this manual.
To get familiar with ***embOS*** you should first use the sample start project.

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’ from your ***embOS*** distribution into your work directory.
- Clear the read only attribute of all files in the new ‘Start’-folder in your working directory.
- Open the folder ‘Start\Boardsupport\MB91362’.
- Open the sample start workspace ‘Start_91362.wsp’ (e.g. by double clicking it). You may be ask to create the output folders for the application. Please accept the default settings.
- Build the start project



Please note:
The sample start project configuration “Debug_Sim” is selected per default and is set up for Softune debugger simulator. This was done to enable an easy start with ***embOS***.
The simulator needs a SIMULATOR define macro to disable initialization of PLL, because this can not be simulated and the simulator would be blocked in the startup code.
We modified the original sample startup code. The SIMULATOR define can be set as Project Tool Option for Assembler. The assembler startup file does not need to be modified.
The “Debug” configuration is prepared for your target. This configuration is prepared to initialize the target CPUs PLL during startup.

3.3. The sample application Main.c

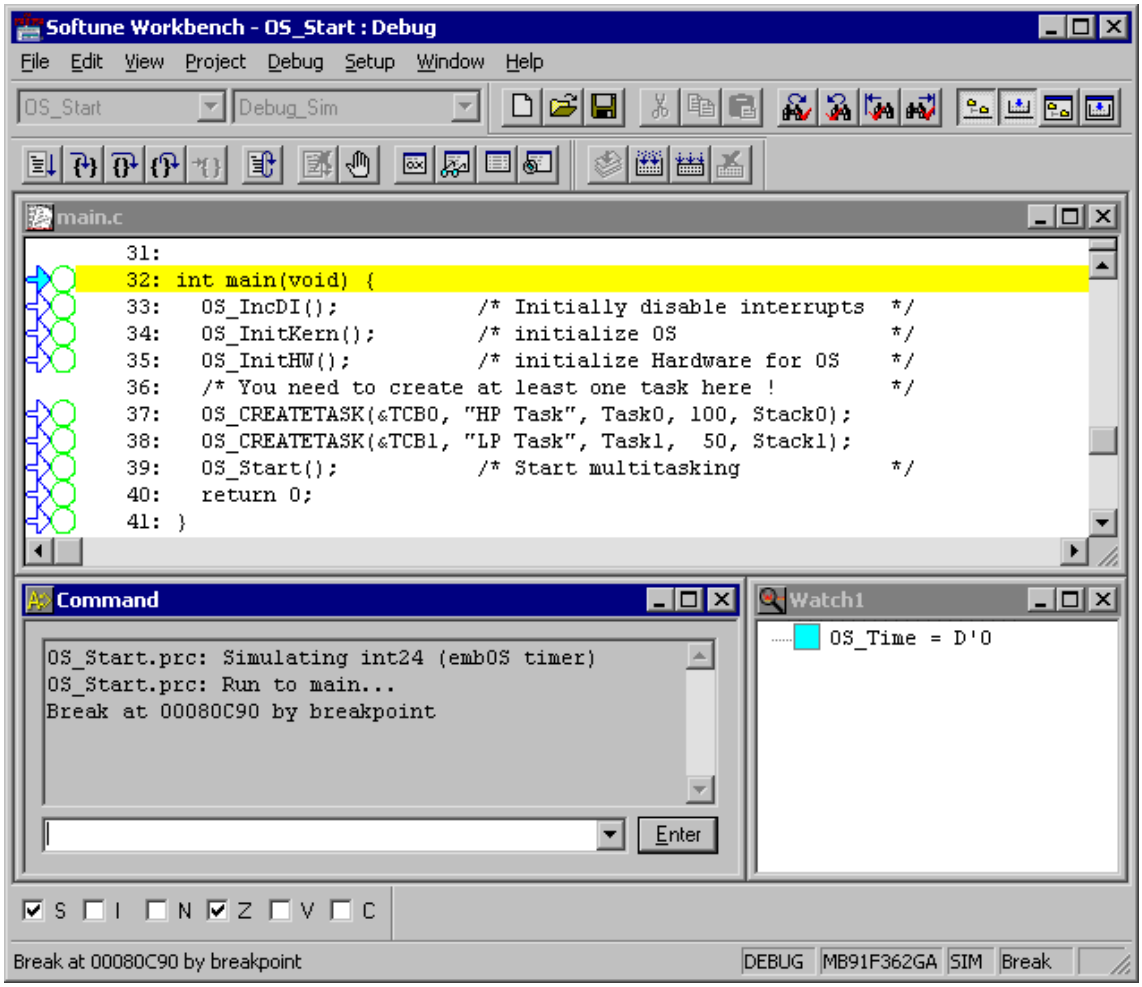
The following is a printout of the sample application main.c. It is a good starting-point for your application.
What happens is easy to see:
After initialization of ***embOS;*** two tasks are created and started
The two tasks are activated and execute until they run into the delay, then suspend for the specified time and continue execution.

```
/*********************************************************
*          SEGGER MICROCONTROLLER SYSTEME GmbH
*   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;
}
```

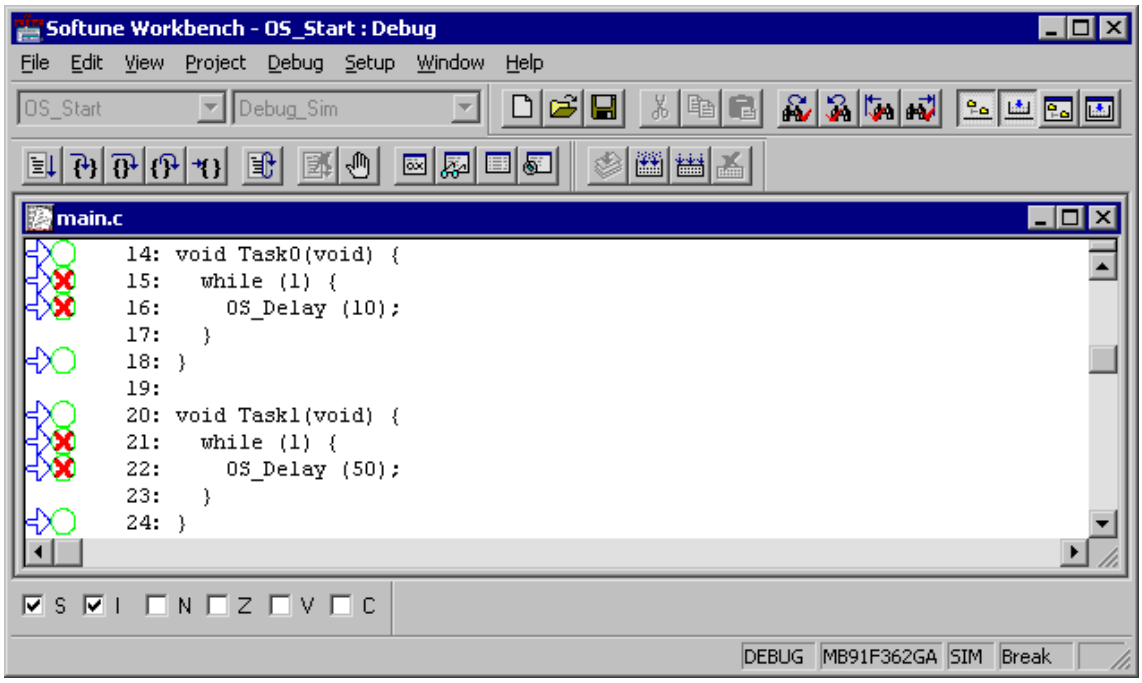
3.4. Stepping through the sample project using Softune simulator

When starting the Softune simulator debugger, you will usually see the main function (very similar to the screenshot below). The simulator procedure file Start_simulator.prc, required for ***embOS*** timer interrupt simulation, also automatically runs to main().
If you look at startup code after starting the simulator, please ensure that Start_simulator.prc is called after download of target file.
If simulator hangs and main is not reached, ensure that startup code is assembled with define SIMULATOR=1, this will inhibit PLL initialization in the simulation target.

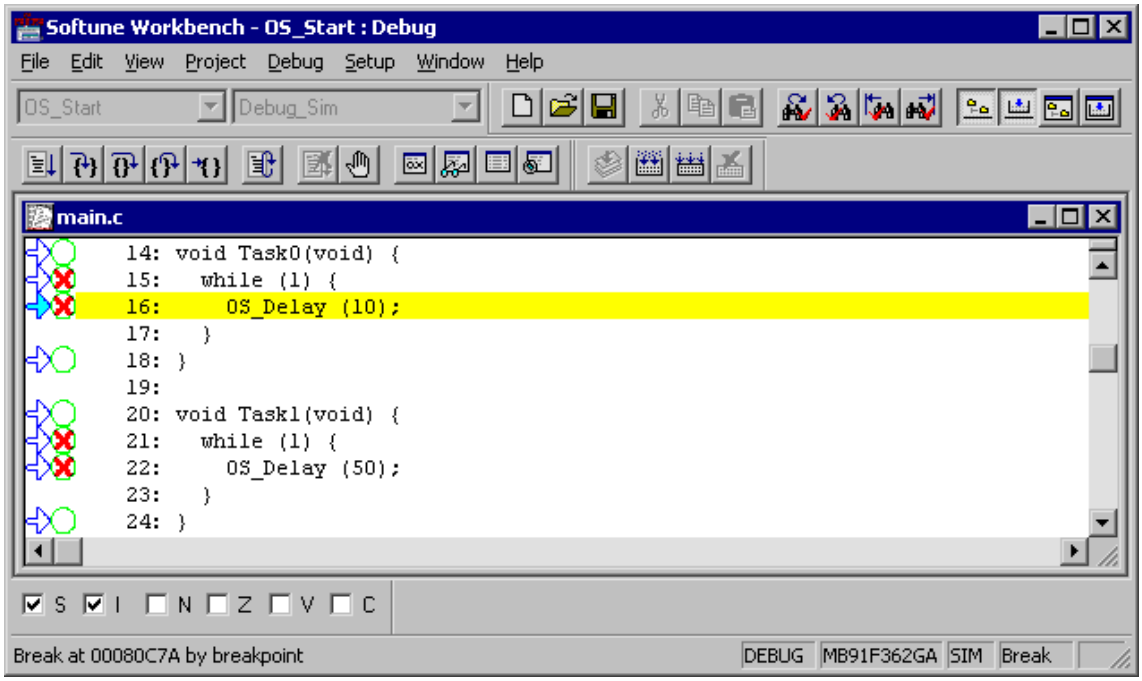
Now you can step through the program.
`OS_IncDI()` initially disables interrupts and avoids re-enabling of interrupts during the call of `OS_InitKern()`.
`OS_InitKern()` is part of the ***embOS*** Library; you can therefore only step into it in disassembly mode. It initializes the relevant ***embOS***-Variables and would enable interrupts if `OS_IncDI()`was not called before.
`OS_InitHW()` is part of CPU specific 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_COM_Init()` in `OS_InitHW()` is optional. It is required if embOSView shall be used. In this case it should initialize the UART used for communication. For the simulator configuration, UART communication is disabled, because it can not be simulated.
`OS_Start()` should be the last line in main, since it starts multitasking and does not return.



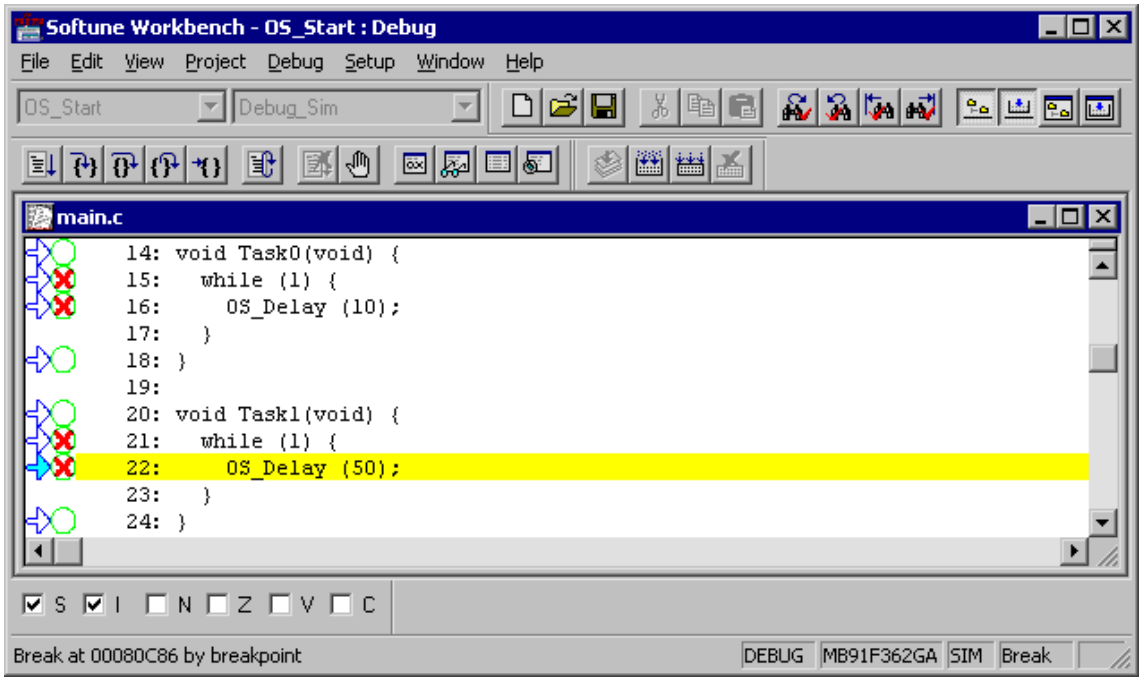
Before you step into `OS_Start()`, you should set breakpoints in the two tasks:



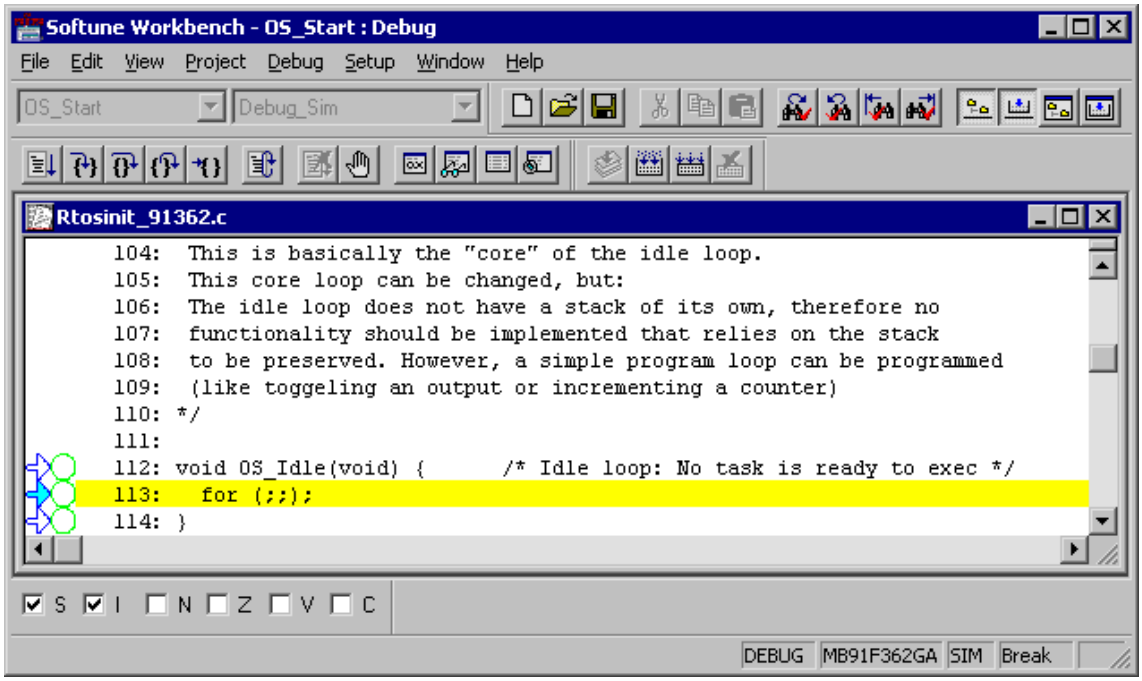
When you step over `OS_Start()`, the next line executed is already in the highest priority task created. (you may also step into `OS_Start()`, then stepping through the task switching process in disassembly mode). In our small start program, `Task0()` is the highest priority task and is therefore active.



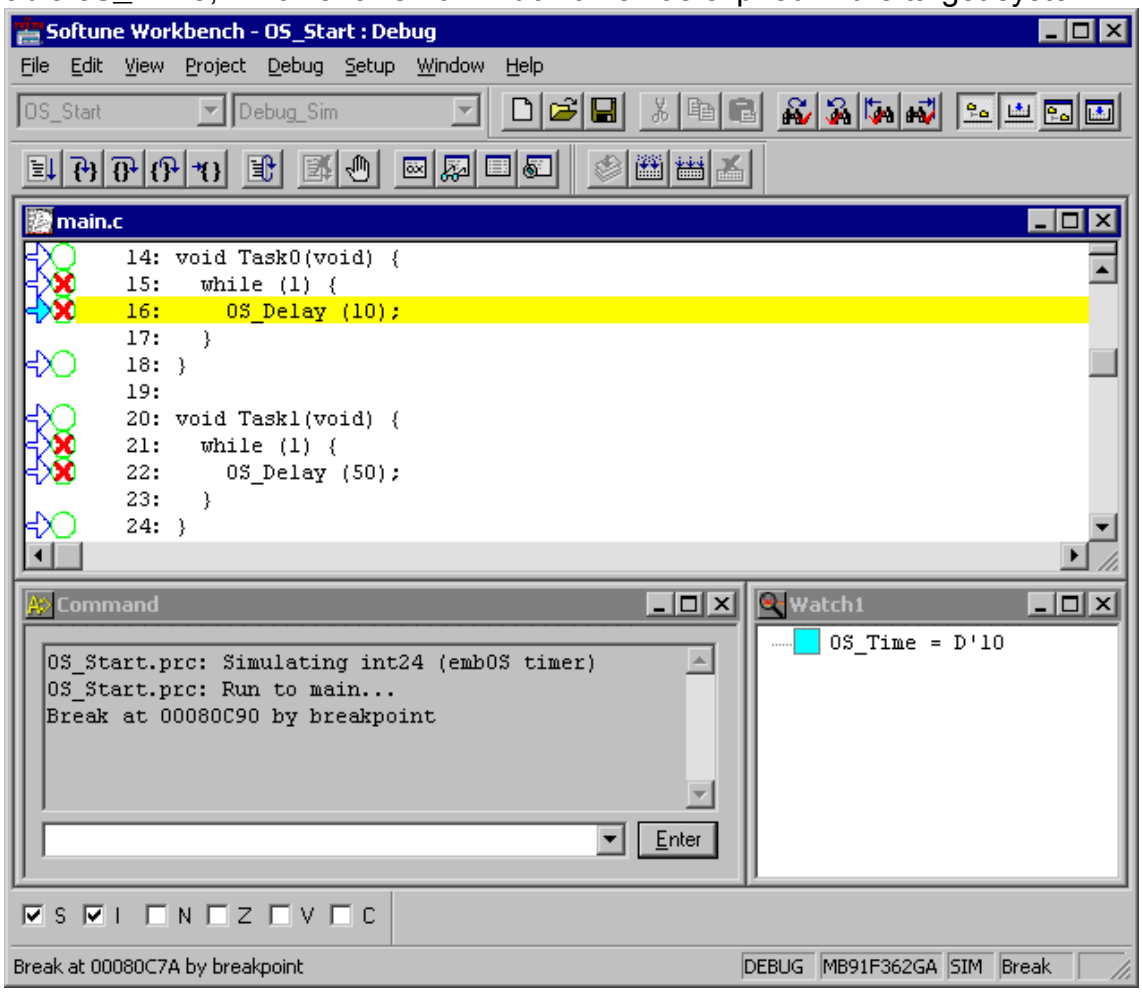
If you continue stepping, you will arrive in the task with the lower priority:



Continuing to step through the program, there is no other task ready for execution. ***embOS*** will suspend Task1 and switch to 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).
`OS_Idle()` is found in Rtosinit_*.c:



If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay.
Coming from `OS_Idle()`, execute the 'Go' command to arrive at the highest priority task after its delay is expired. The watch window shows the system variable `OS_Time`, which shows how much time has expired in the target system.



4. Project and compiler settings

4.1. Available libraries

embOS comes with different libraries with different debug and runtime error check capabilities.
The library files are located in the subfolder 'Lib' in the start project folder.
The library model settings for your target application have to confirm to the library used in your application.

The naming convention for library files is as follows:

RTOS<LIBRARYTYPE>.lib

<LIBRARYTYPE> specifies the type of ***embOS*** -library:

- **XR** stands for eXtreme release build library, which does not support round robin and also does not support task names.
- **R** stands for Release build library.
- **S** stands for Stack check library, which performs stack checks during runtime.
- **SP** stands for Stack check and Profiling library, which performs stack checking and additional runtime (Profiling) calculations
- **D** stands for Debug library which performs error checking during runtime.
- **DP** stands for Debug and Profiling library which performs error checking and additional Profiling during runtime.
- **DT** stands for Debug and Trace library which performs error checking and additional Trace functionality during runtime.

Example:

RTOSSP.lib is the ***embOS*** library with **S**tack check and **P**rofiling functionality.
It is located in the Start\lib\ subdirectory.

For FR CPUs, the following libraries are available (located in the subfolder 'Start\lib\):

Library type	Library	#define (OS_LIBMODE)
Exteme release	RtosXR.lib	OS_LIBMODE_XR
Release	RtosR.lib	OS_LIBMODE_R
Stack-check	RtosS. Lib	OS_LIBMODE_S
Stack-check + Profiling	RtosSP. Lib	OS_LIBMODE_SP
Debug	RtosD. Lib	OS_LIBMODE_D
Debug + Profiling	RtosDP. Lib	OS_LIBMODE_DP
Debug + Profiling + Trace	RtosDT. Lib	OS_LIBMODE_DT

You have to add one library to your project. Ensure that the define for the library type used is set as Tool Option for compiler.
When using the Softune workbench, just check the define that corresponds to the library file used:
Select "Project | Setup Project | C/C++ Compiler | Define Macro".
Using batchfiles, pass the library mode define as parameter to your compiler.

5. Modify the start project for your application

The sample start workspaces and projects delivered with ***embOS*** are set up to run with the Softune simulator and may also contain a configuration for an incircuit emulator or just to run in the target CPU. This ensures an easy start as described in chapter 3 of this manual.
For your target application, you will have to add your own code, you may wish to use an other library or select an other CPU.
You should always use the start project as a starting point for your application, as all necessary files for ***embOS*** are included and specific settings are already prepared.

5.1. Choose an other library

For debugging purposes during program development you may wish to use a debug library. Later in your final application you may use the release library.

To use an other library, proceed as follows:

- All ***embOS*** libraries are included in the project in the folder Lib. Just exclude all libraries from build, except the one you wish to use.
- Set the compiler macro define according to the library type used. Using Softune workbench select “Project | Setup Project | C/C++ Compiler | Category: Define Macro” to uncheck the previous OS_LIBMODE and then check the new library mode.

5.2. Switch between target CPU and Simulator

Simulator and target CPU may require different builds of startup code.
Our modified assembly startup files accepts a define passed to the assembler to select either a build for the simulator or the target CPU.
This define can be modified under “Project | Setup Project | Assembler | Category: Define Macro”.

- For simulator usage, check the define “SIMULATOR=1”
- For target CPU builds, uncheck “SIMULATOR=1”

If you delete the defined macro, the startup code defaults to SIMULATOR=0, thus builds startup code for the target CPU which initializes PLL.

5.3. Select an other CPU

The sample start project described under chapter 3 was built for an MB91F362 CPU. Specific files for other CPUs are located in the “Start\BoardSupport” directory.

To use a different CPU that is already supported, proceed as follows:

- Open the CPU specific subdirectory under “Start\BoardSupport”.
- Build the project

If your CPU is currently not supported, modify the start project:

- Copy any CPU specific folder and rename it according to your target CPU.
- Rename and modify Rtosinit_*.c and startup code in your new folder.
- Check and change project settings of your new project.
- Ensure that your project contains the files of your new CPU folder.

6. Stacks

6.1. Task stack for FR CPUs

Every task has to have its own stack. The task stack-size required is the sum of the stack-size of all routines called by the task plus a basic stack size.
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 FR CPU, this minimum stack size is about 128 bytes in release build.
Every task has its individual stack which may be located in any RAM location. The task stack is addressed with the CPUs user stack pointer.

6.2. FR CPU System (Interrupt) stack

The FR CPU series has been designed with multitasking in mind; it has 2 stack-pointers, USP and SSP. The S-Flag selects the active stack-pointer. During execution of a task, the S-flag is set, thereby selecting the user-stack-pointer. If an interrupt occurs, the FR CPU clears the S-flag and switches to the system-stack-pointer automatically this way. The SSP is active during the entire ISR (interrupt service routine). This way, the interrupt does not use the stack of the task and the stack-size does not have to be increased for interrupt-routines. Additional stack-switching as for other CPUs therefore is not necessary for the FR CPUs.
After reset, the FR 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 the user stack during startup.
The interrupt stack size required by ***embOS*** is about 128 bytes but varies with different library modes. However, since the system stack is also used by the application specific interrupts, the actual stack requirements depend on the application.
The size of the interrupt stack is defined in the startup file, given as SSTACK_SIZE . Initially we define a stack size of 512 bytes and recommend a minimum of 256 bytes:

```
#define SSTACK_SIZE 0x200   // define system (interrupt) stack size: 512 bytes
#define USTACK_SIZE 0x200   // define user stack size: 512 bytes

        .export __systemstack_top
        .export __systemstack
        .export __userstack_top
        .export __userstack

        .section        STACK, stack, align=4
__systemstack:
        .res.b          SSTACK_SIZE
__systemsstack_top:
__userstack:
        .res.b 0         USTACK_SIZE
__userstack_top:
```

6.3. FR CPU User stack

The FR has 2 stack-pointers, USP and SSP. The S-Flag selects the active stack-pointer. During execution of a task, the S-flag is set, thereby selecting the user-stack-pointer.

Since version 3.22, ***embOS*** uses the user stack during startup and execution of main(). 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 library mode and the application.
We recommend at least 256 bytes.
The size of the user stack is defined as USTACK_SIZE) in the startup file.

6.4. Stack specifics of the Fujitsu FR family

The Fujitsu FR family can use the whole memory area as stack, therefore stacks may reside in any RAM location.
For ***embOS***, since version 3.22, it is necessary to setup your startup file to use the user stack as initial stack.
For performance reasons, stacks should be located in fast memory.

6.5. Stack check using *embOS*

embOS performs automatic stack check of all stacks, when a stack check or debug library is used.
Therefore, it Is required, that ***embOS*** knows the size and address of the system and user stack.
The startup file has to use and export the following symbols for the stack definition:

- **__systemstack** is the base address of the system stack.
- **__systemstack_top** is the first location behind the system stack. The system stack pointer has to be initialized with this value.
- **__userstack** is the base address of the user stack.
- **__usestack_top** is the first location behind the user stack. The user stack pointer has to be initialized with this value.

The startup code has to select the user stack as default stack before `main()` is called.
If `main()` is called with the system stack selected, embOS will end in the error handler `OS_Error()` with error code **`OS_ERR_WRONG_STACK`**. (0x67) .

7. Interrupts

7.1. 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 the requesting interrupt priority level, the interrupt is executed
- the CPU switches to system stack
- the CPU saves PC and flags on the stack
- the ILM register is loaded with the priority of the requesting interrupt
- 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 and switching to User stack
- For details, please refer to the FR CPUs users manual.

IMPORTANT:
Fujitsu's FR CPUs does not automatically disable interrupts, when an ISR is entered. Therefore nested interrupts are enabled per default.

7.2. Defining interrupt handlers in "C"

Routines defined with the keyword `__interrupt` automatically save & restore the registers they modify and return with RETI.
The interrupt vector (interrupt function name) has to be added to the interrupt vector table.
To enable hardware interrupts, the appropriate interrupt priority control register has to be initialized also.
For a detailed description on how to define an interrupt routine in "C", refer to the Fcc911s C-Compiler's user's guide.

Example

"Simple" interrupt-routine

```
__interrupt void IntHandlerTimerA1(void) {
  IntCnt++;
}
```

Interrupt-routine calling *embOS* functions

```
__interrupt void IntHandlerTimerA1(void) {
  OS_EnterInterrupt();  /* Inform embOS that interrupt function is running */
  IntCnt++;
  OS_PutMailCond(&MB_Data, &IntCnt);
  OS_LeaveInterrupt();
}
```

`OS_EnterInterrupt()` has to be the first function called in an interrupt handler using *embOS* functions, when nestable interrupts are not required. `OS_LeaveInterrupt()` has to be called at the end the interrupt handler then.

If interrupts should be nested, use `OS_EnterNestableInterrupt()` / `OS_LeaveNestableInterrupt()` instead.

7.3. Defining interrupt vectors

Most commonly, interrupt vector tables are defined in the assembler include file “intvecs.inc” which is then included in “startup.asm”.
Our sample interrupt vector table files delivered with ***embOS*** are derived from Fujitsus sample interrupt vector table files and contain all modifications required for ***embOS***.
Please refer to the comment in the file header of intvecs.inc.

Interrupt vectors could also be defined in “C” source file as described in Fcc911s compiler manual.
Unfortunately this requires all interrupt vectors defined in the same “C”-source file. As ***embOS*** needs a timer interrupt and optionally two UART interrupts, the interrupt vector table may be defined in Rtosinit_*.c. This requires import of application specific interrupt function names in Rtosinit_*.c. As this file may be changed by later updates, we do not recommend to modify Rtoinit_*.c. Therefore we choose the method of defining interrupts in the separate interrupt vector table file intvecs.inc which will be included in startup.asm. This method is most commonly used by all samples delivered with Softune workbench.

7.4. Zero latency, fast interrupts with FR CPUs

Instead of disabling interrupts when ***embOS*** does atomic operations, the interrupt level of the CPU is set to priority level 21. Therefore all interrupts with priorities of 20 to 16 can still be processed.
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 is 21.

7.5. 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 interrupt priorities between 21 and 30. These ***embOS*** interrupt handlers have to start with `OS_EnterInterrupt()` or `OS_EnterNestableInterrupt()` and end with `OS_LeaveInterrupt()` or `OS_LeaveNestableInterrupt()`.
- Any *Fast interrupt* (running at priority 16 to 20 must not call any ***embOS*** API function. Even `OS_EnterInterrupt()` and `OS_LeaveInterrupt()` must not be called.
- Interrupt handler running at low priorities, not calling any ***embOS*** API function, are allowed, but must not re-enable interrupts!

The priority limit between ***embOS*** interrupts and Fast interrupts is pre-defined as 20, thus allowing 5 different interrupt levels for zero latency interrupt handler. This setting can be changed at runtime by a call of the new ***embOS*** function `OS_SetFastIntPriorityLimit ()`.

7.6. Interrupt-stack

Since the FR CPUs have a separate stack pointer for interrupts, there is no need for explicit stack-switching in an interrupt routine. The routines `OS_EnterIntStack()` and `OS_LeaveIntStack()` are supplied for source compatibility to other processors only and have no functionality.

7.7. Special considerations for FR CPUs

When entering an external interrupt, FR CPUs do not automatically disable further interrupts, the interrupt priority for further interrupts is raised to the priority of the accepted interrupt + 1 instead.
Generic rules for ***embOS*** require that interrupts are disabled in an interrupt handler, unless `OS_EnterNestableInterrupt()` is called at the beginning of an interrupt handler.
Therefore interrupts are disabled when calling `OS_EnterInterrupt()`.

Rules for interrupt handlers which should not be interrupted by other interrupts:

- `OS_EnterInterrupt()` has to be the first function called. It disables further interrupts and informs ***embOS*** that interrupt code is running, thus inhibits task switches.
- Interrupts must not be reenabled in the interrupt handler.
- `OS_LeaveInterrupt()` has to be the last function called. It informs ***embOS*** that interrupt function ends and performs a task switch if required.

Rules for interrupt handlers which may be interrupted by other interrupts:

- `OS_EnterNestableInterrupt()` has to be the first function called. It enables further interrupts and informs ***embOS*** that interrupt code is running, thus inhibits task switches.
- `OS_LeaveNestableInterrupt()` has to be the last function called. It informs ***embOS*** that interrupt function ends and performs a task switch if required.

8. Stop / Sleep Mode

Usage of the Sleep mode 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 RtosInit.c.

The stop-mode works without a problem; however the real-time operating system is halted after entering stop mode. With external clock connected to an interrupt input which is able to release stop mode, the scheduler keeps working.
If internal timer is used for ***embOS***, the time variable is not updated. Therefore stop mode should not be used.

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 values in the table are for release build library.

Short description	ROM [byte]	RAM [byte]
Kernel	approx.1700	46
Event-management	< 200	---
Mailbox management	< 550	---
Single-byte mailbox management	< 300	---
Resource-semaphore management	< 250	---
Timer-management	< 250	---
Add. Task	---	32
Add. Semaphore	---	8
Add. Mailbox	---	20
Add. Timer	---	20
Power-management	---	---

10. Files shipped with *embOS* for FR

Directory	File	Explanation
root	`*.pdf`	Generic API and target specific documentation
root	`Release.html`	Release notes of ***embOS*** FR
root	`embOSView.exe`	Utility for runtime analysis, described in generic documentation
Start\BoardSupport*	`Start_*.prj`	Start projects for various FR CPUs
Start\BoardSupport*	`Start_*.dat`	Project settings for various FR CPUs
Start\BoardSupport*	`Start_*.sup`	Debugger support files for various FR CPUs
Start\BoardSupport*	`*.prc`	Debug macros macro for start projects
Start\Inc\	`RTOS.h`	To be included in any file using ***embOS*** functions
Start\Lib\	`*.lib`	***embOS*** libraries

11. Index