embOS

Real Time Operating System

CPU & Compiler specifics for

M16C/80, M32C and NC308 compiler

Document Rev. 4

SEGGER

A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Contents

1. About this document

This guide describes how to use ***embOS*** for M16C/80 and M32C Real Time Operating System for the Renesas M16C/80 and M32C series of microcontrollers using the Renesas NC308 compiler and Renesas HEW.

1.1. How to use this manual

This manual describes all CPU and compiler specifics for ***embOS*** using M16C/80 and M32C with NC308 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 Renesas HEW. If you have no experience using ***embOS***, you should follow this introduction, even if you do not plan to use PD308 debugger or Renesas HEW, 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 M16C/80 and M32C using NC308 compiler.

2. What is new?

- **Zero latency interrupts:**

Since version 3.88e of ***embOS*** for M32C, interrupt handling inside ***embOS*** was modified. Instead of disabling interrupts when ***embOS*** does atomic operations, the interrupt level of the CPU is set to 5. Therefore all interrupts with level 6 or above can still be processed.
The interrupt level for zero latency interrupts can be set at runtime.

2.1. Update / Upgrade information

When you update / upgrade from an ***embOS*** version prior 3.88e, you may have to change your interrupt handlers because of *Zero Latency interrupt* support. All interrupt handlers using ***embOS*** functions have to run on priorities from 1 to 5 (default), or up to the selectable level which was set by the new API function `OS_SetFastIntPriorityLimit()`.
For further information, please read chapter “Interrupts” in this manual.

3. Using *embOS* with Renesas HEW

3.1. Installation

embOS is shipped on a 3½" disk or CD-ROM or as a zip-file in electronic form.

In order to install it, proceed as follows:

- Copy the entire disk to your hard-drive into any folder of your choice. When copying, please keep all files in their sub directories!
- 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 HEW to develop your application, no further installation steps are required. You will find prepared sample start workspaces and projects for M16C80 and M32C CPUs, which you should use and modify to write your application. So follow the instructions in 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 HEW, you should:
Copy either all or only the library-file that you need to your work-directory. This has the advantage that when you switch to an updated version of ***embOS*** later in a project, you do not affect older projects that use ***embOS*** also.
embOS does in no way rely on the Renesas HEW workbench, it may be used with batch files or a make utilities without any problem.

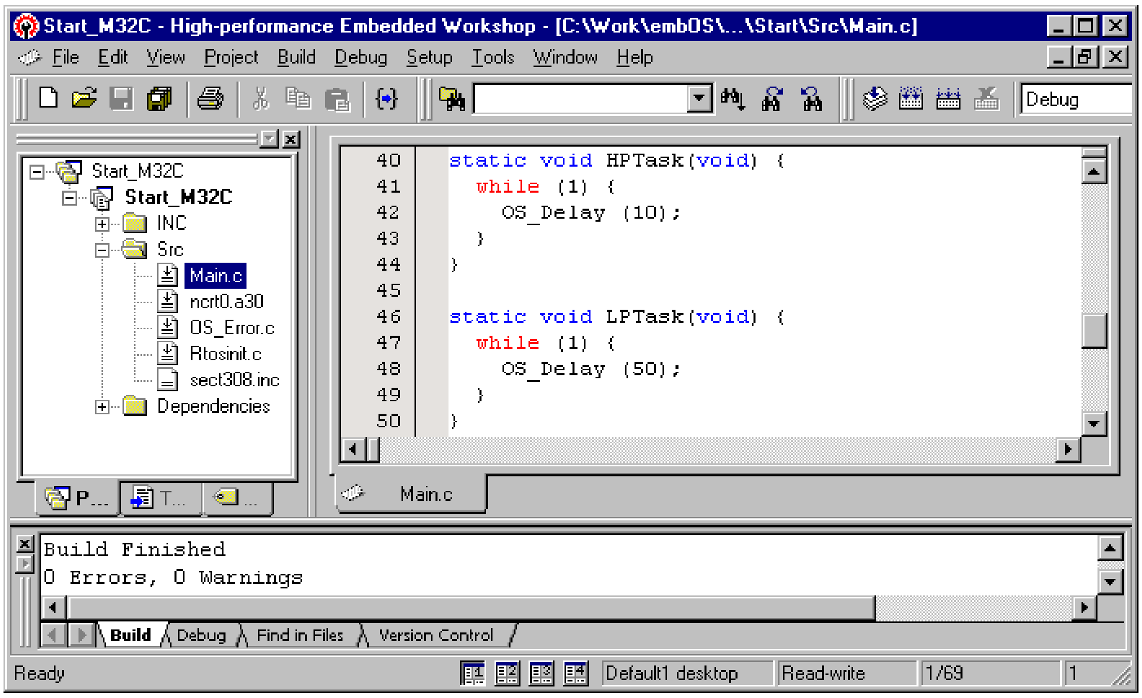
3.2. First steps

After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received ready to go sample start projects and it is a good idea to use them 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 one sample start project in the folder “start\” with Renesas HEW (e.g. by double clicking it) Our further examples show the Project workspace Start\Start_M32C.hws, which is for M32C with far memory model and debug library.
- Build the start project

Your screen should look like follows:


Start_M32C - High-performance Embedded Workshop - [C:\Work\embOS\...\Start\Src\Main.c]
File Edit View Project Build Debug Setup Tools Window Help
Debug
Start_M32C
Start_M32C
INC
Src
Main.c
ncrt0.a30
OS_Error.c
Rtosinit.c
sect308.inc
Dependencies
Main.c
Build Debug Find in Files Version Control
Ready
Default1 desktop
Read-write
1/69
1

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

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

```
Build Finished
0 Errors, 0 Warnings
```

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 OS
--------  END-OF-HEADER  ---------------------------------------------
*/

#include "RTOS.H"

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

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

static void LPTask(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(&TCBHP, "HP Task", HPTask, 100, StackHP);
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
  OS_Start();                      /* Start multitasking            */
  return 0;
}
```

4. Using Debugging tools

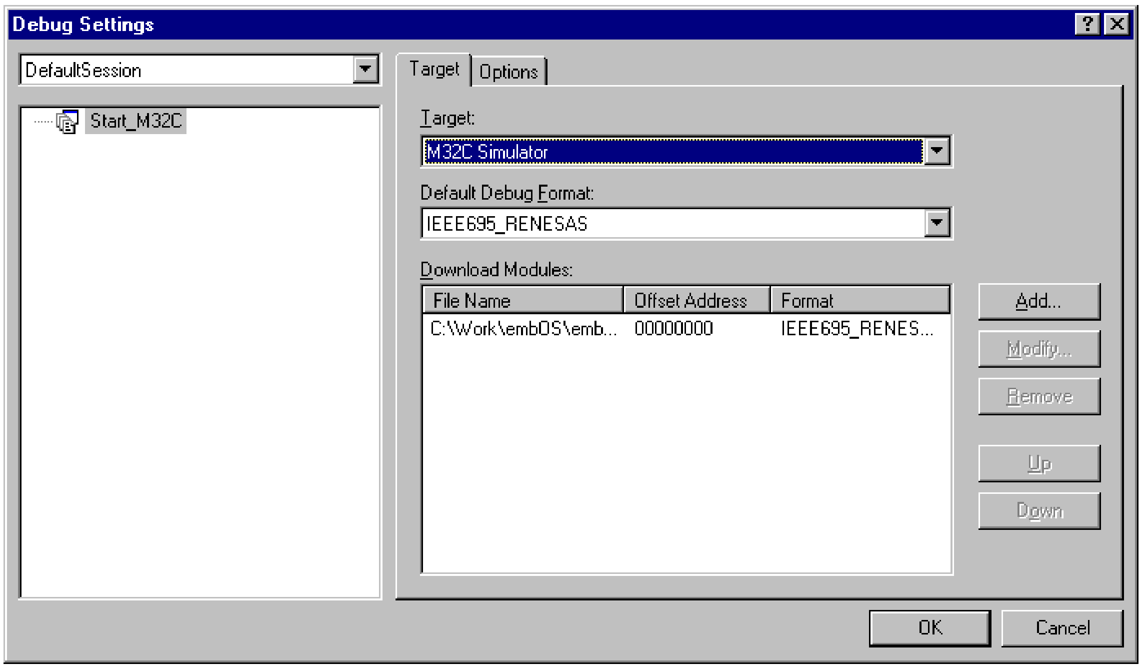
All debugging tools available for M32C and M16C80 CPUs may be used to debug an ***embOS*** application.

4.1. Debug the application using HEW M32C Simulator

The easiest way to debug the start project is using the M32C Simulator which is included in Renesas HEW.
The distribution of ***embOS*** for M32C comes with a simple timer simulation script which can be used to simulate the ***embOS*** system timer.
After building the start project, the M32C Simulator can be selected as debugger by the following procedure:

- From The main menu choose “Debug -> Debug Settings”.
- In the “Debug Settings” dialog, select Target: M32C Simulator.
- In the “Debug Settings” dialog, select Default Debug Format: IEEE695_RENESAS.
  In the “Debug Settings” dialog, add the generated output of the start project to the list “Download Modules”. The output file of the sample start project for M32C is “Start\Start_M32C\debug\Start_M32C.x30”

The dialog should look like follows:



Target: M32C Simulator

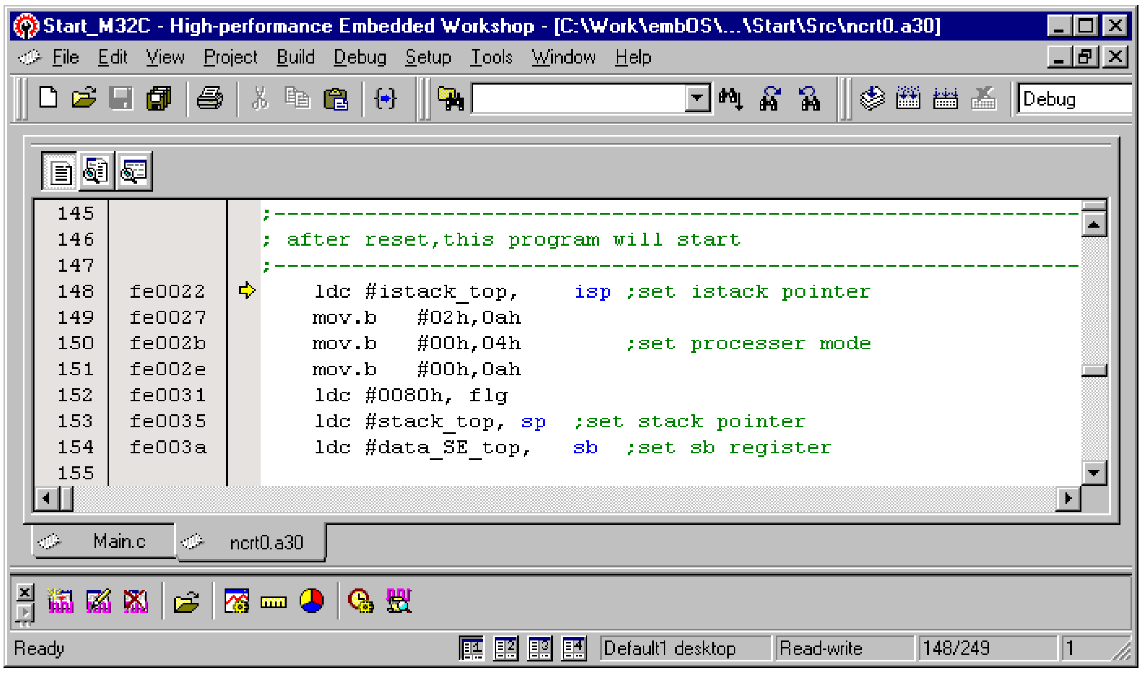
Default Debug Format: IEEE695_RENESAS

Download Modules:

File Name	Offset Address	Format
C:\Work\embOS\emb...	00000000	IEEE695_RENES...

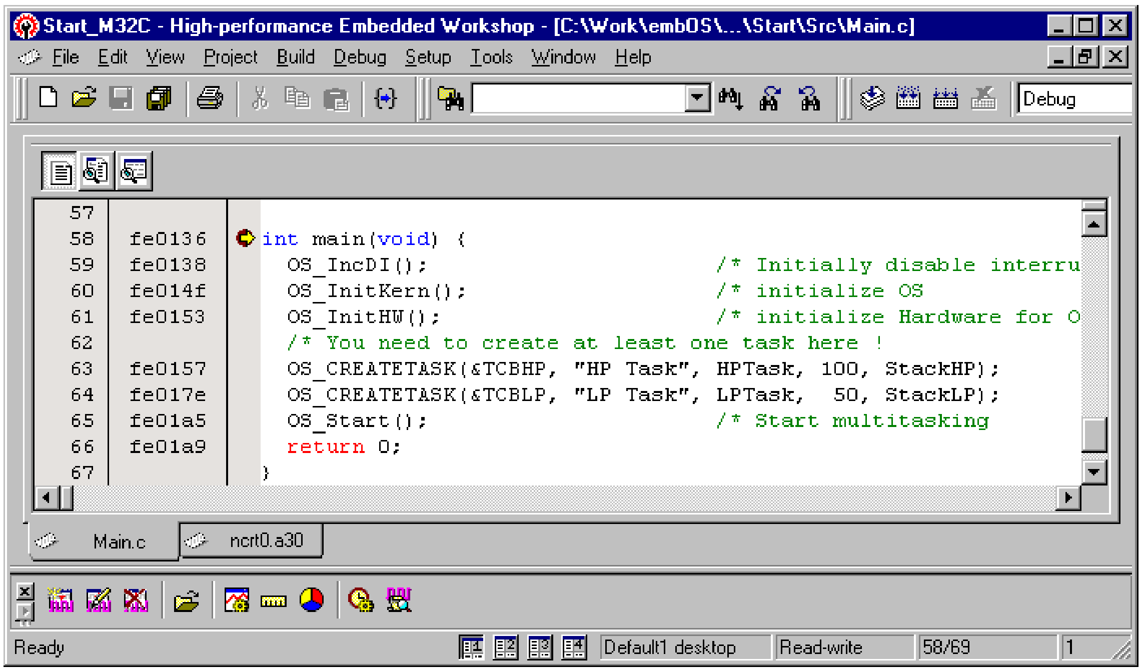
You will then have to setup the M32C Simulator, if not already done.
When you choose “Debug -> Connect” from the main menu, The “Init (M32C Simulator)” dialog appears.
You have to define a CPU in the “MCU” tab. To select a CPU, click the “Refer…” button.
For the Start_M32C sample, select the M32C8x.mcu file.
Depending on other options, the debugger then automatically loads the target file.
To be sure the right target is loaded, you may choose “Debug -> Download Modules” from the main menu and load the Start_M32C.x30 file.

The Debugger will load the file and show the startup code:


Start_M32C - High-performance Embedded Workshop - [C:\Work\embOS\...\Start\Src\ncrt0.a30]
File Edit View Project Build Debug Setup Tools Window Help
Debug
Main.c
ncrt0.a30
Ready
Default1 desktop
Read-write
148/249
1

```
;-----------------------------------------------------------------
; after reset,this program will start
;-----------------------------------------------------------------
fe0022     ldc #istack_top,    isp ;set istack pointer
fe0027     mov.b   #02h,0ah
fe002b     mov.b   #00h,04h        ;set processer mode
fe002e     mov.b   #00h,0ah
fe0031     ldc #0080h, flg
fe0035     ldc #stack_top, sp  ;set stack pointer
fe003a     ldc #data_SE_top,   sb  ;set sb register
```

You should open or select the main.c file and set a breakpoint at `main()`
When you the start the CPU by "Debug -> Go" or just press F5, the simulator stops at main. Alternatively, you may stop through the startup code to get there:

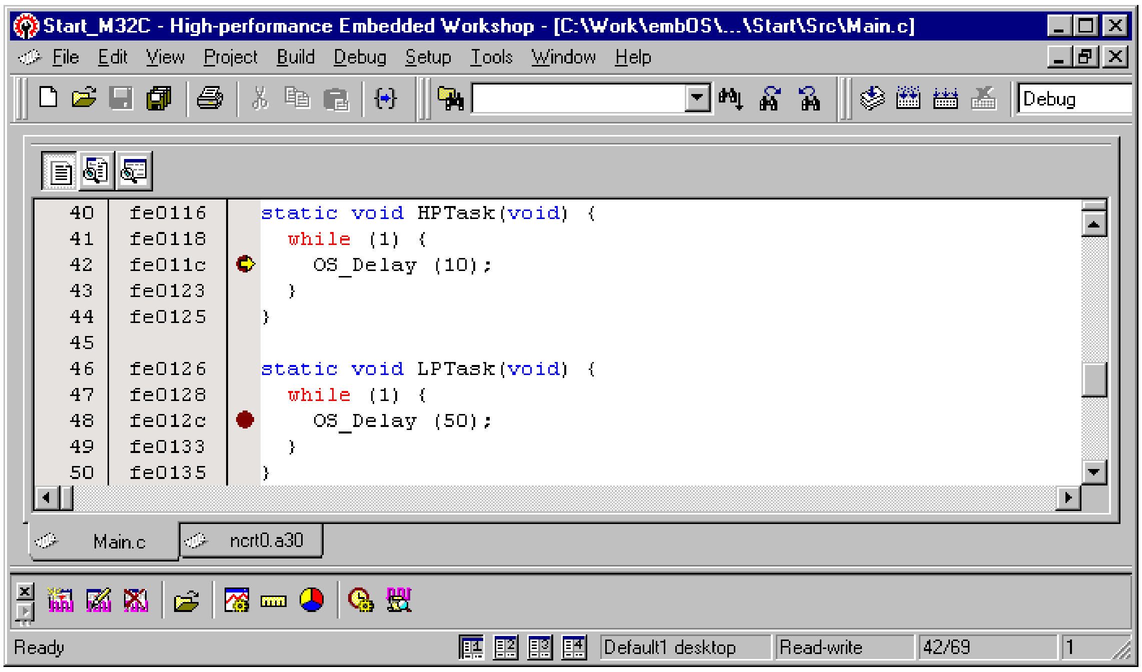

Start_M32C - High-performance Embedded Workshop - [C:\Work\embOS\...\Start\Src\Main.c]
File Edit View Project Build Debug Setup Tools Window Help
Debug
Main.c
ncrt0.a30
Ready
Default1 desktop
Read-write
58/69
1

```
fe0136  int main(void) {
fe0138    OS_IncDI();                         /* Initially disable interru
fe014f    OS_InitKern();                      /* initialize OS
fe0153    OS_InitHW();                        /* initialize Hardware for O
          /* You need to create at least one task here !
fe0157    OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
fe017e    OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
fe01a5    OS_Start();                         /* Start multitasking
fe01a9    return 0;
        }
```

You may now step through the sample application.

- `OS_IncDI()` initially disables interrupts and prevents re-enabling them in `OS_InitKern()`.
- `OS_InitKern()` is part of the ***embOS*** Library; you can therefore only step into it in disassembly mode. It initializes the relevant OS-Variables.
- `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. `OS_Start()` automatically enables interrupts.

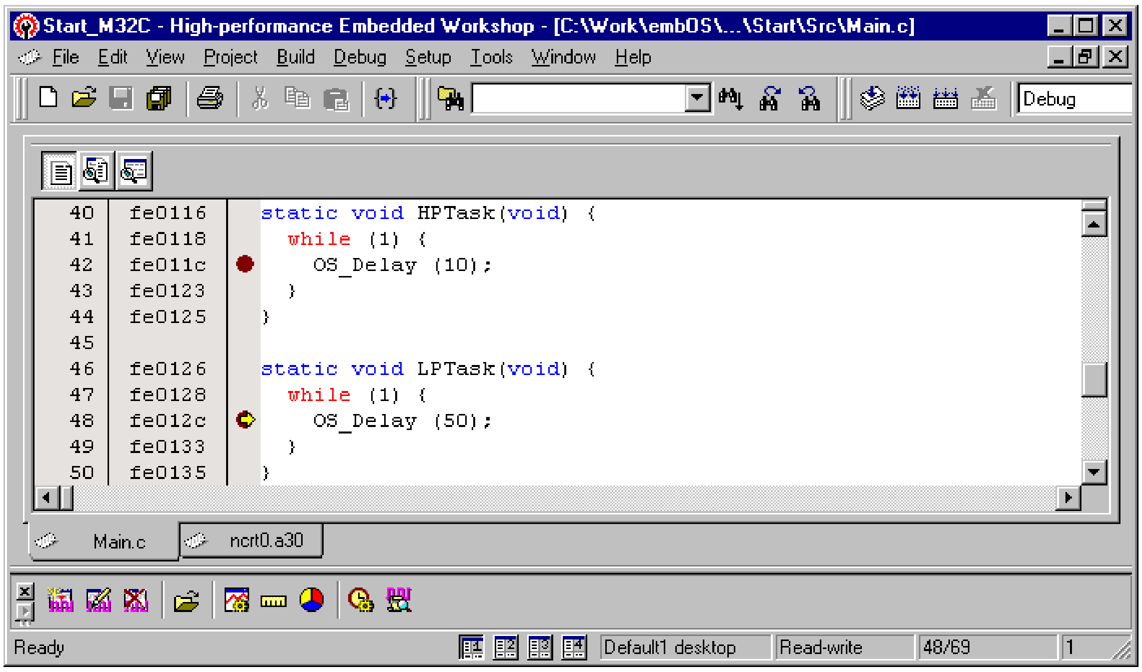
When you step into `OS_Start()`, the next line executed is already in the highest priority task created. (you may also use disassembly mode to get there of course, then stepping through the task switching process, but you must not step over `OS_Start()`). In our small start program, `HPTask()` is the highest priority task and is therefore active:



Start_M32C - High-performance Embedded Workshop - [C:\Work\embOS\...\Start\Src\Main.c]

```
fe0116   static void HPTask(void) {
fe0118     while (1) {
fe011c       OS_Delay (10);
fe0123     }
fe0125   }

fe0126   static void LPTask(void) {
fe0128     while (1) {
fe012c       OS_Delay (50);
fe0133     }
fe0135   }
```

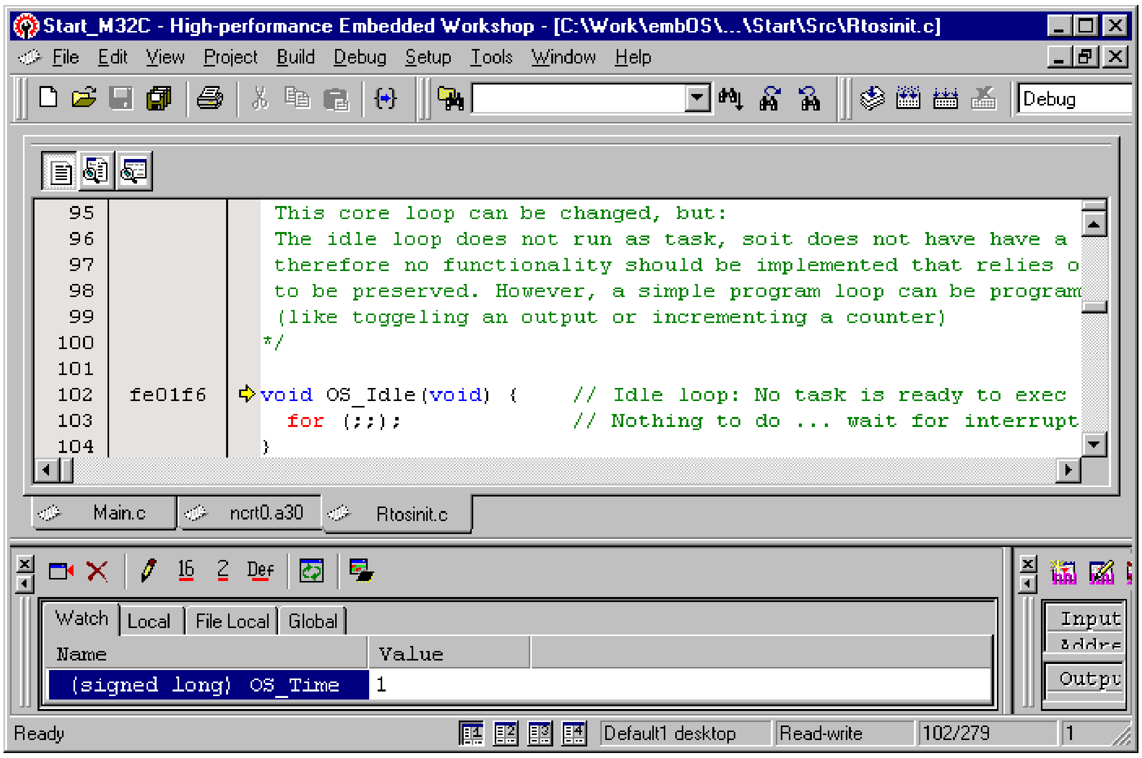
You should set a breakpoint in every task, as shown above. If you continue stepping, you will arrive in the task with the second highest priority:



Start_M32C - High-performance Embedded Workshop - [C:\Work\embOS\...\Start\Src\Main.c]

```
fe0116   static void HPTask(void) {
fe0118     while (1) {
fe011c       OS_Delay (10);
fe0123     }
fe0125   }

fe0126   static void LPTask(void) {
fe0128     while (1) {
fe012c       OS_Delay (50);
fe0133     }
fe0135   }
```

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).
When you step into the `OS_Delay()`, you will arrive there:

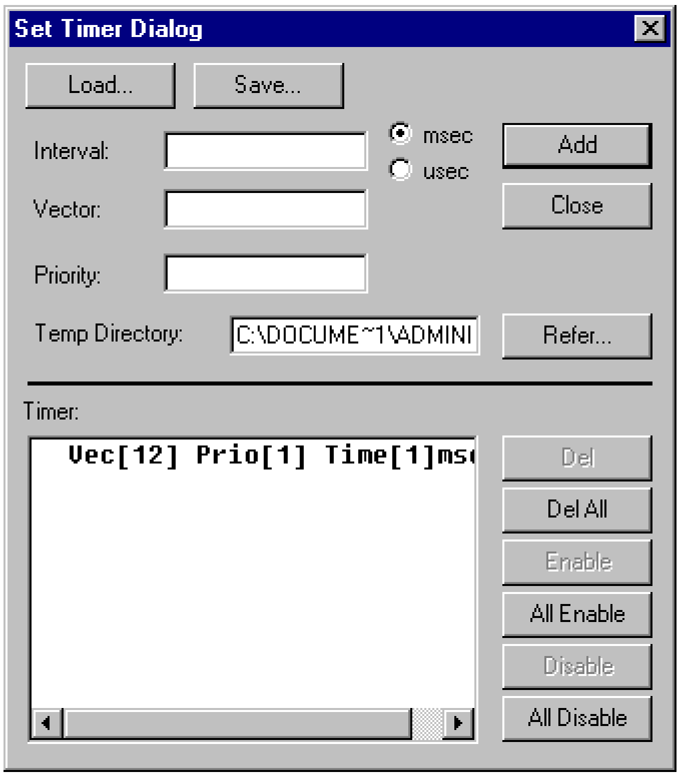

Start_M32C - High-performance Embedded Workshop - [C:\Work\embOS\...\Start\Src\Rtosinit.c]
File Edit View Project Build Debug Setup Tools Window Help
Debug
Main.c
ncrt0.a30
Rtosinit.c
Watch Local File Local Global
Ready
Default1 desktop
Read-write
102/279
1

```
 This core loop can be changed, but:
 The idle loop does not run as task, soit does not have have a
 therefore no functionality should be implemented that relies o
 to be preserved. However, a simple program loop can be program
 (like toggeling an output or incrementing a counter)
*/

void OS_Idle(void) {     // Idle loop: No task is ready to exec
  for (;;);              // Nothing to do ... wait for interrupt
}
```

Name	Value
(signed long) OS_Time	1

If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay. You may open the watch window to display the ***embOS*** time variable OS_Global.Time, which shows how much time has expired in the target system.
Now, it is time to start the embOS timer simulation script:

- From main menu choose View -> CPU -> I/O Timing Setting.
  The I/O Timing setting window opens.
- Select the Timer-symbol.
  The Set Timer Dialog opens
- Choose "Load..."
- Select the file Start\embOS_Timer.stm which is delivered with embOS.

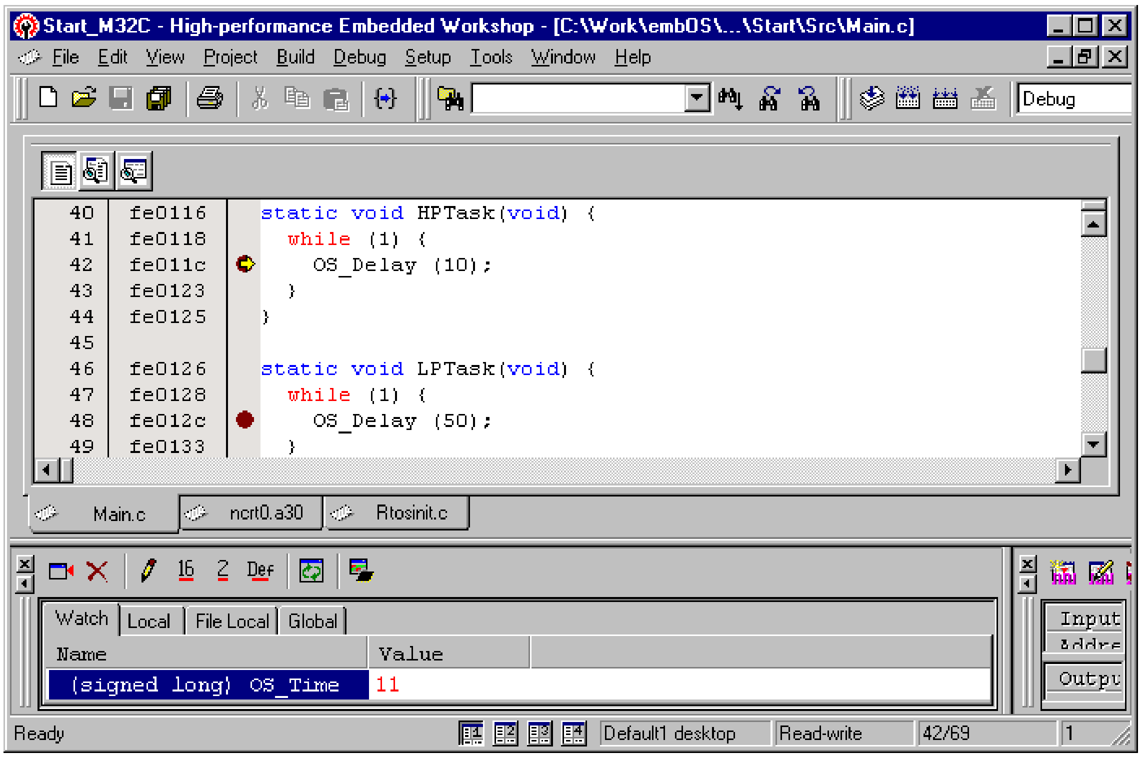


Interval:
Vector:
Priority:
Temp Directory: C:\DOCUME~1\ADMINI
Timer:
Vec[12] Prio[1] Time[1]ms

The timer script generates interrupt 12 which is used for the embOS timer

- Close the Set Timer dialog.
- Do not close the I/O Timing settings window, because the timer only runs as long as this window is open.

Now start the target CPU by “Debug -> Go” or press F5. The HP task will continue after the given delay of 10 ms:


Start_M32C - High-performance Embedded Workshop - [C:\Work\embOS\...\Start\Src\Main.c]
File Edit View Project Build Debug Setup Tools Window Help
Debug
Main.c
ncrt0.a30
Rtosinit.c
Watch Local File Local Global
Ready
Default1 desktop
Read-write
42/69
1

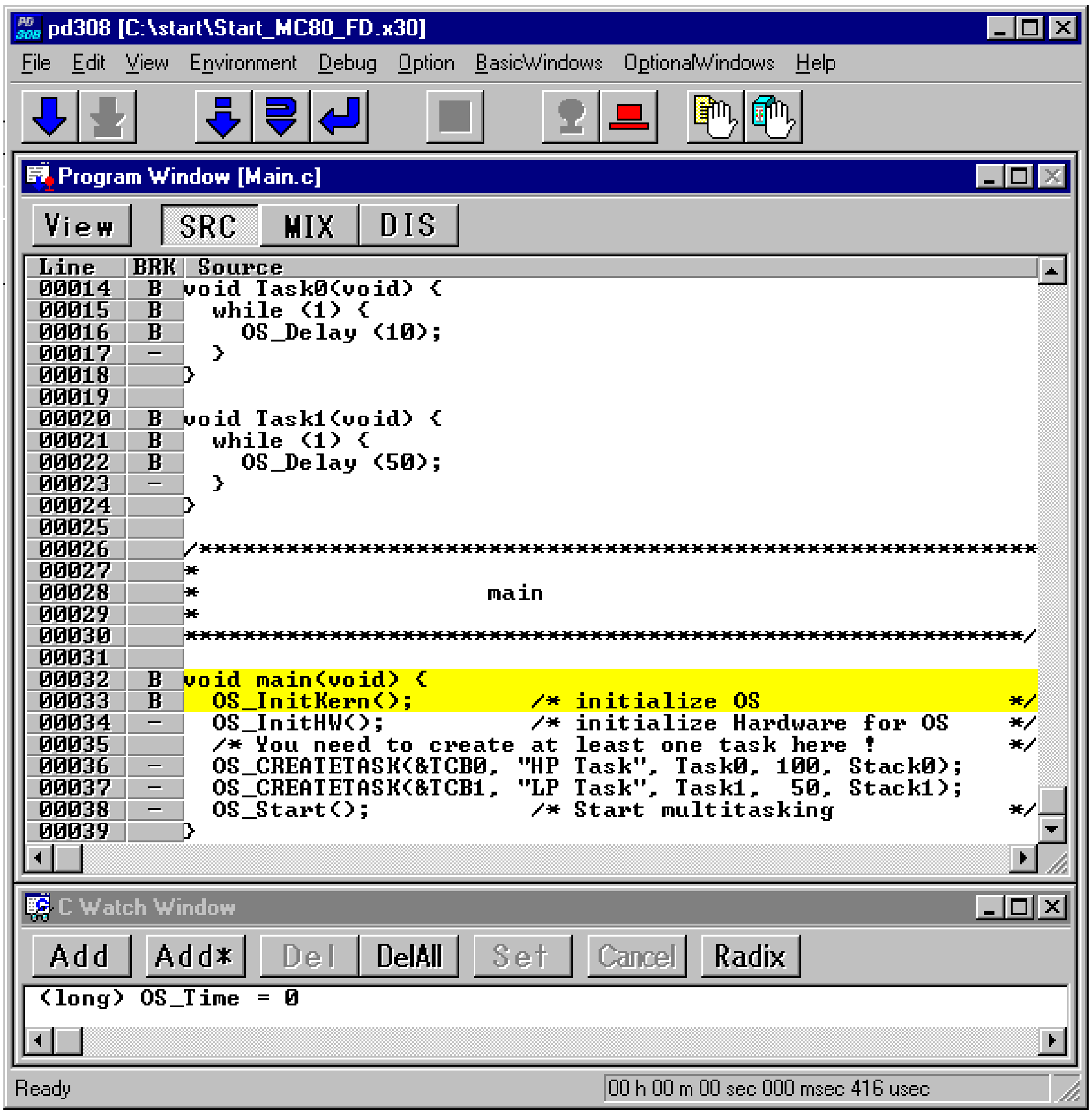
```
fe0116  static void HPTask(void) {
fe0118    while (1) {
fe011c      OS_Delay (10);
fe0123    }
fe0125  }

fe0126  static void LPTask(void) {
fe0128    while (1) {
fe012c      OS_Delay (50);
fe0133    }
```

Name	Value
(signed long) OS_Time	11

4.2. Debug the application using PD308

Start the debugger and load the X30 module file. Usually you will 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 and prevents re-enabling them in `OS_InitKern()`.
- `OS_InitKern()` is part of the ***embOS*** Library; you can therefore only step into it in disassembly mode. It initializes the relevant OS-Variables.
- `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. `OS_Start()` automatically enables interrupts.

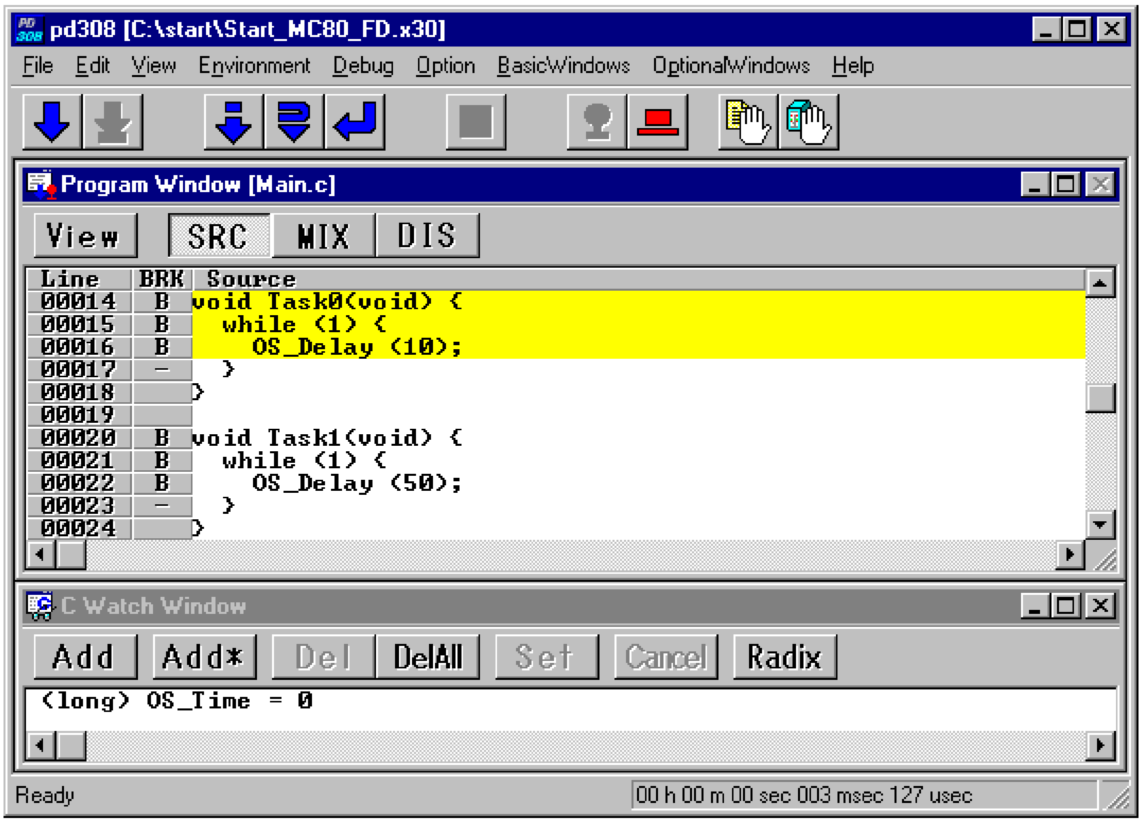

pd308 [C:\start\Start_MC80_FD.x30]
File Edit View Environment Debug Option BasicWindows OptionalWindows Help
Program Window [Main.c]
View SRC MIX DIS
C Watch Window
Add Add* Del DelAll Set Cancel Radix
Ready

```
Line   BRK Source
00014  B   void Task0(void) {
00015  B     while (1) {
00016  B       OS_Delay (10);
00017  -     }
00018      }
00019
00020  B   void Task1(void) {
00021  B     while (1) {
00022  B       OS_Delay (50);
00023  -     }
00024      }
00025
00026      /**********************************************************
00027      *
00028      *                    main
00029      *
00030      **********************************************************/
00031
00032  B   void main(void) {
00033  B     OS_InitKern();         /* initialize OS               */
00034  -     OS_InitHW();           /* initialize Hardware for OS  */
00035        /* You need to create at least one task here !      */
00036  -     OS_CREATETASK(&TCB0, "HP Task", Task0, 100, Stack0);
00037  -     OS_CREATETASK(&TCB1, "LP Task", Task1,  50, Stack1);
00038  -     OS_Start();            /* Start multitasking          */
00039      }
```

```
(long) OS_Time = 0
```

```
00 h 00 m 00 sec 000 msec 416 usec
```

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.

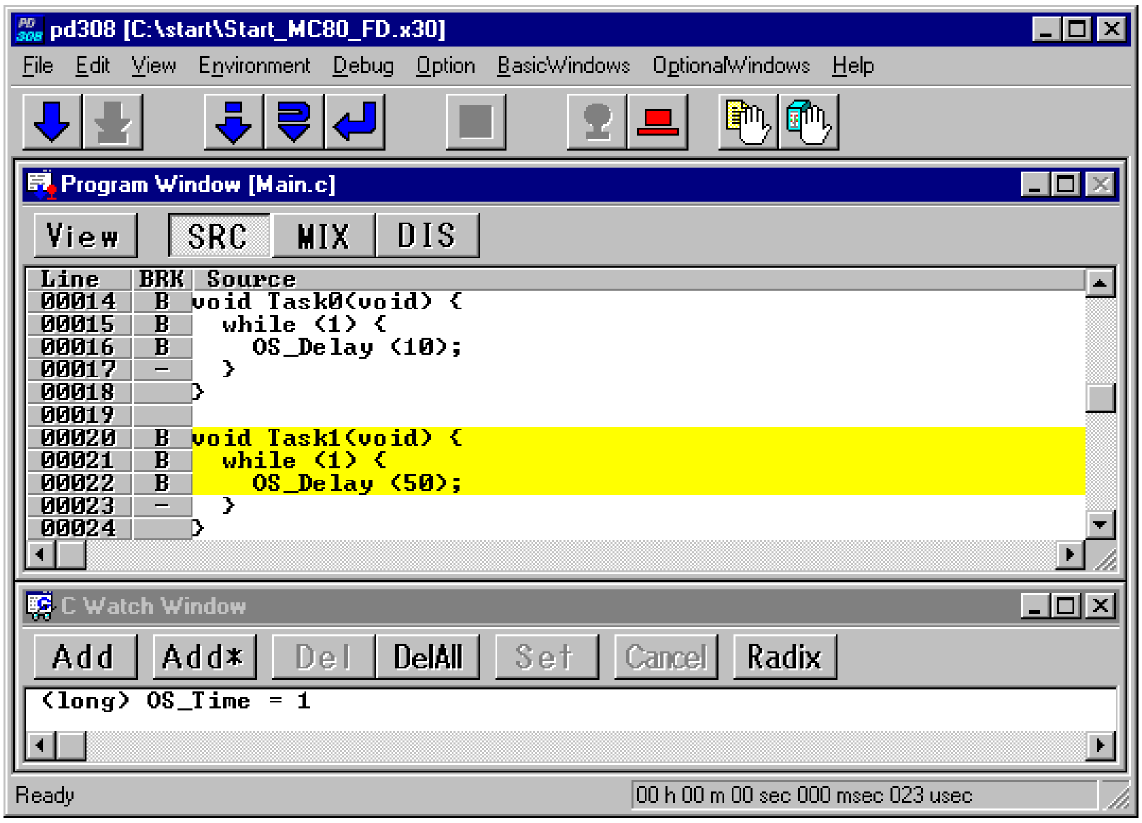

pd308 [C:\start\Start_MC80_FD.x30]
File Edit View Environment Debug Option BasicWindows OptionalWindows Help
Program Window [Main.c]
View SRC MIX DIS
C Watch Window
Add Add* Del DelAll Set Cancel Radix

```
BRK  Source
B    void Task0(void) {
B      while (1) {
B        OS_Delay (10);
-      }
     }

B    void Task1(void) {
B      while (1) {
B        OS_Delay (50);
-      }
     }
```

```
(long) OS_Time = 0
```

Ready 00 h 00 m 00 sec 003 msec 127 usec

You should set a breakpoint in every task, as shown above. If you continue stepping, you will arrive in the task with the second highest priority:

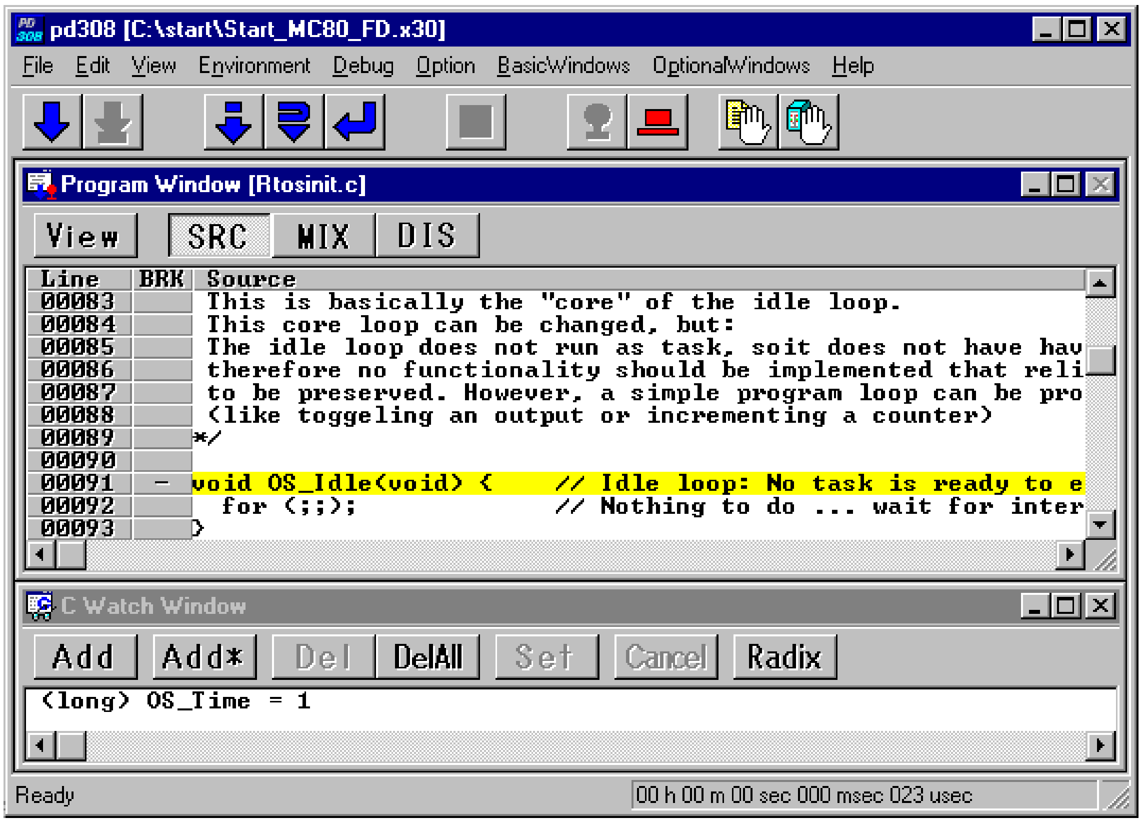

pd308 [C:\start\Start_MC80_FD.x30]
File Edit View Environment Debug Option BasicWindows OptionalWindows Help
Program Window [Main.c]
View SRC MIX DIS
C Watch Window
Add Add* Del DelAll Set Cancel Radix

```
BRK  Source
B    void Task0(void) {
B      while (1) {
B        OS_Delay (10);
-      }
     }

B    void Task1(void) {
B      while (1) {
B        OS_Delay (50);
-      }
     }
```

```
(long) OS_Time = 1
```

Ready 00 h 00 m 00 sec 000 msec 023 usec

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).

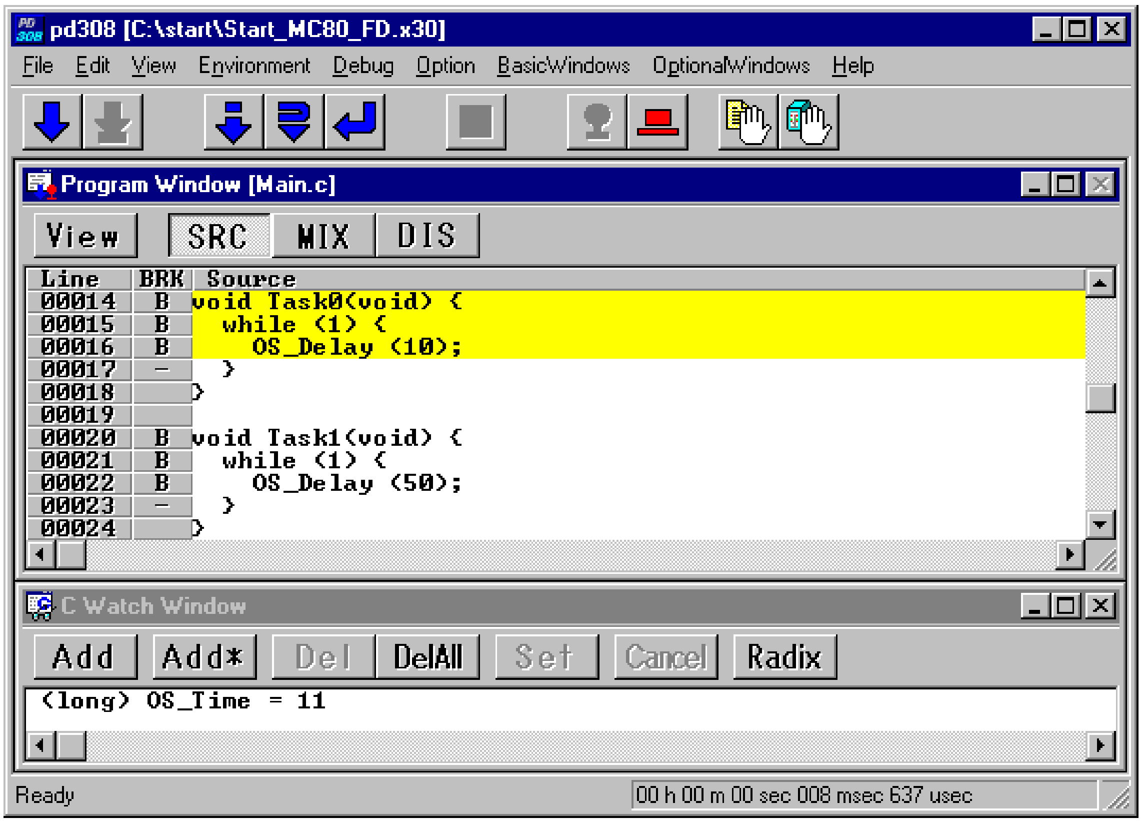

pd308 [C:\start\Start_MC80_FD.x30]
File Edit View Environment Debug Option BasicWindows OptionalWindows Help
Program Window [Rtosinit.c]
View SRC MIX DIS
C Watch Window
Add Add* Del DelAll Set Cancel Radix
Ready

```
Line   BRK  Source
00083        This is basically the "core" of the idle loop.
00084        This core loop can be changed, but:
00085        The idle loop does not run as task, soit does not have hav
00086        therefore no functionality should be implemented that reli
00087        to be preserved. However, a simple program loop can be pro
00088        (like toggeling an output or incrementing a counter)
00089       */
00090
00091   -   void OS_Idle(void) {     // Idle loop: No task is ready to e
00092         for (;;);              // Nothing to do ... wait for inter
00093       }
```

```
(long) OS_Time = 1
```

```
00 h 00 m 00 sec 000 msec 023 usec
```

If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay. You may open the watch window to display the *embOS* time variable OS_Time, which shows how much time has expired in the target system.


pd308 [C:\start\Start_MC80_FD.x30]
File Edit View Environment Debug Option BasicWindows OptionalWindows Help
Program Window [Main.c]
View SRC MIX DIS
C Watch Window
Add Add* Del DelAll Set Cancel Radix
Ready

```
Line   BRK  Source
00014   B   void Task0(void) {
00015   B     while (1) {
00016   B       OS_Delay (10);
00017   -     }
00018       }
00019
00020   B   void Task1(void) {
00021   B     while (1) {
00022   B       OS_Delay (50);
00023   -     }
00024       }
```

```
(long) OS_Time = 11
```

```
00 h 00 m 00 sec 008 msec 637 usec
```

Please note:
As the emulator does not stop the timer when it reaches a breakpoint, the timer continues counting and produces an interrupt as soon as the next step is executed. This results in extra counts of the time variable OS_Time.

4.3. Using ROM-Monitor KD308 or KD3083

The distribution of ***embOS*** M32C for NC308 compiler is prepared for usage of Renesas ROM Monitor KD308 or PD3083.
The ROM Monitor occupies the serial interface UART1 on some target CPUs and the appropriate interrupt vectors.
All necessary modifications are set up in the startup file NCRT0.a30 and interrupt vector definition file Sect308.inc, that are shipped with ***embOS***.
All modifications are documented in those source files. Please read the file header information there.
You should use the KD ROM-Monitor in free running mode.
Please note, that debugging of multitasking applications with KD ROM-Monitor is difficult.
When the ROM-Monitor stops at a breakpoint, the interrupts are not disabled. This may result in task switches, caused by interrupts. As a result, the ROM-Monitor may crash. You may set the interrupt priority of the target CPU to 6 before you start single-stepping to avoid such problems. Of course, the interrupt priority has to be changed to the previous value before you restart the application.

4.4. Using E8 or E8a debugging tool

RENESAS's E8a debugging tool can be used for M32C CPUs without problems.
The standard distribution of ***embOS*** for M16C80 / M32C and IAR compiler does not contain a configuration for the E8a, but an existing configuration can easily be changed to use E8a as debugging tool.
E8a occupies 256bytes of RAM and 2KBytes of ROM in the target CPUI. This memory has to be reserved and can not be used by the application.
The sect308.inc file can be modified to reserve the RAM and ROM required for E8a at the beginning of RAM and ROM. Therefore modify the .org address.

5. Build your own application

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

5.1. 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 Src\.
  It contains hardware dependent initialization code for the ***embOS*** timer and optional UART for embOSView.
- **OS_Error.c** from subfolder Src\.
  It contains the ***embOS*** runtime error handler `OS_Error()`which is used in stack check or debug builds.
- One ***embOS*** **library** from the Lib\ subfolder
- **ncrt0.a30** from subfolder Src\.
  This is the startup code which is modified to be used with ***embOS***.
- **sect308.inc** from subfolder Src\.
  This is the interrupt vector table file which is setup to be used with ***embOS***.

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.
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.

5.2. Select a start project

embOS comes with one start project for an M16C80 CPU and two start projects for M32C CPUs. The start projects were built and tested for standard CPUs. For various CPU variants there may be modifications required.

5.3. 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\.

5.4. Change memory model or library mode

For your application you may have to choose an other data- / memory-model. 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.
Therefore you have to replace the ***embOS*** library in your project or target:

- Build a new group for the library an add it to the selected target.
- Add the appropriate library from the Lib-subdirectory to your new group.
- Remove the previous library group from your target.

Finally check project options about target CPU data / memory model settings and compiler settings according library mode used. Refer to chapter 6 about the library naming conventions to select the correct library.

6. Start projects shipped with *embOS*

embOS for M32C and NC308 compiler is shipped with start projects for the Renesas High-performance Embedded Workshop HEW.
These start projects are located in the subfolder Start\.
There is one start project for an M16C80CPU, named Start_M16C80, and two start projects for M32c CPUs, named Start_M32C, which is for generic M32C CPUs, and Start_M32C87, which is built for an M32C/87 CPU.

Using start projects has the advantage, that all necessary compiler settings and defines are already done.
To develop your own application, you should use one of the start projects and add your sources to the project.

7. M16C/80 and M32C specifics

7.1. Memory models

embOS supports all the memory models that the Renesas NC308-Compiler supports.
For the M16C/80 and M32C, 2 memory models are available:

Model	Code	Data
Near	far (24 bits always)	near (16 bits)
Far	far (24 bits always)	far (24 bits)

7.2. Available libraries

All available libraries are placed in the subdirectory Start\lib.
The files to use are:

CPU	Memorymodel	Library type	Library
M16C/80	Near	Release	Rtos80NR.lib
M16C/80	Near	Stack-check	Rtos80NS.lib
M16C/80	Near	Stack-check + Profiling	Rtos80NSP.lib
M16C/80	Near	Debug	Rtos80ND.lib
M16C/80	Near	Debug + Profiling	Rtos80NDP.lib
M16C/80	Near	Debug + Profiling + Trace	Rtos80NDT.lib
M16C/80	Far	Release	Rtos80FR.lib
M16C/80	Far	Stack-check	Rtos80FS.lib
M16C/80	Far	Stack-check + Profiling	Rtos80FSP.lib
M16C/80	Far	Debug	Rtos80FD.lib
M16C/80	Far	Debug + Profiling	Rtos80FDP.lib
M16C/80	Far	Debug + Profiling + Trace	Rtos80FDT.lib
M32C	Near	Release	Rtos32NR.lib
M32C	Near	Stack-check	Rtos32NS.lib
M32C	Near	Stack-check + Profiling	Rtos32NSP.lib
M32C	Near	Debug	Rtos32ND.lib
M32C	Near	Debug + Profiling	Rtos32NDP.lib
M32C	Near	Debug + Profiling + Trace	Rtos32NDT.lib
M32C	Far	Release	Rtos32FR.lib
M32C	Far	Stack-check	Rtos32FS.lib
M32C	Far	Stack-check + Profiling	Rtos32FSP.lib
M32C	Far	Debug	Rtos32FD.lib
M32C	Far	Debug + Profiling	Rtos32FDP.lib
M32C	Far	Debug + Profiling + Trace	Rtos32FDT.lib

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

8. Compiler settings

8.1. CPU type

The compiler needs information about the CPU type used, to compile and assemble the correct code. Also the CPU type has to be specified to include the appropriate special function register definition header file, which is required to access the necessary peripherals.
This is done via parameter and define:

CPU	Parameter	Define	Explanation
M16C/80		-DCPUMC80	compiles code for M16C80 and includes CPUMC80.h
M32C	-M82	-DCPUM32C	compiles code for M32C and includes CPUM32C.h

The define could be omitted, when the #include directive in RtosInit.c would be changed.

8.2. Memory model

The memory model has to be chosen by compiler settings. The settings have to be confirm to those used for the libraries.
The following compiler settings are required for the different memory models

Model	Parameter	Define	Explanation
Near	-fNP	-OS_NDATA	All data are accessed in near memory area. Pointers are 16 bits wide
Far	-fFRAM	-OS_FDATA	All data may be placed in whole memory area. Pointers are 24 bits wide

8.3. Library type

The library type used for the application project has to be defined. This definition has to be confirm to the library, that is included to the project by linker option.

Library type	Library name	Define
Release	RTOSxxxR	-DOS_LIBMODE_R
Stack check	RTOSxxxS	-DOS_LIBMODE_S
Stack check + Profiling	RTOSxxxSP	-DOS_LIBMODE_SP
Debug	RTOSxxxD	-DOS_LIBMODE_D
Debug + Profiling	RTOSxxxDP	-DOS_LIBMODE_DP
Debug + Profiling + Trace	RTOSxxxDT	-DOS_LIBMODE_DT

9. Stacks

9.1. Task stack for M16C/80 and M32C

Every ***embOS*** task has to have its own stack. The task stacks may reside in any memory location that can be used as RAM. The stack-size required by a task is the sum of the stack-size of all routines that are called plus 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 M16C/80 and M32C, this minimum stack size for a task is about 50 bytes in the far memory model.

9.2. System stack for M16C/80 and M32C

The system stack size required by ***embOS*** is about 30 bytes (60 bytes in. profiling builds) The system stack is used by the application before the start of multitasking (the call of OS_Start()).
Because software-timers also use the system-stack, the actual stack requirements depend on the application.
The size of the system stack is defined in the assembler startup file NCRT0.a30 as STACKSIZE.

9.3. Interrupt stack for M16C/80 and M32C

The M16C/80 and M32C has been designed with multitasking in mind; it has 2 stack-pointers, the USP and the ISP. The U-Flag selects the active stack-pointer. During execution of a task or timer, the U-flag is set thereby selecting the user-stack-pointer. If an interrupt occurs, the M16C/80 and M32C clears the U-flag and switches to the interrupt-stack-pointer automatically this way. The ISP 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 is therefore not necessary for the M16C/80 and M32C.
The size of the interrupt stack is defined in the assembler startup file NCRT0.a30 as ISTACKSIZE. 192 bytes)

9.4. Stack specifics of the Renesas M16C/80 and M32C family

The Renesas M16C/80 and M32C family of microcontrollers can address up to 16MB of memory. Because the stack-pointer can address the entire memory area, stacks can be located anywhere in RAM. For performance reasons you should try to locate stacks in fast RAM.

10. Interrupts

10.1. 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 interrupt priority level, the interrupt is executed
- the CPU switches to the Interrupt stack
- the CPU saves PC and flags on the stack
- the IPL is loaded with the priority of the 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 IRET command, restoring PC, Flags and switching to User stack
- For details, please refer to the Renesas hardware manuals.

10.2. Defining interrupt handlers in "C"

Routines that should run as interrupt handlers have to be preceded by
`#pragma INTERRUPT RoutineName`
The following interrupt routine then automatically saves & restores the registers it modifies and returns with REIT.
For a detailed description on how to define an interrupt routine in "C", also refer to the Renesas C-Compiler's user's guide.

Example

Serial receive interrupt-routine for embOSView

```
#pragma INTERRUPT OS_ISR_rx
void OS_ISR_rx (void) {
  int Data;
  OS_EnterNestableInterrupt();  //  We will enable interrupts
  Data = U1RB;
  if (Data & 0x6000) { // Check if errors occurred
    U1C1 &= ~(1<<2);   // disable Rx
    U1C1 |=  (1<<2);   // enable Rx
  } else {
    OS_OnRx(Data);
  }
  OS_LeaveNestableInterrupt();
}
```

The interrupt vector has to be added to the interrupt vector table in Sect308.inc:

```
    .lword   _OS_ISR_rx      ; (Rx interrupt for embOSView, software int 18)
```

10.3. Interrupt-stack

Since the M16C/80 and M32C have a separate stack pointer for interrupts, there is no need for explicit stack-switching in an interrupt routine. The routines EnterIntStack and LeaveIntStack are supplied for source compatibility to other processors only and have no functionality.

10.4. Special considerations for the M16C/80 and M32C

None.

10.5. Zero Latency interrupts with M16C80 and M32C

Instead of disabling interrupts when ***embOS*** does atomic operations, the interrupt level of the CPU is set to 4. Therefore all interrupts with level 5 or above can still be processed.
These interrupts are named *Fast interrupts*. You must not execute any ***embOS*** function from within a *fast interrupt* function.

10.6. Zero latency interrupts with M16C80/M32C CPUs

Instead of disabling interrupts when ***embOS*** does atomic operations, the interrupt level of the CPU is set to a higher user definable level. Therefore all interrupts with higher levels can still be processed.
These interrupts are named ***Zero latency interrupts***.
The default level limit for zero latency interrupts is set to 5, meaning, any interrupt with level 6 or 7 is never disabled and can be accepted anytime.
You must not execute any *embOS* function from within a *Zero latency interrupt* function.

10.7. Interrupt priorities with *embOS* for M16C80/M32C CPUs

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

- Low priority interrupts with priorities from 1 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 ***Zero latency 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 Zero latency interrupts can be set at runtime by a call of the function `OS_SetFastIntPriorityLimit()`.

10.8. Interrupt latency

With ***embOS*** for M32C, the interrupt latencies are kept as small as possible, because high priority interrupts are never locked by the operating system.
Because the CPU automatically disables all interrupts when accepting an interrupt, the interrupt latency for interrupts with higher priority can not be zero. The interrupt handler has to re-enable interrupts by setting the I-Flag. Using ***embOS***, this is done by a call of the function `OS_EnterNestableInterrupt()` or `OS_EnterInterrupt()`.

Differences between OS_EnterInterrupt() and OS_EnterNestableInterrupt()

OS_EnterInterrupt() shall be used for an interrupt that may use ***embOS*** functions and runs on low priority (below the zero latency priority limit), but shall not be interrupted by other low priority interrupts.
OS_EnterInterrupt() sets the IPL of the CPU up to the zero latency priority limit and the re-enables interrupts.
OS_EnterNestableInterrupt() shall be used for an interrupt that may use ***embOS*** functions and runs on low priority (below the zero latency priority limit), but may be interrupted by other interrupts with higher priority. On entry, the IPL remains unchanged and interrupts are re-enabled.

10.9. OS_SetFastIntPriorityLimit(): Set the interrupt priority limit for Zero latency interrupts

The interrupt priority limit for Zero Latency interrupts is set to 5 by default. This means, all interrupts with higher priority from 6 to 7 will 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. All 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 7 which is the highest interrupt priority for interrupts.
To modify the default priority limit, `OS_SetFastIntPriorityLimit()` should be called before ***embOS*** was started.

11. 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 RtosInit.c.

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.

12. Technical data

12.1. 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 near memory model, release build.

Short description	ROM [byte]	RAM [byte]
Kernel	approx.1451	25
Event-management	< 200	---
Mailbox management	< 550	---
Single-byte mailbox management	< 300	---
Resource-semaphore management	< 250	---
Timer-management	< 250	---
Add. Task	---	19
Add. Resource Semaphore	---	4
Add. Mailbox	---	11
Add. Timer	---	11
Power-management	---	---

13. Files shipped with *embOS*

embOS for M32C and nc308 compiler is shipped with documentation in PDF format and release notes as html.
The start projects, source files, all libraries and additional files required for linker or emulator / simulator are located in the sub folder ‘Start’.
The distribution of ***embOS*** contains the following files:

Directory	File	Explanation
Start\Inc	`RTOS.h`	Include file for ***embOS***, to be included in every "C"-file using ***embOS*** -functions
Start\Inc	`CPU*.h`	Definition of special function registers. Needed for hardware initialization.
Start\Lib	`RTOS*.lib`	Libraries for all CPU / memory model / Library type combination
Start\Src	`Main.c`	Frame program to serve as a start
Start\Src	`RTOSINIT.c`	To be compiled & linked with your program, initializes the hardware, can be modified
Start\Src	`ncrt0.a30`	Startup code, modified for use with ***embOS***
Start\Src	`sect308.inc`	Sections and interrupt vector table, set up for ***embOS***
Start\Src	`OS_Error.c`	The ***embOS*** runtime error handler, used in stack check or debug builds.
Start\	`embOS_Timer.stm`	Simple timer simulation script for M32C Simulator to simulate ***embOS*** timer interrupt
Start*\	`*.*`	Sample start projects for HEW

Any add. files shipped serve as examples.

14. Index