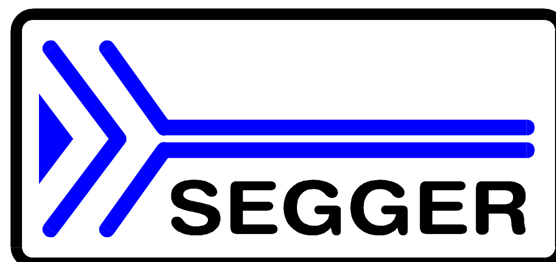
embOS

Real Time Operating System

CPU & Compiler specifics for
RENESAS M16C CPUs
and IAR compiler

Document Rev. 5



A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

© 2013 SEGGER Microcontroller GmbH & Co. KG

Contents

Contents ... 3
1. About this document ... 4
  1.1. How to use this manual ... 4
2. What's new? ... 4
  2.1. Update / Upgrade information ... 4
3. Using *embOS* with IAR's Embedded Workbench ... 5
  3.1. Installation ... 5
  3.2. First steps ... 6
  3.3. The sample application Main.c ... 6
4. Using debugging tools to debug the application ... 8
  4.1. Using IAR's C-Spy simulator ... 8
  4.2. Using KD30 ROM Monitor ... 11
  4.3. Interrupt vector definition file KD30Vect.asm ... 11
  4.4. Using PD30 / PC4701 in circuit emulator ... 12
  4.5. Using PD30Sim ... 16
  4.6. Common debugging hints ... 16
5. Build your own application ... 17
  5.1. Required files for an *embOS* application ... 17
  5.2. Select a start project ... 17
  5.3. Add your own code ... 17
  5.4. Change memory model or library mode ... 17
6. IAR compiler specifics ... 19
  6.1. Data / Memory models, compiler options ... 19
  6.2. Available libraries ... 19
  6.3. Distributed project files ... 20
  6.4. Distributed target configurations ... 20
7. M16C6N and M16C62P CPU specifics ... 21
  7.1. Clock settings and corrections for *embOS* timer interrupt ... 21
  7.2. Clock settings and corrections for UART used for embOSView ... 21
  7.3. PLL settings ... 22
  7.4. Conclusion about clock settings ... 22
8. Stacks ... 23
  8.1. Task stack for M16C ... 23
  8.2. System stack for M16C ... 23
  8.3. Interrupt stack for M16C ... 23
  8.4. Reducing the stack size ... 23
9. Interrupts ... 24
  9.1. What happens when an interrupt occurs? ... 24
  9.2. Defining interrupt handlers in "C" ... 24
  9.3. Interrupt vector table ... 24
  9.4. Interrupt-stack ... 24
  9.5. Fast interrupts with M16C ... 25
  9.6. Interrupt priorities ... 25
10. STOP / WAIT Mode ... 26
11. Technical data ... 27
  11.1. Memory requirements ... 27
12. Files shipped with *embOS* M16C for IAR compiler ... 27
13. Index ... 28

© 2013 SEGGER Microcontroller GmbH & Co. KG

1. About this document

This guide describes how to use ***embOS*** for M16C Real Time Operating System for the RENESAS M16C series of microcontroller using IAR compiler and IARs Embedded Workbench.

1.1. How to use this manual

This manual describes all CPU and compiler specifics for ***embOS*** using M16C CPUs with IAR 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 IAR C compiler and IAR's Embedded Workbench.. If you have no experience using ***embOS***, you should follow this introduction, even if you do not plan to use IAR's Embedded Workbench, 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 CPUs and IAR compiler.

2. What's new?

- **Additional libraries delivered with *embOS***

Since version 3.20, libraries for byte aligned objects and near constants in near memory model are delivered with ***embOS*** for M16C. Near constants are required for R8C CPU which is supported by new IAR workbench and compiler since version 2.12.

- **Fast interrupts:**

Since version 3.10p of ***embOS*** for M16C, interrupt handling inside ***embOS*** was modified. 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.

2.1. Update / Upgrade information

When you update / upgrade from an ***embOS*** version prior 3.10p, 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 1 to 4.
Please read chapter "Interrupts" in this manual.

© 2013 SEGGER Microcontroller GmbH & Co. KG

3. Using *embOS* with IAR's Embedded 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 IAR's Embedded Workbench to develop your application, no further installation steps are required. You will find a prepared sample start project for M16C 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 IAR's Embedded Workbench for your application development in order to become familiar with *embOS.*

embOS does in no way rely on IAR's Embedded Workbench, it may be used without the workbench using batch files or a make utility without any problem.

© 2013 SEGGER Microcontroller GmbH & Co. KG

3.2. First steps

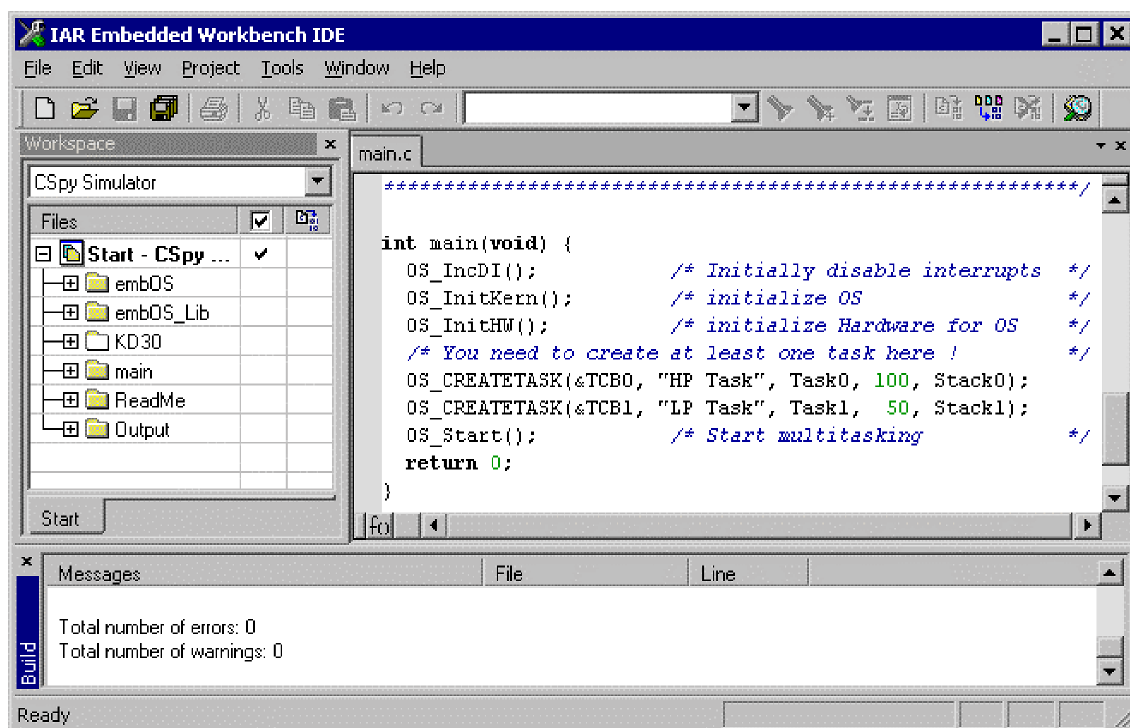
After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received a ready to go sample start workspace and project for M16C CPUs and it is a good idea to use this as a starting point of all your applications.

Your ***embOS*** distribution contains one folder 'Start' which contains the sample start workspace and project and every additional files used to build your application.

To get your application running, you should proceed as follows:

- Create a work directory for your application, for example c:\work
- Copy all files and subdirectories from the ***embOS*** distribution disk 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' in your work directory.
- Open the start workspace 'Start.eww'. (e.g. by double clicking it)
- Build the start project

After building the start project your screen should look like follows:



Initially a target for Near memory model for IAR's simulator / debugger CSpy should be selected.
If you do not have CSpy installed, you may select an other target which is useable for your simulator / debugger.

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.

© 2013 SEGGER Microcontroller GmbH & Co. KG

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;
}
```

© 2013 SEGGER Microcontroller GmbH & Co. KG

4. Using debugging tools to debug the application

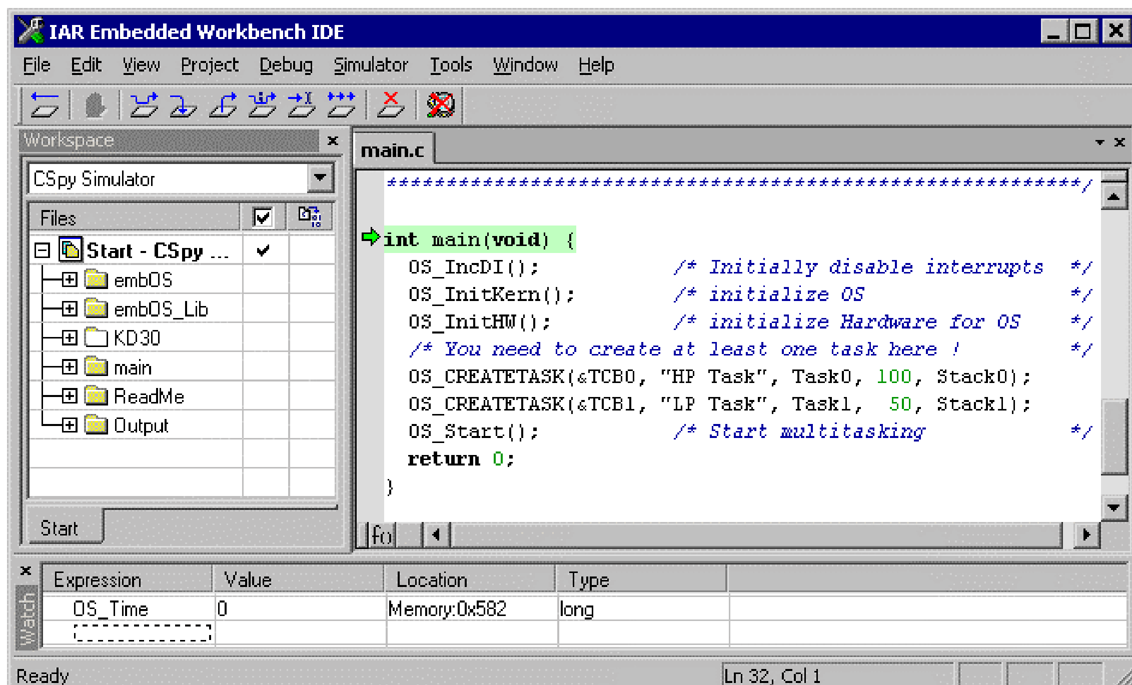
The ***embOS*** start project contains targets which are already setup for the following debugging tools:

- IAR's debugger / simulator CSpy. This target is named "CSpy_Simulator".
- RENESAS's ROM Monitor KD30. This target is named "Target_KD30" and may also be used with IAR CSpy in serial ROM monitor mode
- RENESAS's in circuit emulator PD30. This target is named "Target_PD30"

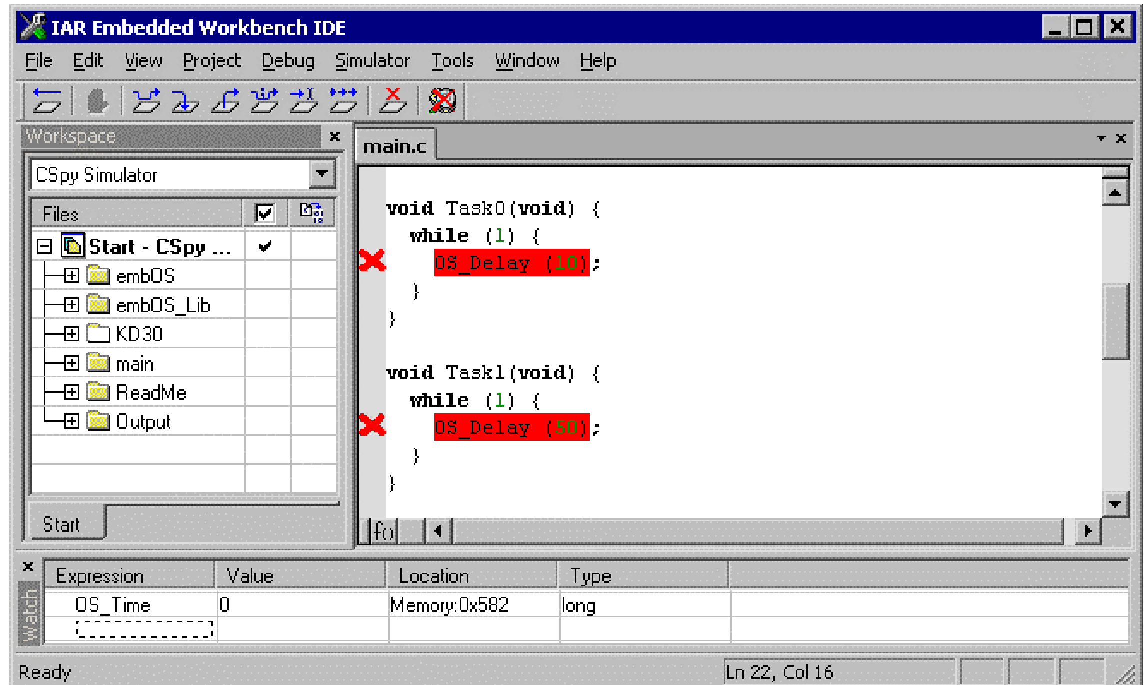
These targets are prepared to produce the appropriate output files required by your debugger.
The following chapters describe a sample session based on our sample application main.

4.1. Using IAR's C-Spy simulator

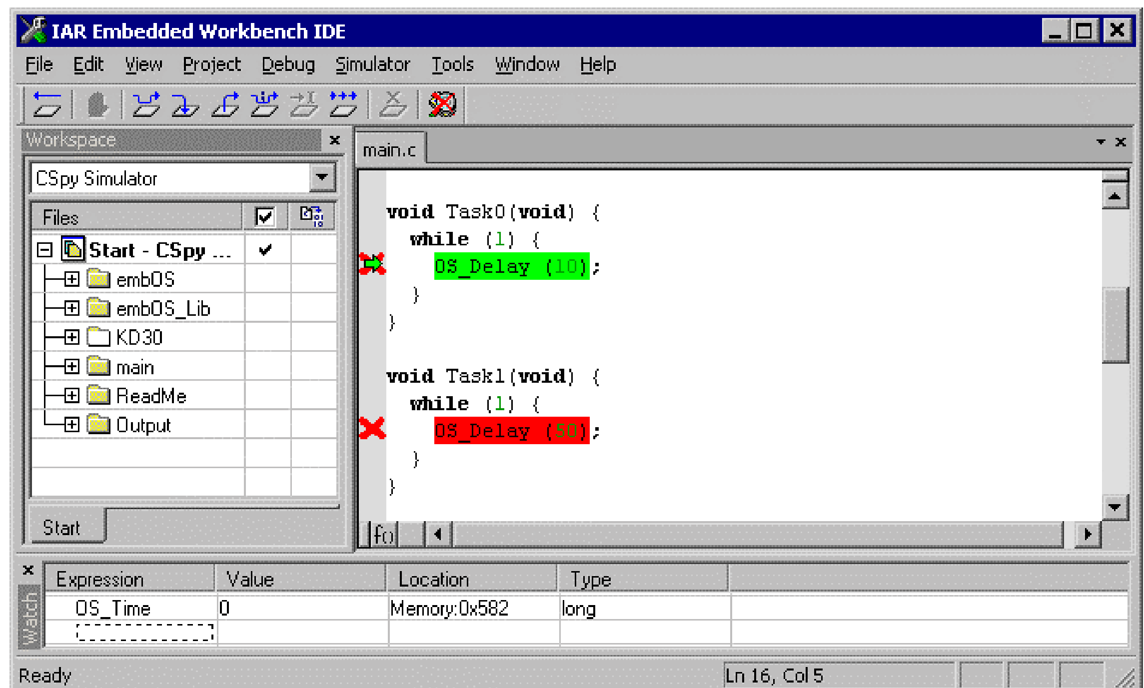
When starting C-Spy simulator after building the C-Spy target, you will usually see the main function, or you may look at the startup code and have to set a breakpoint at main. Now you can step through the program.
`OS_IncDI()`initially disables interrupts and prevents OS_InitKern() from re-enabling them.
`OS_InitKern()`initializes ***embOS*** -Variables. As this function is part of the ***embOS*** library, you may step into it in disassembly mode only.
`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_COM_Init()` in `OS_InitHW()` is optional. It is required if embOSView shall be used. As simulators usually can not simulate UART operations, OS_UART should be defined as (-1) to disable UART initialization and communication.
`OS_Start()` should be the last line in main, since it starts multitasking and does not return.



Before stepping over `OS_Start()`, you should set two breakpoints in our tasks in main.c as shown below:

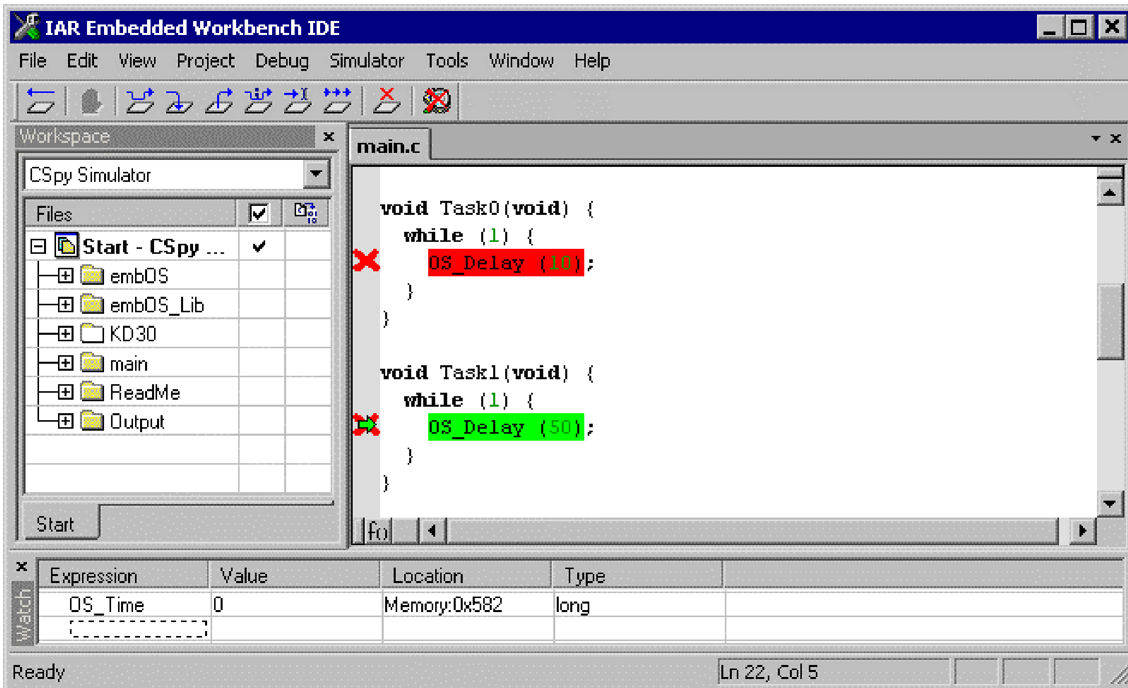


When you step over `OS_Start()` the next source line executed is Task0 which is the task with the highest priority in our start project and is therefore activated.

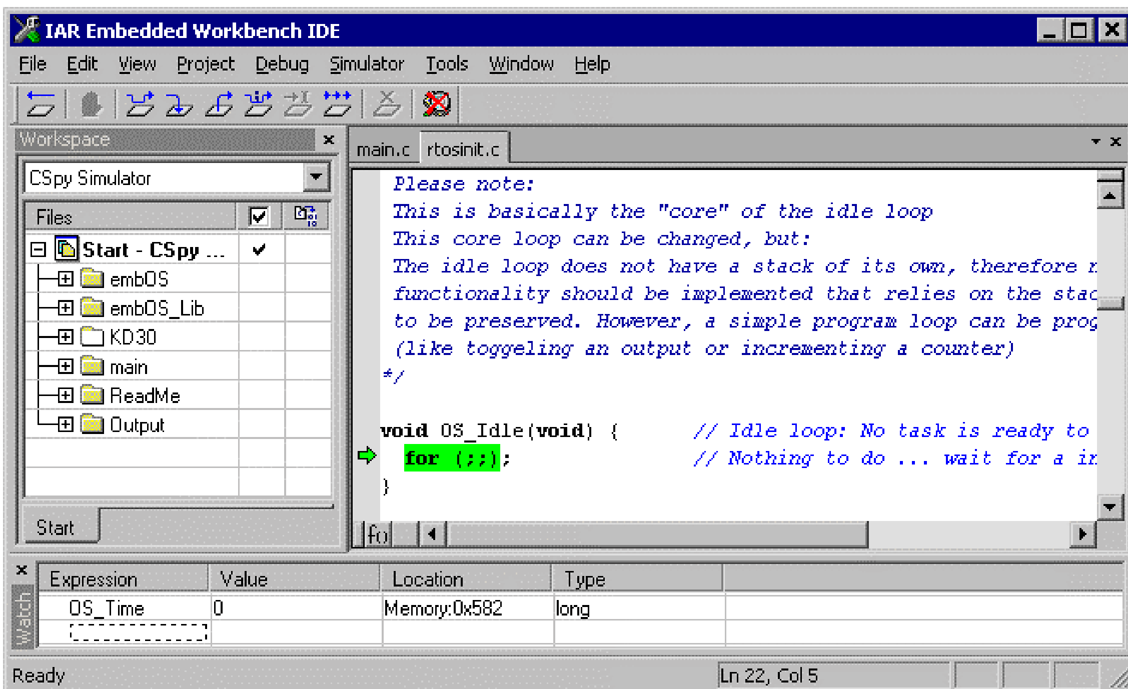


If you continue stepping, you will arrive in the task with the lower priority:

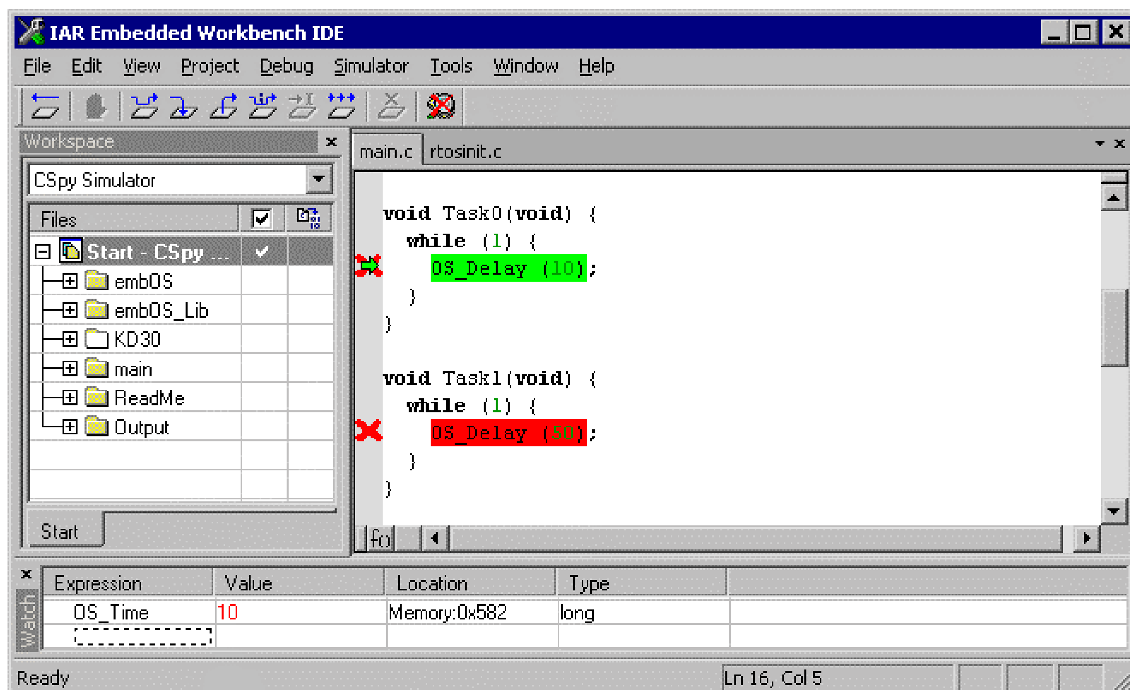
© 2013 SEGGER Microcontroller GmbH & Co. KG



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 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 both of our tasks, you will see that they continue execution after the given delay.
Coming from `OS_Idle()`, you should execute the ‘Go’ command:



As can be seen by the value of ***embOS*** timer variable `OS_Time`, shown in the watch window, Task0 continues operation after expiration of the 10 ms delay.

4.2. Using KD30 ROM Monitor

The distribution of ***embOS*** for M16C is prepared for usage of KD30 ROM monitor. KD30 ROM monitor needs UART interrupt vectors which point to KD30 internal interrupt functions.
These vectors are defined in ‘`KD30Vect.asm`’ which is included in the target “Target_KD30” which is setup for KD30. Please check whether these vectors fit to your version of ROM monitor.
As KD30 usually communicates via UART1 of the target CPU, this UART can not be selected as communication port for embOSView or for your application.
Note also, that the variable interrupt vector table has to be located below the highest target CPUs flash sector, which is used by KD30 ROM Monitor.
The linker file used by ***embOS*** KD30 target is set up accordingly.

Problem with KD30 ROM monitor running ***embOS*** application:

When ROM monitor stopped at a breakpoint, it may happen, that any interrupt activates a task switch while stepping through the program, as interrupts are enabled during stepping. This task switch can not be handled by KD30 and KD30 crashes.
To overcome this problem, you should open the register window and set interrupt priority (IPL) to 6 immediately after the breakpoint was reached. This enables stepping without any task switches, as all ***embOS*** interrupts normally run with lower priorities.
How to step through the sample application can be seen chapter 3.4 “Using PD30 / PC4701 in circuit emulator. PD30 is similar to KD30.

4.3. Interrupt vector definition file KD30Vect.asm

This file defines two interrupt vectors for UART1 used by KD30 ROM Monitor. When not using KD30, an interrupt vector definition file is not required, as ***embOS*** interrupts are defined in ‘C’-source code.

© 2013 SEGGER Microcontroller GmbH & Co. KG

When using KD30, please check whether the vector itself fits to your KD30 monitor version:

- 0FCB6Bh for old version of KD30
- 0FF900h for newer version of KD30 (above 3.0).

Both vectors for Rx- and Tx- interrupt point to the same address.

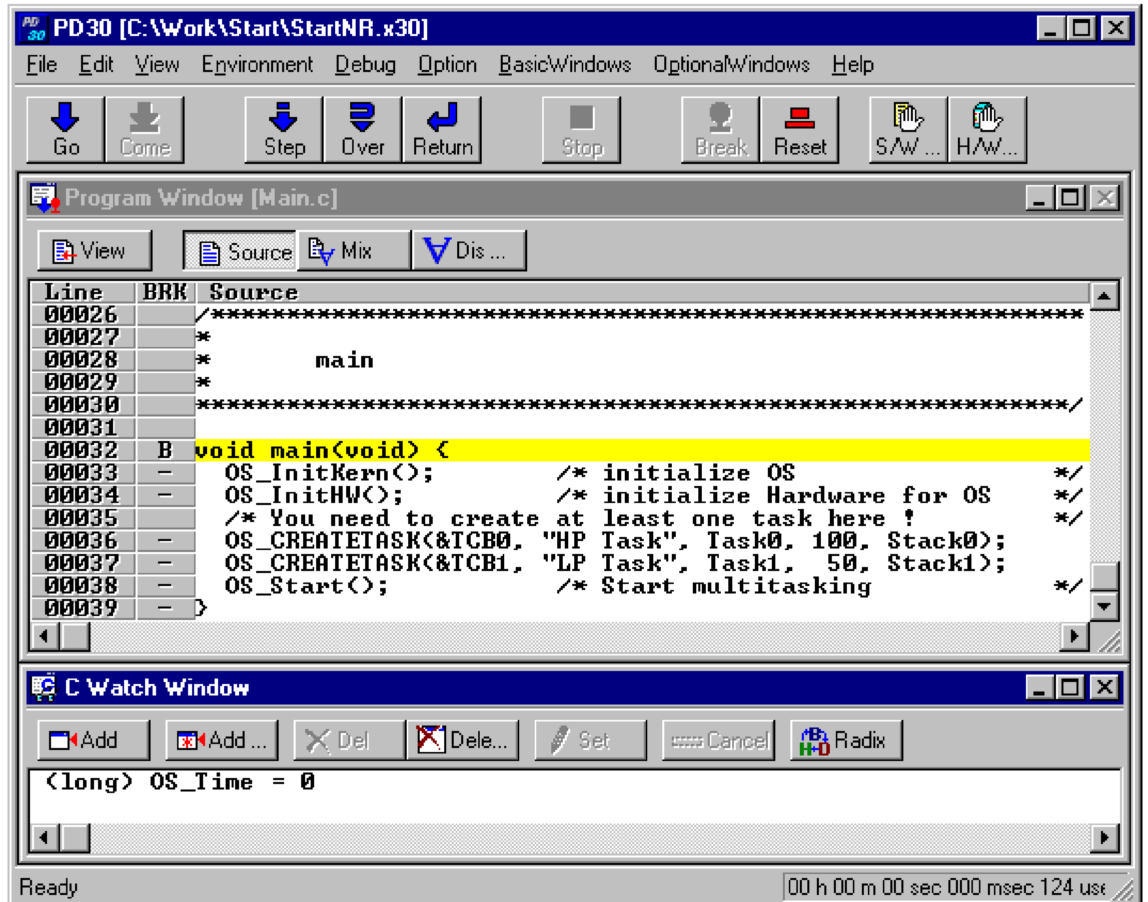
Important:

Please ensure, that this file is linked to your application, when needed for KD30 ROM-Monitor.
Check the project options for assembler AM16C, code generation:
'Make a LIBRARY module' option has to be unchecked. Otherwise the linker would optimize those vectors away, as they are not referenced by your application.

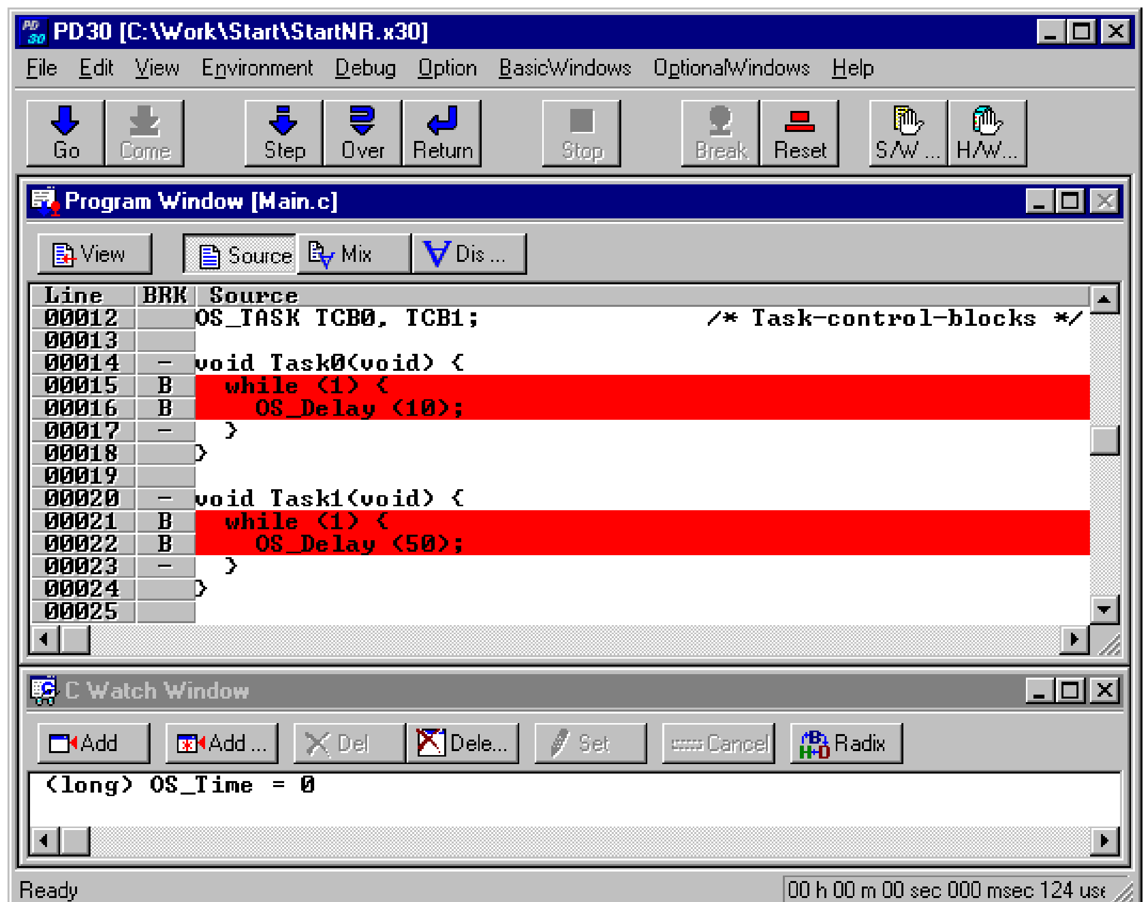
4.4. Using PD30 / PC4701 in circuit emulator

The standard distribution of ***embOS*** for M16C and IAR compiler contains a target for RENESAS's PD30 / PC4701 in circuit emulator.
This target is named "Target_PD30" and it produces an 'X30' output file with debug information which may be loaded into PD30 eg. RENESAS's PC4701 in circuit emulator to debug the application.
When starting the debugger and load the application, you will usually see the main function (very similar to the screenshot below). or you may look at the startup code and have to set a breakpoint at main. 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 ***embOS*** -Variables and enables interrupts. If you do want to enable interrupts from start, you are free to change your code by incrementing the interrupt-disable counter using `OS_IncDI()` before calling `OS_InitKern()`.
`OS_InitHW()` is part of RTOSINIT.c and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the timer-tick-interrupt for ***embOS.*** Step through it to see what is done.
`OS_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.
`OS_Start()` should be the last line in main, since it starts multitasking and does not return.

© 2013 SEGGER Microcontroller GmbH & Co. KG

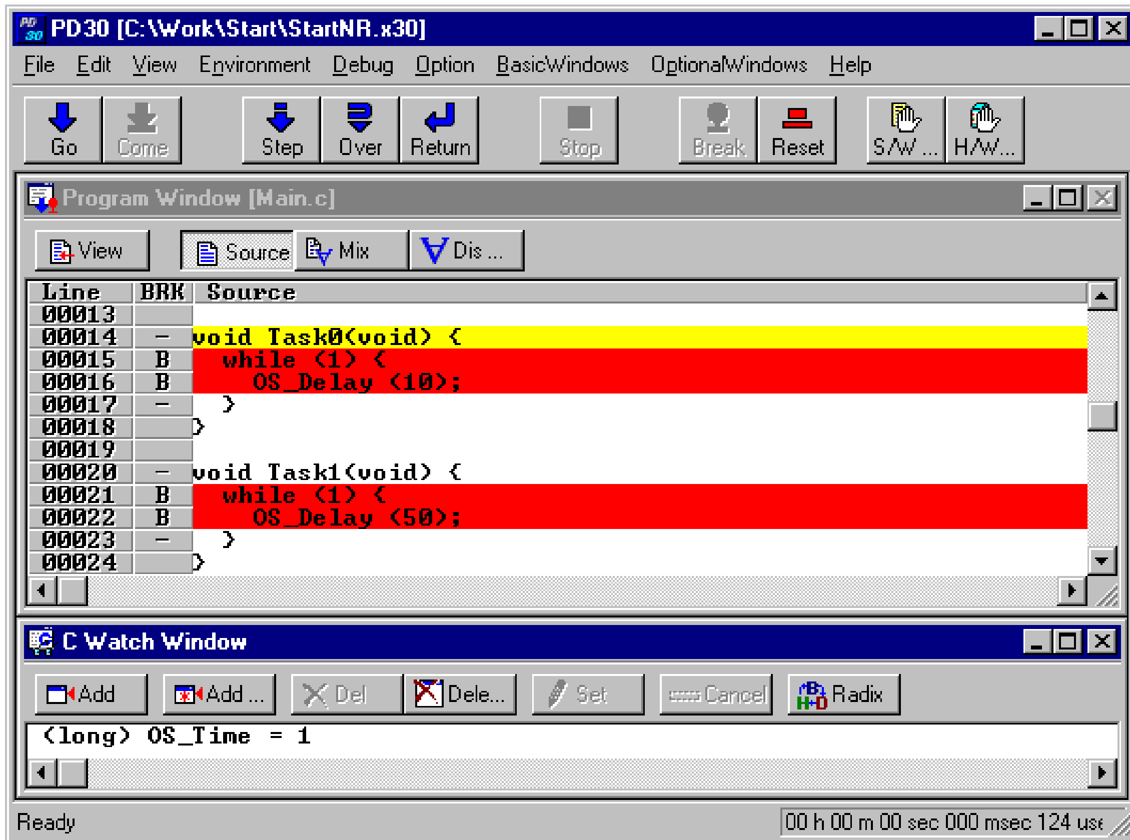


Before you step into `OS_Start()`, you should set breakpoints in the two tasks as shown below:

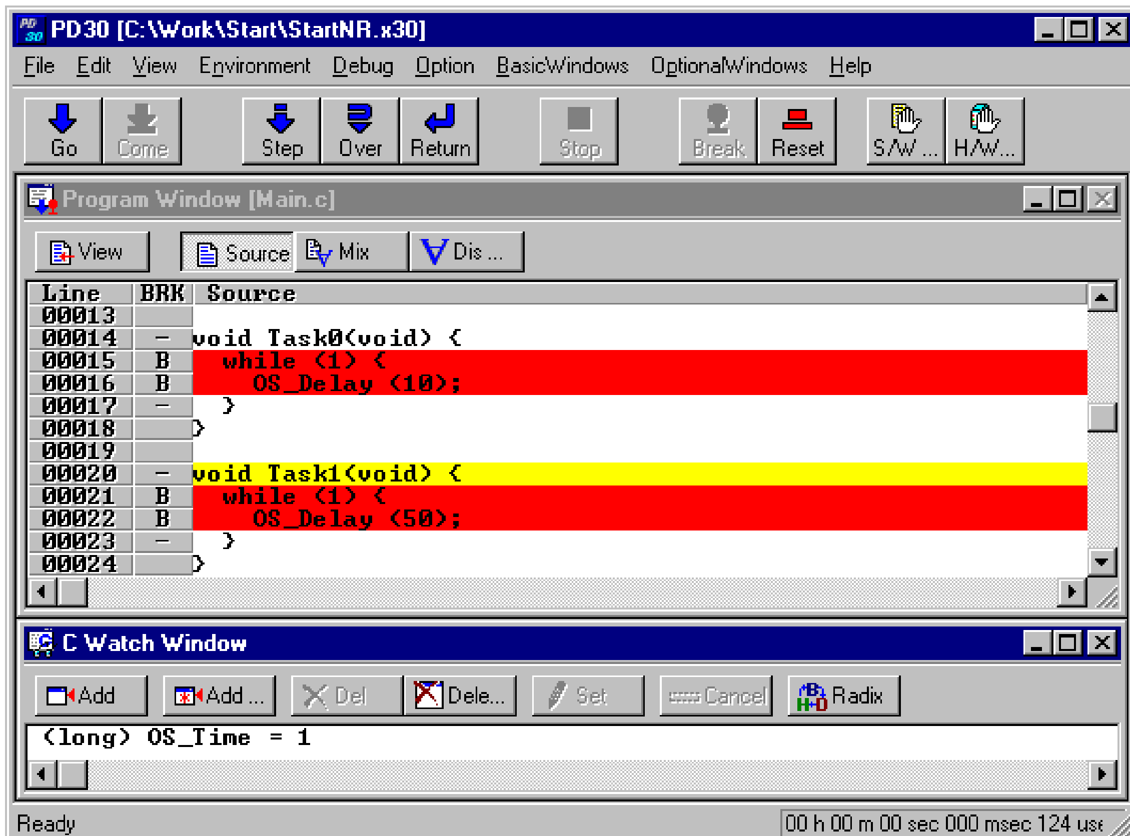


© 2013 SEGGER Microcontroller GmbH & Co. KG

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:



© 2013 SEGGER Microcontroller GmbH & Co. KG

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 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 both of our tasks, you will see that they continue execution after the given delay.
Coming from `OS_Idle()`, you should execute the 'Go' command:

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 which is shown in the watch window.

The screenshots above are taken with an older version of PD30 as those look the same for PD30, KD30 and PD30Sim.
Version 6 of PD30 looks a little bit different, but the principle of operation is the same.

4.5. Using PD30Sim

The PD30 simulator may be used to examine or debug your application. The peripherals of M16C are not simulated automatically. To simulate interrupts, you have to load an I/O script file.
The ***embOS*** distribution contains the I/O script file '`PD30_SimInt21.scr`' which simulates timer A0 interrupt, which is normally used as ***embOS*** timer interrupt.
The interrupt simulation should be started after the call of `OS_InitHW()`.
To start the simulation, proceed as follows:

- Open the I/O script window by menu 'Optional Window | IO Window'.
- From the menu in the IO Window, choose 'Load'
- Open the I/O script file 'PD30_SimInt21.scr' found in the start directory

Please note:

The simulation starts immediately, but stops as soon as the IO Window is closed. Therefore the IO Window may be minimized, but must not be closed during the debugging session.

4.6. Common debugging hints

For debugging your application, you should use a debug build, e.g. use the debug build libraries in your projects if possible. The debug build contains additional error check functions during runtime.
When an error is detected, the debug libraries call `OS_Error()`, which is defined in the separate file `OS_Error.c`.
Using an emulator you should set a breakpoint there. The actual error code is assigned to the global variable `OS_Status`. The program then waits for this variable to be reset. This allows to get back to the program-code that caused the problem easily: Simply reset this variable to 0 using your in circuit-emulator, and you can step back to the program sequence causing the problem. Most of the time, a look at this part of the program will make the problem clear.
For M16C CPUs, the error code is contained in the R0 register (refer to IAR documentation for details on the calling convention)
How to select an other library with debug code for your projects is described later on in this manual.

© 2013 SEGGER Microcontroller GmbH & Co. KG

5. Build your own application

To build your own application, you should start with the 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 *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
- **KD30Vect.asm** from subfolder Src\ for KD30 targets
  If target should be built for KD30, the interrupt vectors for KD30 UART have to be defined in your project. This is done in KD30Vect.asm.

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 which includes different targets for different output formats or debug tools. The start project was built and tested for standard M16C CPUs. For various CPU variants there may be modifications required as described later in this manual.

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.

© 2013 SEGGER Microcontroller GmbH & Co. KG

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.

© 2013 SEGGER Microcontroller GmbH & Co. KG

6. IAR compiler specifics

6.1. Data / Memory models, compiler options

embOS for M16C for IAR compiler is delivered with libraries for the most common data models and other optional settings used by IAR compiler.
The following limitations exist when standard *embOS* libraries are used:

- Calling convention Simple not supported by standard libraries.
- 64bit IEEE floating point option not supported by standard libraries

When *embOS* sources are used or recompiled with the appropriate options, all of these options may be used.

IAR compiler offers three main data models:

Data Model	Variables in	
Near pointers	far (20 bits always)	near (16 bits)
Far pointers	far (20 bits always)	far (20 bits, 64K segments) Default placement far
Huge pointers	far (20 bits always)	huge (20 bits)

6.2. Available libraries

embOS is shipped with libraries for most commonly used data model options. The library name is composed as follows:

rtos u v w x y z LM.r34

Parameter	Meaning	Values	
u	Data model	n:	Near pointers
		f:	Far pointers
		h:	Huge pointers
v	Variable placement	n:	Near memory
		f:	Far memory
		h:	Huge memory
w	Constant placement	n:	Near memory
		f:	Far memory
		h:	Huge memory
x	size of doubles	f:	32 bit
		d:	64 bit IEEE float *Note
y	Alignment of objects	w:	word aligned
		b:	byte aligned
z	Writeable strings	w:	writeable (located in RAM)
		c:	constant
LM	Library mode	R:	Release
		S:	Stack check
		SP:	Stack check + profiling
		D:	Debug
		DP:	Debug + profiling
		DT:	Debug + profiling + Trace

*Note: Not supported by standard *embOS* libraries
The following data model / variable placement options are supported by standard *embOS* libraries:

© 2013 SEGGER Microcontroller GmbH & Co. KG

Data model	Variables	Constants	Library name	Note
Near pointers	Near	Near	rtos_nnnfyw_LM.r34	Writeable strings
Near pointers	Near	Near	rtos_nnnfyc_LM.r34	
Near pointers	Near	Far	rtos_nnffyc_LM.r34	
Near pointers	Near	Huge	rtos_nnhfyc_LM.r34	
Far pointers	Near	Far	rtos_fnffyc_LM.r34	
Far pointers	Far	Far	rtos_ffffyc_LM.r34	
Huge pointers	Near	Huge	rtos_hnhfyc_LM.r34	
Huge pointers	Huge	Huge	rtos_hhhfyc_LM.r34	

y may be b for Byte aligned objects or w for word aligned objects.

LM For each data model / variable placement options, all ***embOS*** Library modes are available:

Library mode	Meaning	define
XR	Extreme Release	OS_LIBMODE_XR
R	Release	OS_LIBMODE_R
S	Stack check	OS_LIBMODE_S
SP	Stack check + Profiling	OS_LIBMODE_SP
D	Debug + stack check	OS_LIBMODE_D
DP	Debug + stack check + Profiling	OS_LIBMODE_DP
DT	Debug + stack check + profiling + Trace	OS_LIBMODE_DT

This results in 96 different libraries delivered with ***embOS***.

When using IAR workbench, please check the following points:

- The data / memory model is set as general project option
- One ***embOS*** library is part of your project (included in one group of your target). To find out the correct ***embOS*** library, you may look at XLINK options “Include, Library”. ***embOS*** libraries follow the same naming conventions as IAR runtime libraries included by linker.
- The appropriate define according to ***embOS*** library mode is set as compiler preprocessor option for your project.

6.3. Distributed project files

The distribution of ***embOS*** for M16C and IAR compiler contains one start project for IAR’s Embedded Workbench that contains targets for near memory model and SP library types.

6.4. Distributed target configurations

Different targets are included in the sample start project. The targets are named according the output format they were built for:

- **Target**: Produces an Motorola output file
- **CSpy Simulator**: Is set up for CSPY Simulator, produces the correct output file and starts embOS timer interrupt simulation.
- **Target_KD30**: Includes UART vectors used for KD30 ROM monitor. It may be used with CSpy or KD30.
- **Target_PD30**: Produces an X30 output file used for in circuit emulator.

7. M16C6N and M16C62P CPU specifics

The hardware initialization routines and default settings in RTOSInit.c were designed for M16C/62 CPUs.
M16C6N and M16C62P CPUs are equipped with additional prescaler that is activated per default after reset and divide the peripheral clock for timer and UART by two.
This results in wrong settings for ***embOS*** timer tick and baudrate for UART used for embOSView.
As far as possible, you should not modify RTOSInit.c, as this has the disadvantage, that this modifications have to be tracked when you update to a newer version of ***embOS***.

7.1. Clock settings and corrections for *embOS* timer interrupt

`OS_InitHW()` routine in `RTOSInit.c` derives timer init values from the constant define `OS_PCLK_TIMER`. Per default, the value of `OS_PCLK_TIMER` equals `OS_FSYS`, which defines the CPU clock of the target system. As M16C6N and M16C62P CPUs have additional prescaler for timer peripherals, the calculated values derived from `OS_PCLK_TIMER` are wrong, the timer will run at half the estimated speed without correction.

To correct the ***embOS*** timer tick frequency, you may:

- Reprogram the Peripheral function clock select register (`PCLKR`) at address 0x025E to disable the prescaler for timer peripherals. This should be done before calling `OS_InitHW()` either during your own target specific hardware initialization or during `__low_level_init()` which is called from startup code. The protection register bit 0 has to be set to enable modification of `PCLKR`.
- You may alternatively define `OS_PCLK_TIMER` as project option (compiler preprocessor option). This value is used to calculate values used to initialize ***embOS*** timer.

7.2. Clock settings and corrections for UART used for embOSView

`OS_COM_Init()` routine in `RTOSInit.c` derives baudrate generator init values from the constant define `OS_PCLK_UART`. Per default, the value of `OS_PCLK_UART` equals `OS_FSYS`, which defines the CPU clock of the target system. As M16C6N and M16C62P CPUs have additional prescaler for UART peripherals, the calculated values derived from `OS_PCLK_UART` are wrong, the UART will run at half the estimated speed without correction.

To correct the ***embOS*** UART baudrate for embOSView, you may:

- Reprogram the Peripheral function clock select register (`PCLKR`) at address 0x025E to disable the prescaler for UART peripherals. This should be done before calling `OS_InitHW()` either during your own target specific hardware initialization or during `__low_level_init()` which is called from startup code. The protection register bit 0 has to be set to enable modification of `PCLKR`.
- You may alternatively define `OS_PCLK_UART` as project option (compiler preprocessor option). This value is used to calculate values used to initialize UART used for communication with embOSView.

© 2013 SEGGER Microcontroller GmbH & Co. KG

7.3. PLL settings

M16C62P group CPUs are equipped with internal PLL and other clock options. Standard RTOSInit.c routines are written for CPUs without PLL.
Normally, PLL should be initialized as early as possible. You may initialize PLL in `__low_level_init()` which is called during startup code.
When using PLL, `OS_InitHW()` which initializes ***embOS*** timer may have to be modified.

7.4. Conclusion about clock settings

- **OS_FSYS** has to be defined according to your CPU clock frequency. This should be defined as compiler preprocessor option in your project.
- **OS_PCLK_TIMER** has to be defined to fit the frequency used as peripheral clock for the ***embOS*** timer. The value defaults to OS_FSYS. It should be modified and defined as compiler preprocessor option if modification is required.
- **OS_PCLK_UART** has to be defined to fit the frequency used as peripheral clock for the UART used for communication with embOSView. The value defaults to OS_FSYS. It should be modified and defined as compiler preprocessor option if modification is required.
- **PLL** settings should be checked. `OS_InitHW()` in RTOSInit.c might have to be modified, as this function modifies clock options of CPU.

© 2013 SEGGER Microcontroller GmbH & Co. KG

8. Stacks

8.1. Task stack for M16C

Every ***embOS*** task has to have its own stack. Task stacks can be located in any RAM memory location that can be used as stack by the M16C CPU.
As M16C CPUs have a 16bit stack pointer only, this may be any RAM located from 0x0400..0xFFFF.
The stack-size required is the sum of the stack-size of all routines 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, this minimum stack size is about 42 bytes in the near memory model.

8.2. System stack for M16C

The system stack size required by ***embOS*** is about 40 bytes (65 bytes in. profiling builds) However, since the system stack is also used by the application before the start of multitasking (the call to `OS_Start()`), and because software-timers also use the system-stack, the actual stack requirements depend on the application.
The stack used as system stack is the one defined as CSTACK in the linker command file (*.xcl).
A good value for the system stack is typically about 80 to 200 bytes.
The stack size itself can be set as project option under “General Options, Heap / Stack”.

8.3. Interrupt stack for M16C

The M16C CPU 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 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 M16CC.
The stack used as interrupt stack is the one defined as ISTACK in the linker command file (*.xcl).
The interrupt stack size itself can be set as project option under “General Options, Heap / Stack”.

8.4. Reducing the stack size

The stack check libraries check the used stack of every task and the system and interrupt stack also. Using embOSView the total size and used size of any stack can be examined. This may be used to reduce the stack sizes, if RAM space is a problem in your application.

© 2013 SEGGER Microcontroller GmbH & Co. KG

9. Interrupts

9.1. What happens when an interrupt occurs?

- The CPU-core receives an interrupt request
- As soon as interrupts are enabled and the processor interrupt priority level is below or equal to the interrupt priority level of the interrupting device, 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 REIT command, restoring PC, Flags and switching to User stack
- For details, please refer to the RENESAS users manual.

9.2. Defining interrupt handlers in "C"

Routines preceded by the keyword `__interrupt` save & restore the registers they modify and return with REIT.
The corresponding interrupt vector number should be written prior the function using a `#pragma` directive.
For a detailed description on how to define an interrupt routine in "C", refer to the IAR C-Compiler's user's guide.

Example

"Simple" interrupt-routine

```
#pragma vector = 21
__interrupt void ISR_Timer (void) {
  OS_EnterInterrupt();
  HandleTimer();
  OS_LeaveInterrupt();
}
```

9.3. Interrupt vector table

Normally there is no need to define a separate interrupt vector table when using IAR compiler for M16C, as interrupt routines may be written in “C” source as described above. If for some reason, you have to define a vector table as assembler file, please refer to IAR documentation or take the vector definition file KD30Vect.asm for KD30 ROM Monitor interrupt vectors as reference.

9.4. Interrupt-stack

Since the M16C 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.

© 2013 SEGGER Microcontroller GmbH & Co. KG

9.5. Fast interrupts with M16C

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.

9.6. 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 from 1 to 4. These ***embOS*** interrupt handlers have to start with `OS_EnterInterrupt()` or `OS_EnterNestableInterrupt()` and must end with `OS_LeaveInterrupt()` or `OS_LeaveNestableInterrupt()`.
- Any *Fast interrupt* (running at priorities from 5 to 7) must not call any ***embOS*** API function. Even `OS_EnterInterrupt()` and `OS_LeaveInterrupt()` must not be called.
- Interrupt handler running at low priorities (from 1 to 4) not calling any ***embOS*** API function are allowed, but must not re-enable interrupts!

The priority limit between *embOS* interrupts and Fast interrupts is fixed to 4 and can only be changed by recompiling *embOS* libraries!

© 2013 SEGGER Microcontroller GmbH & Co. KG

10. 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.

© 2013 SEGGER Microcontroller GmbH & Co. KG

11. Technical data

11.1. 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 and release build library.

Short description	ROM [byte]	RAM [byte]
Kernel	approx.1650	28
Add. Task	---	18
Add. Semaphore	---	4
Add. Mailbox	---	12
Add. Timer	---	12
Power-management	---	---

12. Files shipped with *embOS* M16C for IAR compiler

embOS for M16C and IAR compiler is shipped with documentation in PDF format and release notes as html.
The start project, 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\	`Start.eww`	Start workspace for IAR Embedded Workbench.
Start\	`Start.ewp`	Start project for IAR Embedded Workbench.
Start\	`Cspy.mac`	***embOS*** timer Interrupt simulation macro for IAR C-Spy simulator
Start\	`PD30_SimInt21.scr`	IO script file for ***embOS*** timer interrupt simulation under PD30Sim
Start\Inc\	`RTOS.h`	***embOS*** API header file. To be included in any file using ***embOS*** functions
Start\Lib\	`*.r34`	***embOS*** libraries
Start\Src\	`main.c`	Frame program to serve as a start
Start\Src\	`RtosInit.c`	Hardware setup functions used for ***embOS***
Start\Src\	`OS_Error.c`	***embOS*** runtime error handler.
Start\Src\	`KD30Vect.asm`	Interrupt vector definition sample for KD30 ROM monitor.
GenOsSrc\	`*.*`	***embOS*** sources (Source version only)
	`*.Bat`	Batch files to build ***embOS*** libraries from sources (Source version only)

embOSView and the manuals are found in the root directory of the distribution.

© 2013 SEGGER Microcontroller GmbH & Co. KG

13. Index

_

__low_level_init()........................21

C

Clock settings...............................22
Clock settings, timer interrupt......21
Clock settings, UART..................21
C-Spy...8
CSTACK.....................................23

F

Fast interrupt................................25

I

Installation.....................................5
Interrupt priority..........................25
Interrupt stack..............................23
Interrupt vector table....................24
Interrupt, fast................................25
Interrupts......................................24
Interrupt-stack..............................24
ISTACK.......................................23

K

KD30...11
KD30Vect.asm............................11

M

M16C62P.....................................21
M16C6N......................................21
Memory models...........................19
Memory requirements..................27

O

OS_Error()...........................16, 17
OS_FSYS..............................21, 22
OS_PCLK_TIMER................21, 22
OS_PCLK_UART..................21, 22

P

PCLKR..21
PD30..12
PD30Sim.....................................16
PLL settings.................................22

R

ROM Monitor..............................11

S

Stacks..23
Stacks, interrupt stack..................23
Stacks, system stack.....................23
Stacks, task stacks.......................23
Stop-mode...................................26
System stack................................23

T

Task stacks..................................23
Technical data..............................27

W

Wait-mode...................................26

© 2013 SEGGER Microcontroller GmbH & Co. KG