embOS

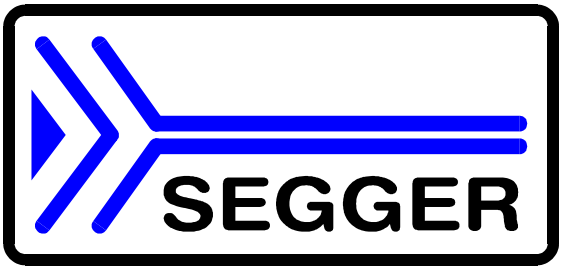
Real Time Operating System

CPU & Compiler specifics for

Mitsubishi M16C CPUs

and TASKING compiler

Document Rev. 2



A product of Segger Microcontroller Systeme GmbH

www.segger.com

Contents

1. About this document

This guide describes how to use ***embOS*** for M16C Real Time Operating System for the Mitsubishi M16C series of microcontroller using TASKING compiler and TASKING EDE.

1.1. How to use this manual

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

2. Using *embOS* with TASKING EDE

2.1. Installation

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

In order to install it, proceed as follows:

If you received a CD, copy the entire contents to your hard-drive into any folder of your choice. When copying, please keep all files in their respective sub directories. Make sure the files are not read only after copying.
If you received a zip-file, please extract it to any folder of your choice, preserving the directory structure of the zip-file.

Assuming that you are using TASKING EDE to develop your application, no further installation steps are required. You will find a prepared start project workspace 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 TASKING EDE for your application development in order to become familiar with ***embOS.***

embOS does in no way rely on TASKING EDE, it may be used without the workbench using batch files or a make utility without any problem.

2.2. First steps

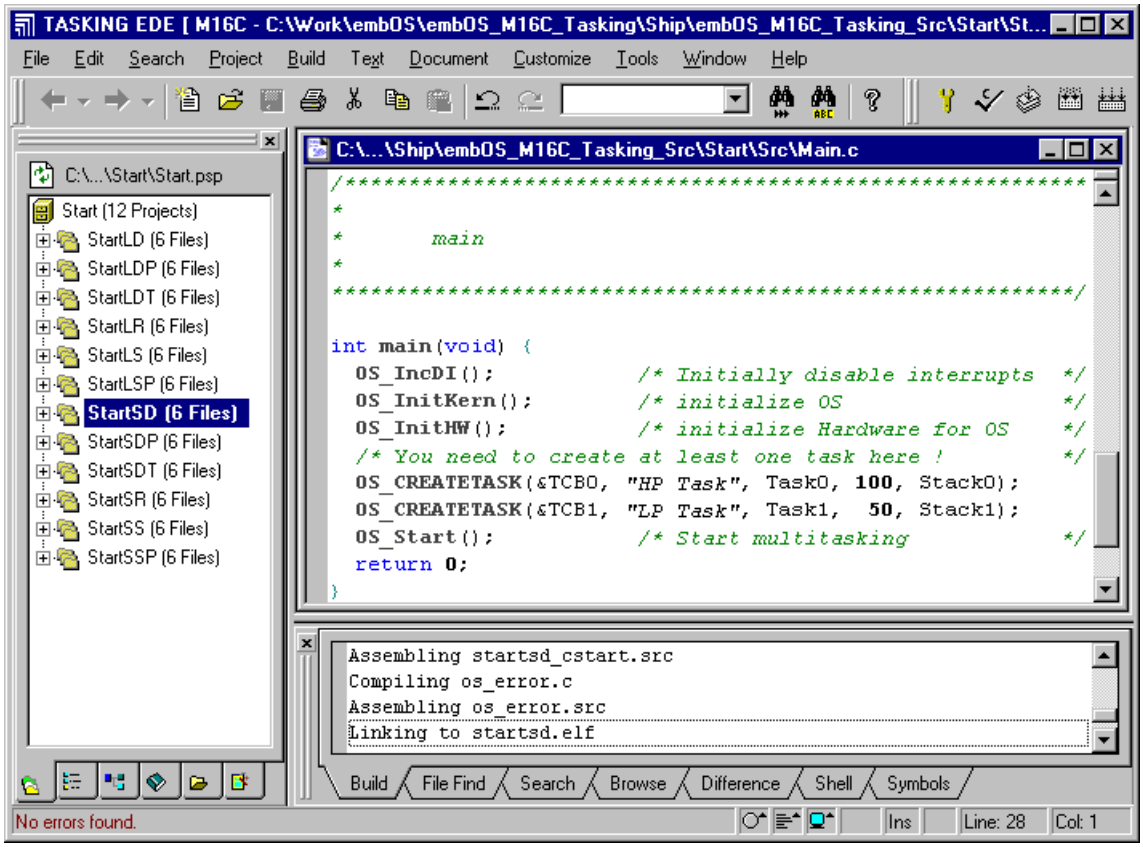
After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received a ready to go sample start project workspace for M16C CPUs and it is a good idea to use this as a starting point of all your applications.

Your ***embOS*** distribution contains one folder 'Start' which contains the sample start project workspace, the start projects 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 project workspace 'Start.psp'. (e.g. by double clicking it)
- Select and build one start project

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



Initially a target for small memory model with debug and stack check functionality is selected. You may select any other project, as anyone will run on your simulator or debugger.

2.3. 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.

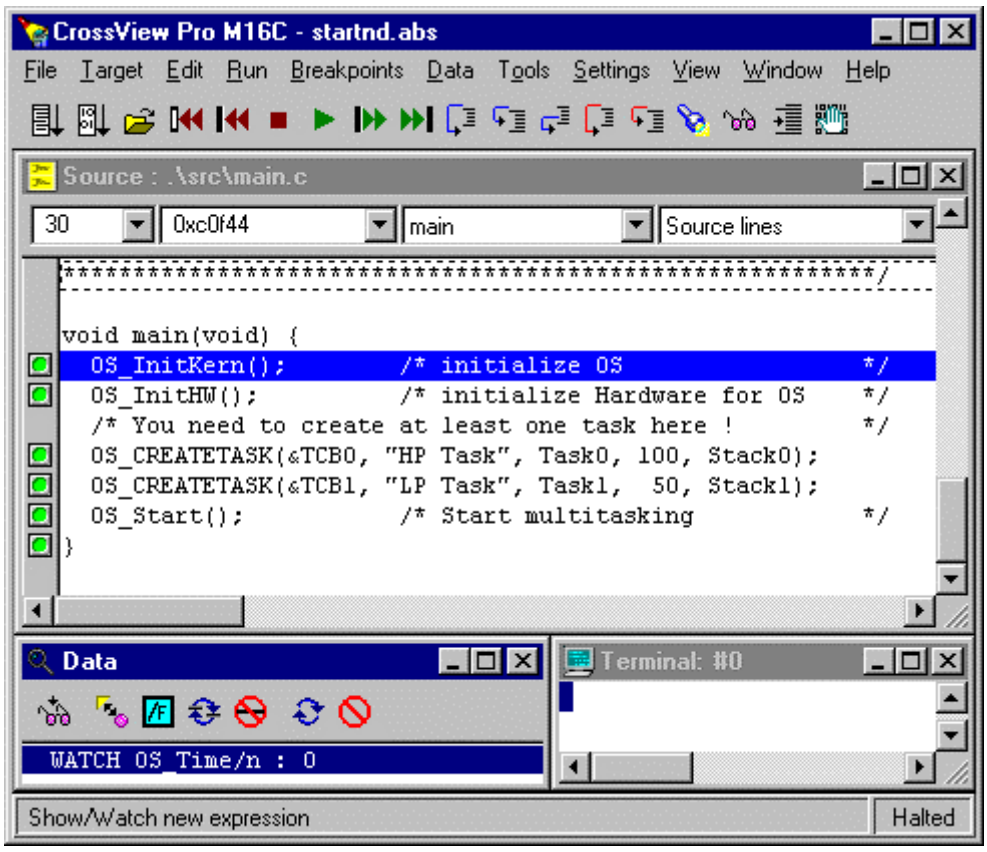
```
/*********************************************************
*          SEGGER MICROCONTROLLER SYSTEME GmbH
*   Solutions for real time microcontroller applications
**********************************************************
File    : Main.c
Purpose : Skeleton program for embOS
--------- END-OF-HEADER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

#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. Using debugging tools to debug the application

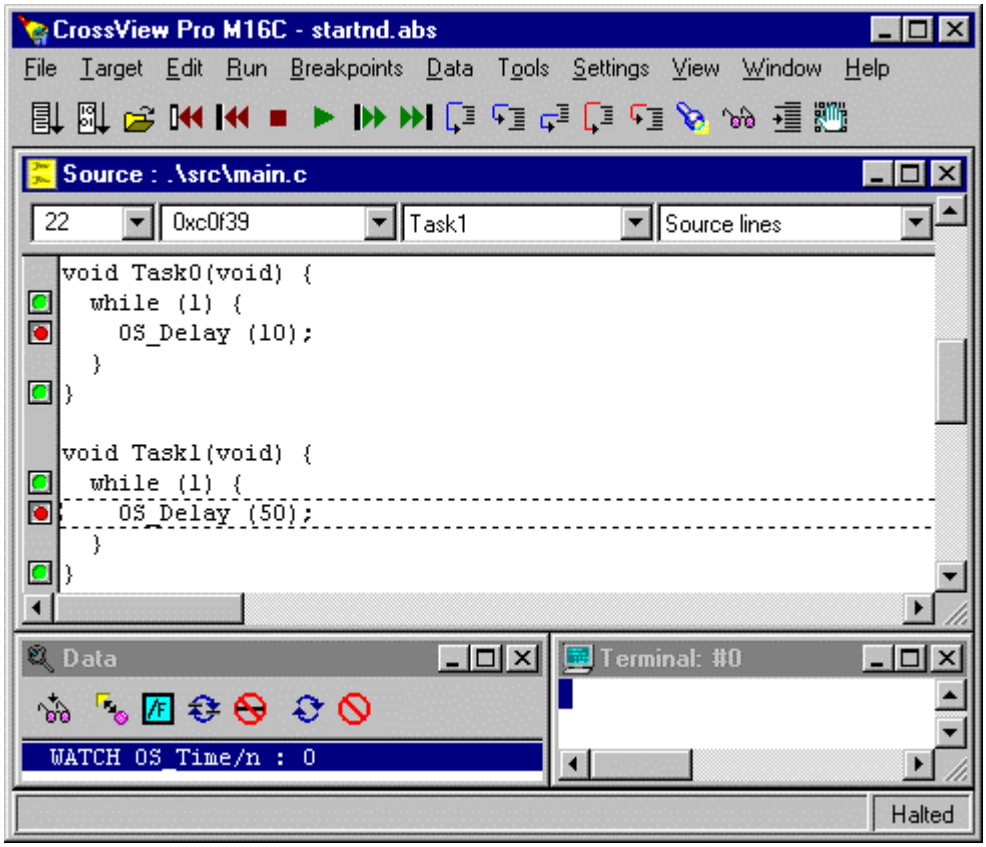
The ***embOS*** start projects produce Intel-Hex output files and ELF files, which may be used for debugging tools.
Simulation using Crossview Pro simulator is supported.
The following chapter describe a sample session based on our sample application main.

3.1. Using Crossview Pro simulator

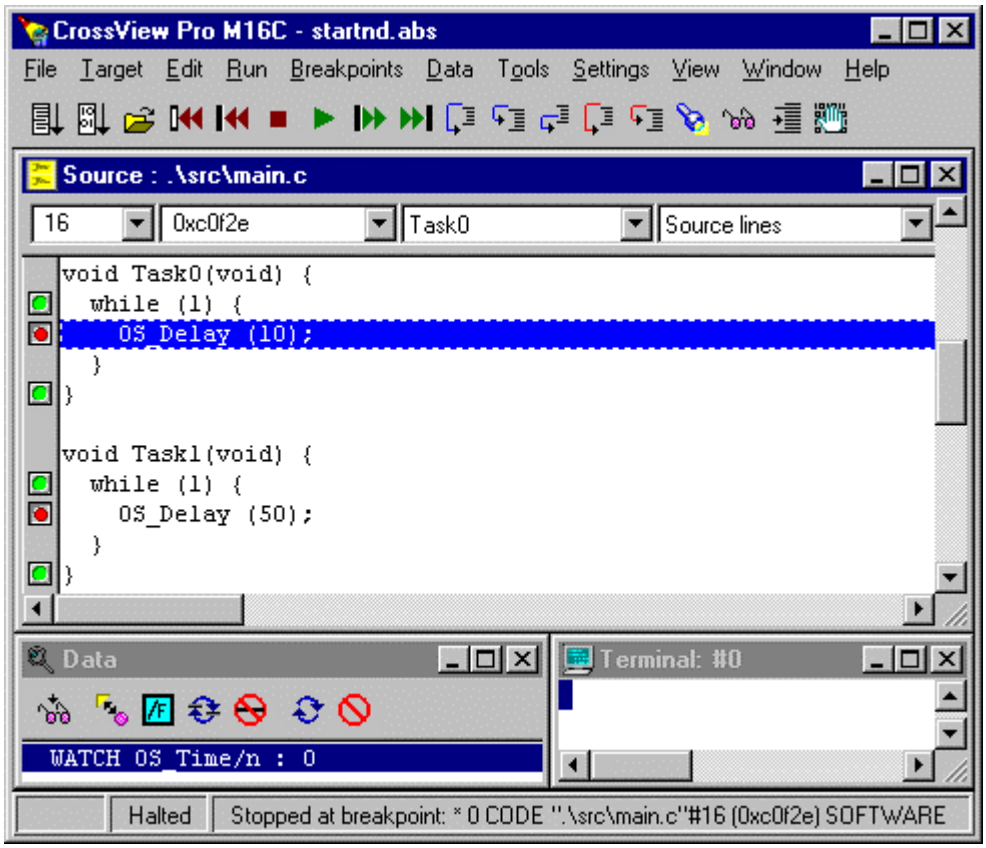
When starting Crossview Pro simulator after building the start project, you will usually see the main function, or you may look at the startup code and have to set a breakpoint at main. Now you can step through the program.
`OS_IncDI()` initially disables interrupts and tells ***embOS*** not to re-enable interrupts during internal initialization.
`OS_InitKern()`initializes ***embOS*** -Variables and enables interrupts. 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 UART reception from embOSView can not be simulated, OS_UART may be defined as (-1) to disable UART initialization and communication.
`OS_Start()` is the last line in main which is executed, since it starts multitasking and does not return.



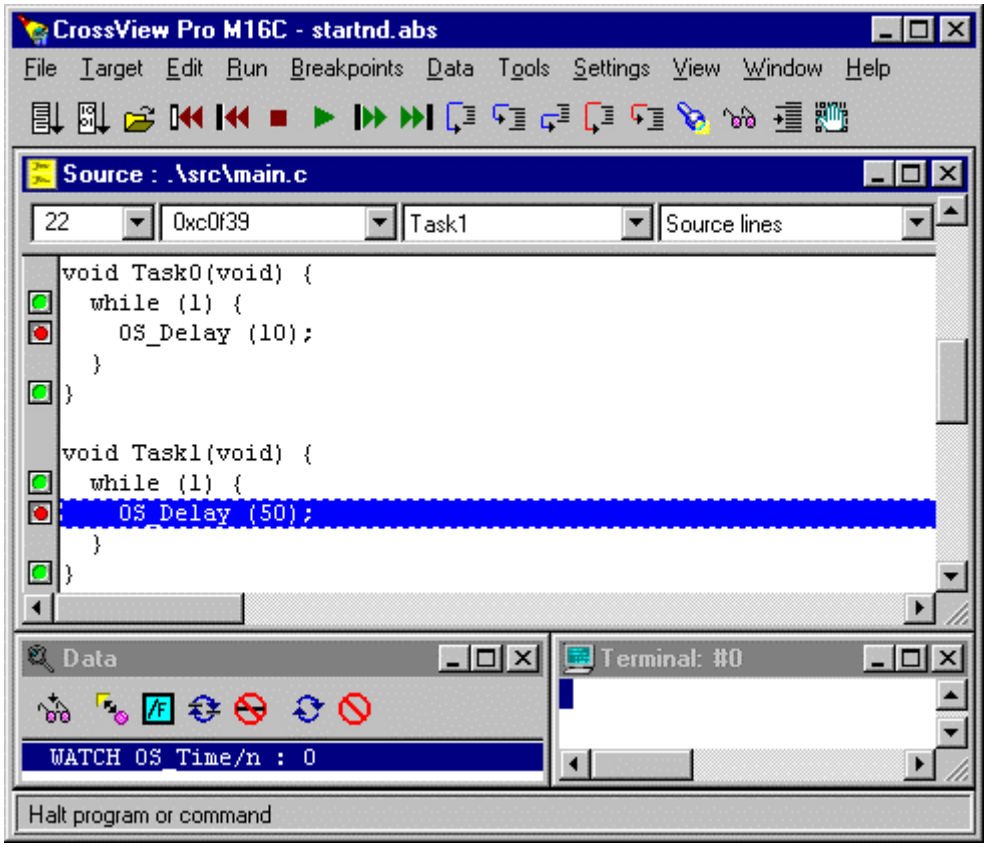
Before stepping over `OS_Start()`, you should set two breakpoint in our tasks in main.c as shown below:



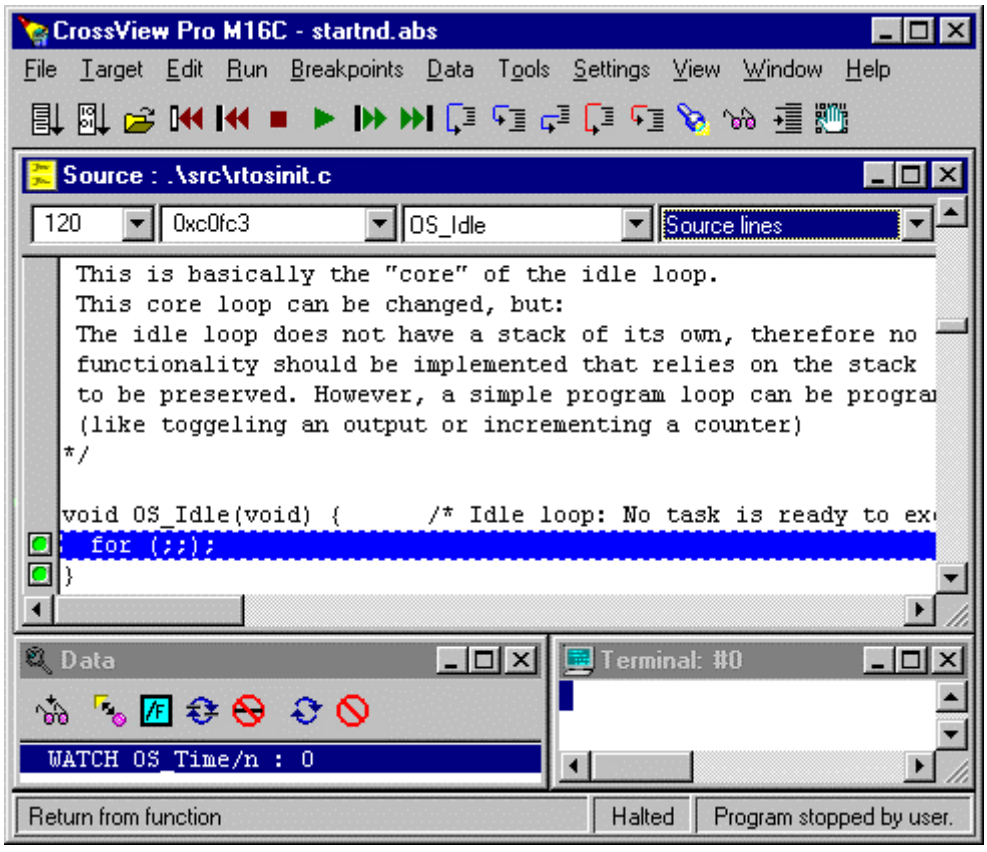
When you step over `OS_Start()` the next source line executed is Task0 which is the task with the highest priority in our start project and is therefore activated.



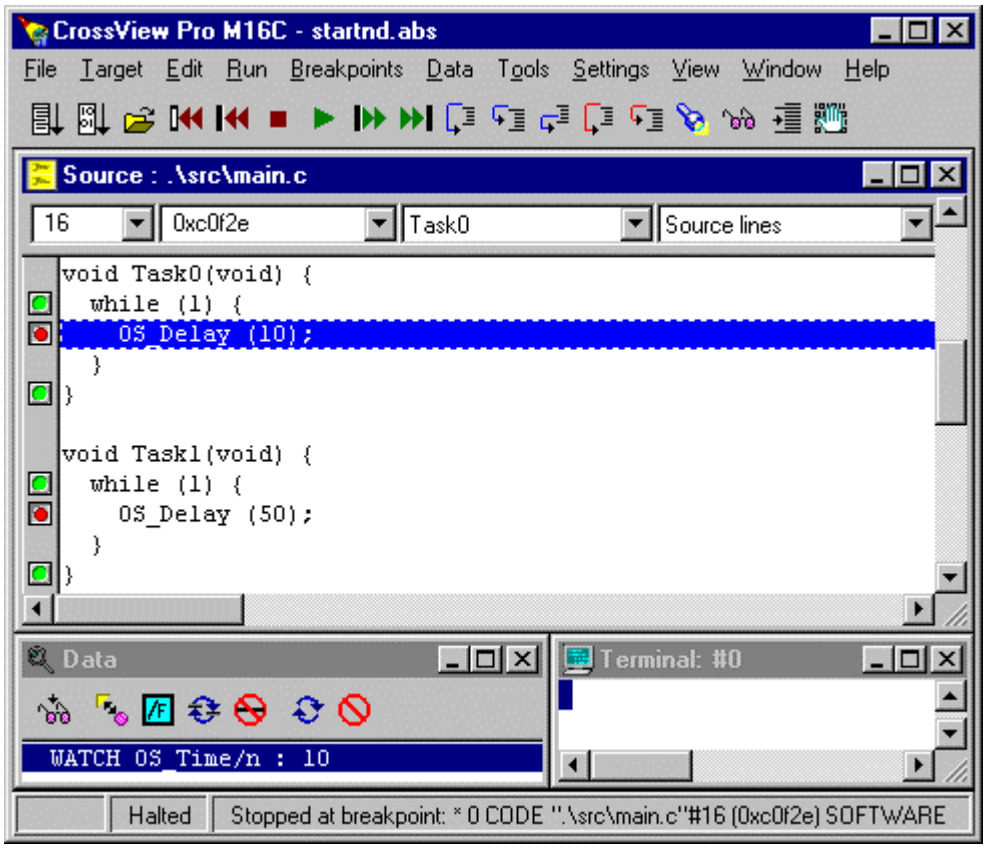
If you continue stepping, you will arrive in the task with the lower priority:



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. You will arrive there when you set a breakpoint there, or by stepping through the task switch process in assembly mode:



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.

3.2. 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 part of RTOSInit.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.
How to select an other library with different runtime debug options for your project is described later on in this manual.

4. Build your own application

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

4.1. Required files for an *embOS* application

To build an application using ***embOS***, the following files from your ***embOS*** distribution are required and have to be included in your project:

- **RTOS.h** from sub folder Inc\
  This header file declares all ***embOS*** API functions and data types and has to be included in any source file using embOS functions.
- **RTOSInit.c** from subfolder Src\.
  It contains hardware dependent initialization code for ***embOS*** timer and optional UART for embOSView.
- One ***embOS* library** from the Lib\ subfolder
- **OS_Error.c** from subfolder Src\.
  It contains the ***embOS*** runtime error handler `OS_Error()`which is used in stack check or debug builds.

When you decide to write your own startup code, or modify the project options, 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*** function except `OS_IncDI()` is called.

4.2. Select a start project

embOS comes with one start project workspace which includes start projects for small and large memory model and all different ***embOS*** library types. All projects are tested with standard M16C/62 CPUs. For some CPU variants there may be modifications required as described later in this manual.

4.3. Add your own code

For your own code, you may add a new project to the workspace.
We strongly recommend to modify an existing project, as this ensures, that all necessary settings are already done. You may modify or replace the main.c source file in the subfolder src\.

4.4. Change memory model or library mode

For your application you may have to choose an other 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.
Just select the project which is set up for your needs and add your own code there.
When you built a new project on your own, check project options about memory model settings and compiler settings according library mode used. Refer to chapter 5 about the library naming conventions to select the correct library.

5. TASKING compiler specifics

5.1. Data / Memory models, compiler options

embOS for M16C for TASKING compiler is delivered with libraries for small, medium and large model and most common options supported by TASKING compiler.
When ***embOS*** sources are used or recompiled with the appropriate options, all options may be used.

TASKING compiler offers three memory models:

Model	Data	Constants	Pointers
Small	__near: in first 64 KB	__near	__near
Medium	__near: in first 64 KB	__paged	__paged
Large	__far: anywhere in 1 MB	__far	__far

5.2. Available libraries

embOS is shipped with libraries for all memory models and different runtime debug capabilities:

Memory model	Library type	Library	define
Small	Release	rtosSR	OS_LIBMODE_R
Small	Stack-check	rtosSS	OS_LIBMODE_S
Small	Stack-check + Profiling	rtosSSP	OS_LIBMODE_SP
Small	Debug	rtosSD	OS_LIBMODE_D
Small	Debug + Profiling	rtosSDP	OS_LIBMODE_DP
Small	Debug + Profiling + Trace	rtosSDT	OS_LIBMODE_DT
Medium	Release	rtosMR	OS_LIBMODE_R
Medium	Stack-check	rtosMS	OS_LIBMODE_S
Medium	Stack-check + Profiling	rtosMSP	OS_LIBMODE_SP
Medium	Debug	rtosMD	OS_LIBMODE_D
Medium	Debug + Profiling	rtosMDP	OS_LIBMODE_DP
Medium	Debug + Profiling + Trace	rtosMDT	OS_LIBMODE_DT
Large	Release	rtosLR	OS_LIBMODE_R
Large	Stack-check	rtosLS	OS_LIBMODE_S
Large	Stack-check + Profiling	rtosLSP	OS_LIBMODE_SP
Large	Debug	rtosLD	OS_LIBMODE_D
Large	Debug + Profiling	rtosLDP	OS_LIBMODE_DP
Large	Debug + Profiling + Trace	rtosLDT	OS_LIBMODE_DT

When using TASKING EDE or compiler with make utilities, please check the following points:

- The memory model is set as general project (compiler) option
- One ***embOS*** library is included in your project.
- The appropriate define is set as compiler option for your project.

5.3. Distributed project files

The distribution of ***embOS*** containes one start project workspace and projects for small and large memory model with any available library type in the ‘start’ subdirectory. The naming convention of these projects follows the convention for the library names: Startxxx <=> rtosxxx.

6. 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 may result 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***.

6.1. 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 enabled after reset, the value `OS_PCLK_TIMER` has to be corrected.
In RTOSInit.c, this correction is done, if one of the defines like:
_REGM*_SFR,
is valid
One of these defines is normally set automatically by TASKING EDE, if one of the related CPUs is selected.
When you do not use TASKING EDE, or you did not select one of these CPUs but still use one of them, you may correct the timer init value calculation as follows:

- 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()` during your own target specific hardware initialization. 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.

6.2. 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 enabled after reset, value of `OS_PCLK_UART` can not be set to OS_FSYS and has to be corrected. Without correction, the UART will run at half the estimated speed without correction.
In RTOSInit.c, this correction is done, if one of the defines like:
_REGM*_SFR,
is valid
One of these defines is normally set automatically by TASKING EDE, if one of the related CPUs is selected.
When you do not use TASKING EDE, or you did not select one of these CPUs but still use one of them, you may correct the timer init value calculation as follows:

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()` during your own target specific hardware initialization. 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.

6.3. PLL settings

M16C62P group CPUs are equipped with internal PLL and additional 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 before calling `OS_InitHW()` during your own hardware initialization. You may also modify `OS_InitHW()` to initialize PLL.
When using PLL, `OS_InitHW()` which initializes ***embOS*** timer may have to be modified.

6.4. 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. Normally this is done automatically by defining the CPU in TASKING EDE.
- **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. Normally modification according to CPU type used, is done automatically by defining the CPU in TASKING EDE.
- **PLL** settings should be checked. `OS_InitHW()` in RTOSInit.c might have to be modified, as this function modifies clock options of CPU.

You may also check the output listings of RTOSInit.c to examine whether OS_PCLK_TIMER and OS_PCLK_UART are set as required.

7. Stacks

7.1. 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.
M16C USP is used for task stacks.

7.2. System stack for M16C

The system is the stack used during startup and main until OS_Start() is called. The startup code initializes M16Cs ISP as system stack.
The 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 and embOS internal functions during task switch also use the system-stack, the actual stack requirements depend on the application.
System stack size is defined in the Linker-option file, or is set as option from the EDE.
A good value for the system stack size is at minimum 200 bytes.

7.3. 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 M16C.

7.4. 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.

8. Interrupts

8.1. 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 previous stack
- For details, please refer to the Mitsubishi users manual.

8.2. Defining interrupt handlers in "C"

Routines preceded by the keyword `__interrupt(n)` save & restore the registers they modify and return with `REIT`.
By specifying the corresponding interrupt vector number (parameter n) in the range of 0..63, the interrupt function is automatically inserted in the interrupt vector table, unless this feature is not disabled by project settings.
For a additional information refer to the TASKING C-Compiler's user's guide.

Example

```
__interrupt (18) void UART_ISR_rx(void) {
  int Data = U0RB;
  if (Data & 0x6000) {    /* Check if errors occurred */
    U0C1 &= 255-(1<<2);   // disable Rx
    U0C1 |=  (1<<2);      // enable Rx => error reset
  } else {
    HandleRx(Data);
  }
}
```

8.3. Interrupt vector table

Normally there is no need to define a separate interrupt vector table when using TASKING 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 TASKING documentation.

8.4. Interrupt priorities

Any interrupt priority may be used for any interrupt handler.
embOS internal interrupts for timer tick and UART for communication with embOSView run on lowest interrupt priority per default.

9. STOP / WAIT Mode

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

10. Technical data

10.1. Memory requirements

These values are neither precise nor guaranteed but they give you a good idea of the memory-requirements. They vary depending on the current version of ***embOS***. The values in the table are for the near memory model and release build library.

Short description	ROM [byte]	RAM [byte]
Kernel	approx.1680	29
Add. Task	---	17
Add. Semaphore	---	4
Add. Mailbox	---	11
Add. Timer	---	11
Power-management	---	---

11. Files shipped with ***embOS*** M16C

embOS for M16C and TASKING 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\	`Start.psp`	Start project workspace for TASKING EDE.
Start\	`Start*.prj`	Start projects
Start\	`CLEAN.BAT`	Batch file to erase outputs and temporary files.
Start\Inc\	`RTOS.h`	***embOS*** API header file. To be included in any file using ***embOS*** functions
Start\Lib\	`rtos*.a`	***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*** error handler. Used for all builds with runtime error check functionality.
GenOsSrc\	`*.*`	***embOS*** sources (Source version only)
CPU\	`*.*`	CPU specific sources (Source version only)
	`*.Bat`	Batch files to build ***embOS*** libraries from sources (Source version only)

embOSView, Release notes and manuals are found in the root directory of the distribution.
Additional files serve as sample

12. Index