embOS

Real Time Operating System

CPU & Compiler specifics for C166/C167 core with TASKING Tools v7.5r5

Document Rev. 1

SEGGER

A product of Segger Microcontroller Systeme GmbH

www.segger.com

Contents

1. About this document

This guide describes how to use embOS Real Time Operating System for the C166/C167 series of microcontrollers using *TASKING Tools*.

1.1. How to use this manual

This manual describes all CPU and compiler specifics for embOS using C166/C167 based controllers with *TASKING Tools*. 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 Tools*. If you have no experience using ***embOS***, you should follow this introduction, because it is the easiest way to learn how to use ***embOS*** in your application.
Most of the other chapters in this document are intended to provide you with detailed information about functionality and fine-tuning of embOS for the C166/C167 based controllers using *TASKING Tools*.

2. Using ***embOS*** with TASKING Tools

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 Tools'* project manager to develop your application, no further installation steps are required. You will find a prepared sample start application, which you should use and modify to write your application. So follow the instructions of the next chapter 'First steps'.

You should do this even if you do not intend to use the project manager for your application development in order to become familiar with ***embOS.***

If for some reason you will not work with the project manager, 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 *TASKING Tools'* project manager, it may be used without the project manager using batch files or a make utility without any problem.

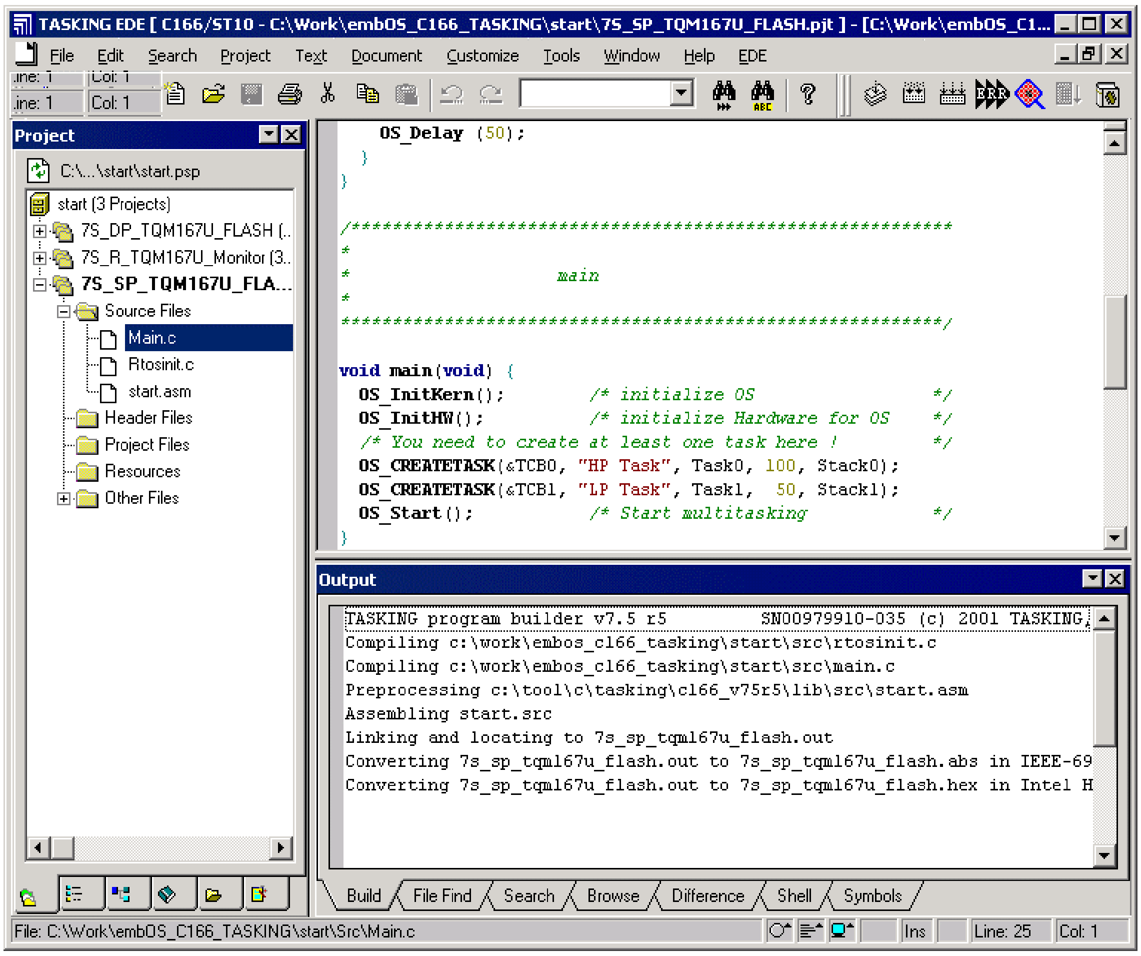
2.2. First steps

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

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

Create a work directory for your application, for example c:\work
Copy the whole folder 'Start' which is part of your ***embOS*** distribution into your work directory
Clear the read only attribute of all files in the new 'start' folder.
Open the sample project start\start.psp with *TASKING Tools* project manager (e.g. by double clicking it)
Build the start project

Your screen should look like follows:


TASKING EDE [C166/ST10 - C:\Work\embOS_C166_TASKING\start\7S_SP_TQM167U_FLASH.pjt] - [C:\Work\embOS_C1...
File Edit Search Project Text Document Customize Tools Window Help EDE
Line: 1 Col: 1
Project
C:\...\start\start.psp
start (3 Projects)
7S_DP_TQM167U_FLASH (..
7S_R_TQM167U_Monitor (3..
7S_SP_TQM167U_FLA...
Source Files
Main.c
Rtosinit.c
start.asm
Header Files
Project Files
Resources
Other Files
Output
Build File Find Search Browse Difference Shell Symbols
File: C:\Work\embOS_C166_TASKING\start\Src\Main.c
Ins Line: 25 Col: 1

```
    OS_Delay (50);
  }
}

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

void main(void) {
  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            */
}
```

```
TASKING program builder v7.5 r5        SN00979910-035 (c) 2001 TASKING,
Compiling c:\work\embos_c166_tasking\start\src\rtosinit.c
Compiling c:\work\embos_c166_tasking\start\src\main.c
Preprocessing c:\tool\c\tasking\c166_v75r5\lib\src\start.asm
Assembling start.src
Linking and locating to 7s_sp_tqm167u_flash.out
Converting 7s_sp_tqm167u_flash.out to 7s_sp_tqm167u_flash.abs in IEEE-69
Converting 7s_sp_tqm167u_flash.out to 7s_sp_tqm167u_flash.hex in Intel H
```

For latest information you should open the file start\ReadMe.txt.

2.3. The sample application Main.c

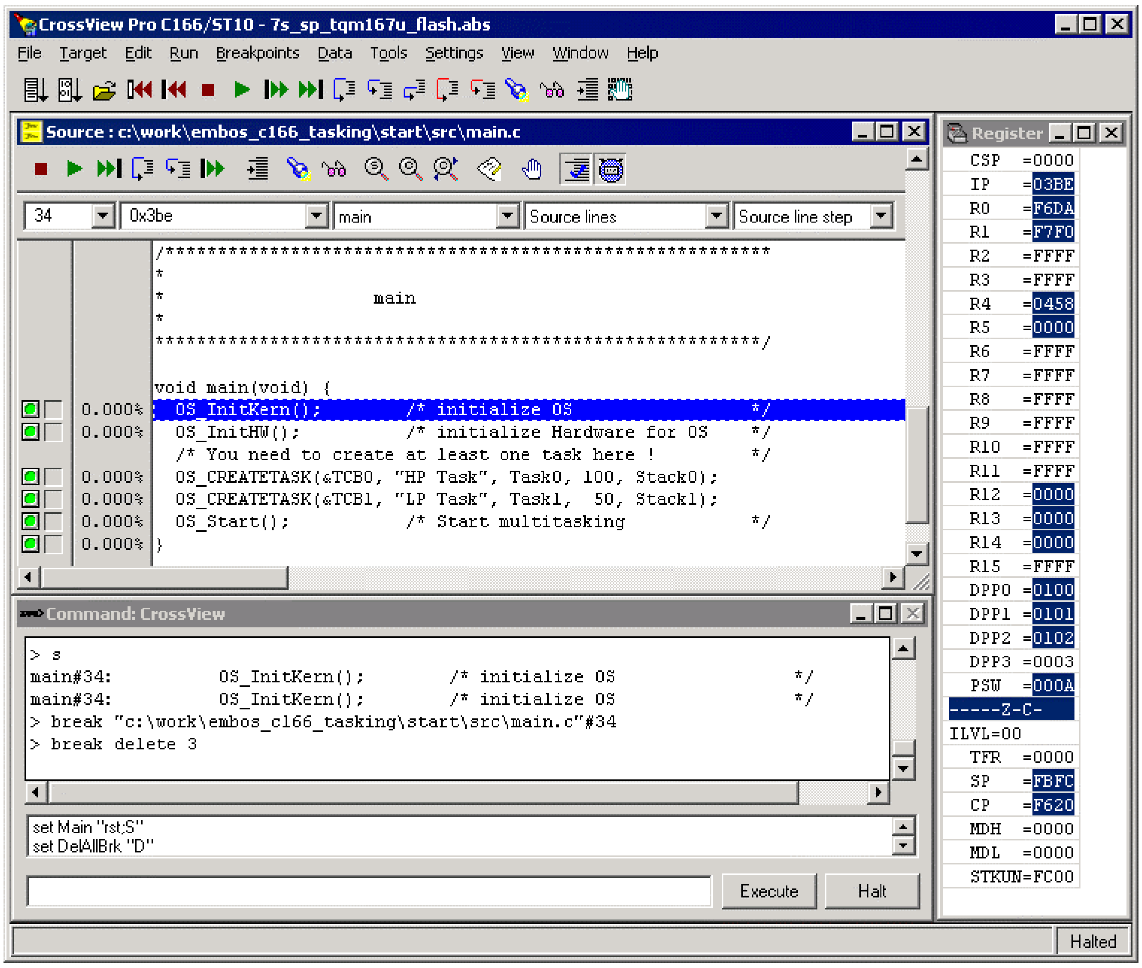
The following is a printout of the sample application main.c. It is a good starting-point for your application. (Please note that the file actually shipped with your port of ***embOS*** may look slightly different from this one)
What happens is easy to see:
After initialization of ***embOS;*** two tasks are created and started
The 2 tasks are activated and execute until they run into the delay, then suspend for the specified time and continue execution.

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

#include "RTOS.H"

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


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

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

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

void main(void) {
  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_SendString("Start project will start multitasking !\n");
  OS_Start();           /* Start multitasking */
}
```

2.4. Stepping through the sample application Main.c using Cross-View Debugger

When starting the debugger, you will usually see the main function (very similar to the screenshot below). Depending on your debugger settings, 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 the relevant OS-Variables and enables interrupts. If you do not like this behavior, you are free to change it by incrementing the interrupt-disable counter using `OS_IncDI()` before the call to `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 thru it to see what is done.
`OS_COM_Init()` 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.


CrossView Pro C166/ST10 - 7s_sp_tqm167u_flash.abs
File Target Edit Run Breakpoints Data Tools Settings View Window Help
Source : c:\work\embos_c166_tasking\start\src\main.c
34
0x3be
main
Source lines
Source line step
Command: CrossView
Execute
Halt
Register
Halted

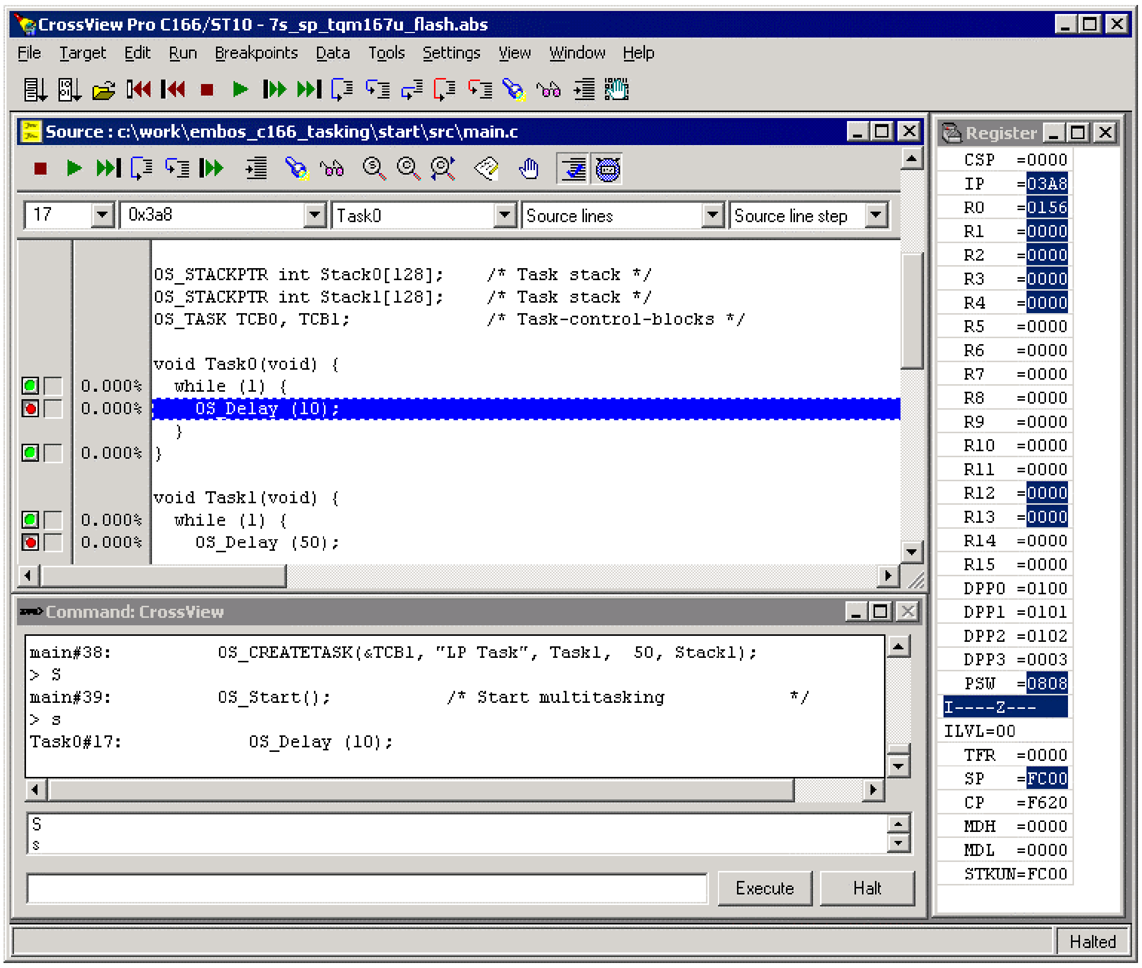
```
/*****************************************************************
*
*                    main
*
*****************************************************************/

void main(void) {
  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            */
}
```

```
> s
main#34:          OS_InitKern();        /* initialize OS                 */
main#34:          OS_InitKern();        /* initialize OS                 */
> break "c:\work\embos_c166_tasking\start\src\main.c"#34
> break delete 3
```

```
set Main "rst;S"
set DelAllBrk "D"
```

```
  CSP   =0000
  IP    =03BE
  R0    =F6DA
  R1    =F7F0
  R2    =FFFF
  R3    =FFFF
  R4    =0458
  R5    =0000
  R6    =FFFF
  R7    =FFFF
  R8    =FFFF
  R9    =FFFF
  R10   =FFFF
  R11   =FFFF
  R12   =0000
  R13   =0000
  R14   =0000
  R15   =FFFF
  DPP0  =0100
  DPP1  =0101
  DPP2  =0102
  DPP3  =0003
  PSW   =000A
-----Z-C-
ILVL=00
  TFR   =0000
  SP    =FBFC
  CP    =F620
  MDH   =0000
  MDL   =0000
  STKUN=FC00
```

Before you step into OS_Start(), set one break point in Task0 and one in Task1. When you step into OS_Start(), you will only step into it in disassembly mode, because this function is part of the ***embOS*** library. However, you can press GO now or step in disassembly mode until you reach the highest priority task.

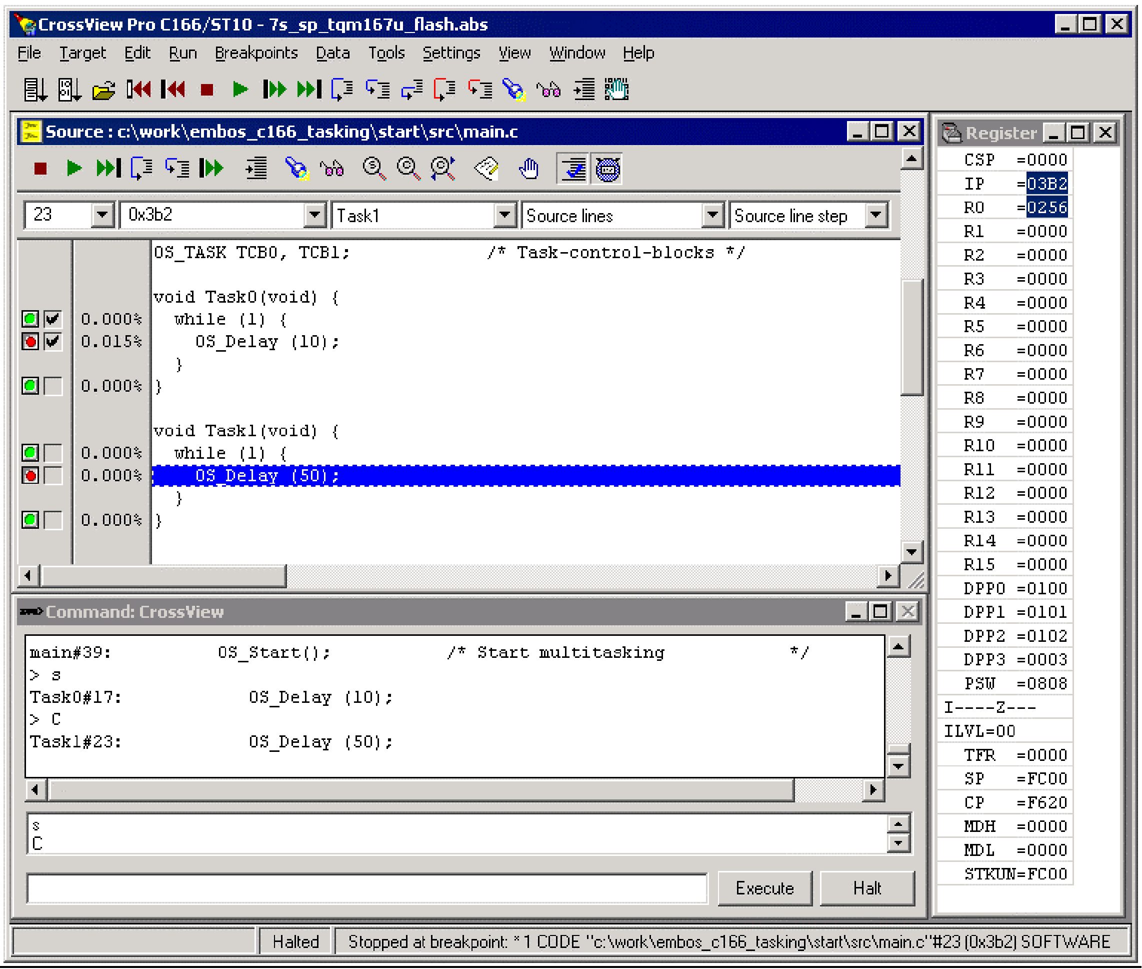

CrossView Pro C166/ST10 - 7s_sp_tqm167u_flash.abs
File Target Edit Run Breakpoints Data Tools Settings View Window Help
Source : c:\work\embos_c166_tasking\start\src\main.c
17
0x3a8
Task0
Source lines
Source line step
Command: CrossView
Execute
Halt
Register
Halted

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

void Task0(void) {
0.000%    while (1) {
0.000%      OS_Delay (10);
          }
0.000%  }

void Task1(void) {
0.000%    while (1) {
0.000%      OS_Delay (50);
```

```
main#38:          OS_CREATETASK(&TCB1, "LP Task", Task1,  50, Stack1);
> S
main#39:          OS_Start();           /* Start multitasking          */
> s
Task0#17:             OS_Delay (10);
```

```
CSP   =0000
IP    =03A8
R0    =0156
R1    =0000
R2    =0000
R3    =0000
R4    =0000
R5    =0000
R6    =0000
R7    =0000
R8    =0000
R9    =0000
R10   =0000
R11   =0000
R12   =0000
R13   =0000
R14   =0000
R15   =0000
DPP0  =0100
DPP1  =0101
DPP2  =0102
DPP3  =0003
PSW   =0808
I----Z---
ILVL=00
TFR   =0000
SP    =FC00
CP    =F620
MDH   =0000
MDL   =0000
STKUN=FC00
```

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


CrossView Pro C166/ST10 - 7s_sp_tqm167u_flash.abs
File Target Edit Run Breakpoints Data Tools Settings View Window Help
Source : c:\work\embos_c166_tasking\start\src\main.c
23
0x3b2
Task1
Source lines
Source line step
Command: CrossView
Execute
Halt
Register
Halted

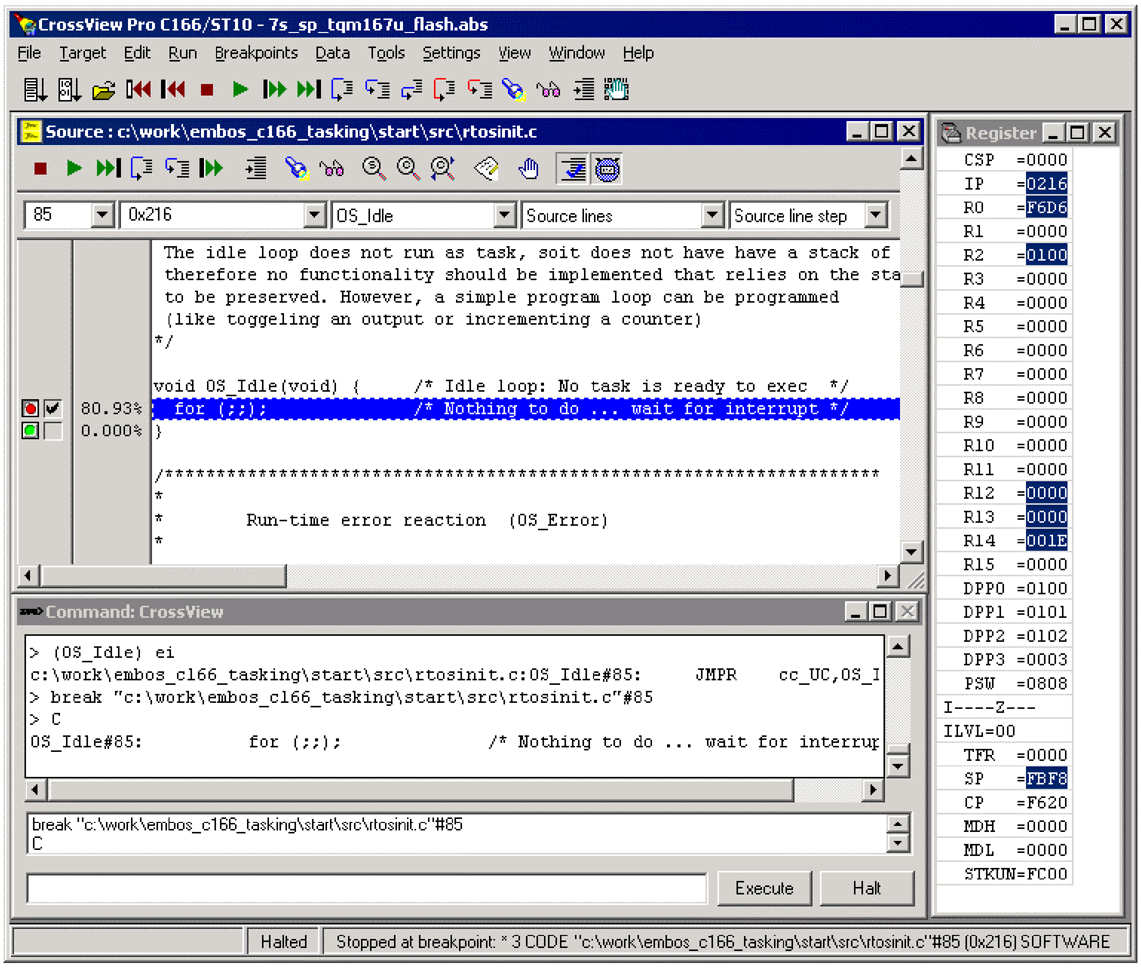
```
OS_TASK TCB0, TCB1;             /* Task-control-blocks */

void Task0(void) {
0.000%    while (1) {
0.015%      OS_Delay (10);
          }
0.000%  }

void Task1(void) {
0.000%    while (1) {
0.000%      OS_Delay (50);
          }
0.000%  }
```

```
main#39:          OS_Start();           /* Start multitasking          */
> s
Task0#17:             OS_Delay (10);
> C
Task1#23:             OS_Delay (50);
```

```
CSP   =0000
IP    =03B2
R0    =0256
R1    =0000
R2    =0000
R3    =0000
R4    =0000
R5    =0000
R6    =0000
R7    =0000
R8    =0000
R9    =0000
R10   =0000
R11   =0000
R12   =0000
R13   =0000
R14   =0000
R15   =0000
DPP0  =0100
DPP1  =0101
DPP2  =0102
DPP3  =0003
PSW   =0808
I----Z---
ILVL=00
TFR   =0000
SP    =FC00
CP    =F620
MDH   =0000
MDL   =0000
STKUN=FC00
```

Stopped at breakpoint: *1 CODE "c:\work\embos_c166_tasking\start\src\main.c"#23 (0x3b2) SOFTWARE

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).


CrossView Pro C166/ST10 - 7s_sp_tqm167u_flash.abs
File Target Edit Run Breakpoints Data Tools Settings View Window Help
Source : c:\work\embos_c166_tasking\start\src\rtosinit.c
85
0x216
OS_Idle
Source lines
Source line step
Register
Command: CrossView
Execute
Halt

```
 The idle loop does not run as task, soit does not have have a stack of
 therefore no functionality should be implemented that relies on the sta
 to be preserved. However, a simple program loop can be programmed
 (like toggeling an output or incrementing a counter)
*/

void OS_Idle(void) {      /* Idle loop: No task is ready to exec  */
80.93%   for (;;);                /* Nothing to do ... wait for interrupt */
0.000% }

/***********************************************************************
*
*       Run-time error reaction  (OS_Error)
*
```

```
CSP  =0000
IP   =0216
R0   =F6D6
R1   =0000
R2   =0100
R3   =0000
R4   =0000
R5   =0000
R6   =0000
R7   =0000
R8   =0000
R9   =0000
R10  =0000
R11  =0000
R12  =0000
R13  =0000
R14  =001E
R15  =0000
DPP0 =0100
DPP1 =0101
DPP2 =0102
DPP3 =0003
PSW  =0808
I----Z---
ILVL=00
TFR  =0000
SP   =FBF8
CP   =F620
MDH  =0000
MDL  =0000
STKUN=FC00
```

```
> (OS_Idle) ei
c:\work\embos_c166_tasking\start\src\rtosinit.c:OS_Idle#85:     JMPR    cc_UC,OS_I
> break "c:\work\embos_c166_tasking\start\src\rtosinit.c"#85
> C
OS_Idle#85:          for (;;);             /* Nothing to do ... wait for interrup
```

```
break "c:\work\embos_c166_tasking\start\src\rtosinit.c"#85
C
```

Halted | Stopped at breakpoint: * 3 CODE "c:\work\embos_c166_tasking\start\src\rtosinit.c"#85 (0x216) SOFTWARE

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

3. C166/C167 specifics

3.1. Supported controllers

embOS does support all C166 and C167 compatible controllers. C167 libraries have been built with EXTEND and -xmifp control directives and take advantage of the enhanced instruction set.

3.2. Memory models

There are ***embOS*** libraries, which support SMALL, MEDIUM and LARGE memory model of the *TASKING Tools.* If you need support for TINY memory model, please contact us.

3.3. Available libraries

Core	Model	Library type	Library
C166	SMALL	Release	RTOS6S_R.lib
C166	SMALL	Stack-check	RTOS6S_S.lib
C166	SMALL	Stack-check + Profiling	RTOS6S_SP.lib
C166	SMALL	Debug	RTOS6S_D.lib
C166	SMALL	Debug + Profiling	RTOS6S_DP.lib
C166	SMALL	Debug + Trace	RTOS6S_DT.lib
C166	MEDIUM	Release	RTOS6M_R.lib
C166	MEDIUM	Stack-check	RTOS6M_S.lib
C166	MEDIUM	Stack-check + Profiling	RTOS6M_SP.lib
C166	MEDIUM	Debug	RTOS6M_D.lib
C166	MEDIUM	Debug + Profiling	RTOS6M_DP.lib
C166	MEDIUM	Debug + Trace	RTOS6M_DT.lib
C166	LARGE	Release	RTOS6L_R.lib
C166	LARGE	Stack-check	RTOS6L_S.lib
C166	LARGE	Stack-check + Profiling	RTOS6L_SP.lib
C166	LARGE	Debug	RTOS6L_D.lib
C166	LARGE	Debug + Profiling	RTOS6L_DP.lib
C166	LARGE	Debug + Trace	RTOS6L_DT.lib
C167	SMALL	Release	RTOS7S_R.lib
C167	SMALL	Stack-check	RTOS7S_S.lib
C167	SMALL	Stack-check + Profiling	RTOS7S_SP.lib
C167	SMALL	Debug	RTOS7S_D.lib
C167	SMALL	Debug + Profiling	RTOS7S_DP.lib
C167	SMALL	Debug + Trace	RTOS7S_DT.lib
C167	MEDIUM	Release	RTOS7M_R.lib
C167	MEDIUM	Stack-check	RTOS7M_S.lib
C167	MEDIUM	Stack-check + Profiling	RTOS7M_SP.lib
C167	MEDIUM	Debug	RTOS7M_D.lib
C167	MEDIUM	Debug + Profiling	RTOS7M_DP.lib
C167	MEDIUM	Debug + Trace	RTOS7M_DT.lib
C167	LARGE	Release	RTOS7L_R.lib
C167	LARGE	Stack-check	RTOS7L_S.lib
C167	LARGE	Stack-check + Profiling	RTOS7L_SP.lib
C167	LARGE	Debug	RTOS7L_D.lib

C167	LARGE	Debug + Profiling	RTOS7L_DP.lib
C167	LARGE	Debug + Trace	RTOS7L_DT.lib

3.4. Register banks

embOS does use the default register bank for all tasks, which is active when entering the main function. Interrupt functions can use any register bank. If an interrupt causes a task switch, all registers of the default register bank are pushed on the current user stack. So each task does have its own set of registers.

4. Stacks

4.1. Stack specifics of C166/C167

Using the *TASKING Tools*, C166/C167 does have two kind of stacks. One is the *system stack*, which is the real CPU stack. This stack has to be located inside the internal RAM and is used to save registers or for calling subroutines. The other stack is the *user stack*, which is used for automatic variables and parameter passing.
During a task change, the current *system stack* is copied to the *user stack*. When ***embOS*** reactivates a task, the *system stack* is restored. So each task does have its own *system* and *user stack*.
When using LARGE or MEDIUM memory model, the user stack has to be located in *xnear memory* area, because the *TASKING* compiler generates code, which accesses the *xnear memory* area for automatic variables and parameter passing.

4.2. Task stack for C166/C167

For C166/C167 the task stack is used as *user stack* for that specific task. So it is used for automatic variables and parameter passing by this task. When a task change takes place, contents of the *system stack* are also copied to the task stack. Therefore the minimum task stack size for C166/C167 is about 78 bytes. Please be aware, that when using LARGE or MEDIUM memory model, all task stacks have to be located in the *xnear memory* area; see also 4.1. Using the ***embOS*** storage modifier OS_STACKPTR when declaring a stack is the easiest way to achieve this.

4.3. System stack for C166/C167

For ***embOS***, *system stack* does usually mean the stack area, which is used by the application, before OS_Start() is called. ***embOS*** does use this stack area also for its software-timers and during task change.
Using the C166/C167 with *TASKING* compiler, *system stack* does also specify the memory area of the real CPU stack. The ***embOS*** *system stack* does mean the *user* and *system stack*, which are active before OS_Start() is called.
Size of *TASKING user* and *system stack* can be changed in the settings of a project; for details for details, please refer to *TASKING Tools* documentation. For ***embOS*** the minimum *system stack* size is about 88 bytes and the minimum *user stack size* is about 20 bytes (used for parameter passing).

4.4. Interrupt stack for C166/C167

C166/C167 does not support a separate stack for interrupt service routines. Interrupt functions use the *system stack* for saving CPU registers and the current *user stack* for automatic variables and parameter passing.

5. Interrupts

5.1. What happens when an interrupt occurs?

- The CPU-core receives an interrupt request.
- As soon as interrupts are enabled and the interrupt's level is higher than current CPU interrupt level, the interrupt is executed.
- CPU registers PSW and PC are pushed to the system stack by hardware
- The interrupt function does save contents of CP on the system stack.
- The interrupt function does activate its corresponding register bank.
- If you are using ***embOS*** function `OS_EnterInterrupt()`, interrupts will be disabled during execution of the interrupt function to avoid nesting of interrupts.
- If you are using ***embOS*** function `OS_EnterNestableInterrupt()`, interrupts will remain enabled during execution of the interrupt function and allow interrupts with higher interrupt level to take place.
- Execution of the interrupt function's code
- If you are using ***embOS*** function `OS_LeaveInterrupt()` or `OS_LeaveNestableInterrupt()` at the end of the interrupt function, ***embOS*** will check for pending task switches and change the task is required during the end of the interrupt function.
- The interrupt function does reactivate the register bank, which was active before executing the interrupt function.
- Interrupt function does end with a RETI instruction, which restores PC and PSW.

5.2. Defining interrupt handlers in "C"

TASKING compiler does have a specific syntax for defining interrupt functions. If you want to use the ***embOS*** API inside your interrupt function, you have to tell the OS, that you are executing an interrupt function. You do this by calling `OS_EnterInterrupt()` or `OS_EnterNestableInterrupt()` at the beginning of your interrupt function and by calling `OS_LeaveInterrupt()` or `OS_LeaveNestableInterrupt()` at its end.

Example

"Simple" interrupt-routine

```
_interrupt (0x2b) void OS_ISR_rx(void) {
  volatile unsigned char x;
  OS_EnterNestableInterrupt();  /*  We will enable interrupts */
  x = S0RBUF;
  if (S0CON & 0x0600) {
    S0CON  =0x80D1;
  }
  else {
    OS_OnRx(x);
  }
  OS_LeaveNestableInterrupt();
}
```

5.3. Interrupt-stack

C166/C167 interrupt functions use the current user stack for automatic variables and parameter passing. For register saving, the system stack is used. The routines `OS_EnterIntStack()` and `OS_LeaveIntStack()` are supplied for source compatibility to other processors only and have no functionality.

5.4. Special considerations for C166/C167

embOS has been designed to use its API even within interrupt functions and to allow nesting of interrupts. Further more, a task switch caused by an interrupt function is executed immediately when returning from interrupt instead of waiting for next timer tick or OS call to take place.
Because C166/C167 does not disable further interrupts after accepting one and also disabling interrupts does not work immediately, there are some restrictions for interrupt functions using ***embOS*** API, because ***embOS*** has to keep track on nesting interrupts using its API.
All interrupt functions, which use the ***embOS*** API must have same interrupt level. Any interrupt function not using the ***embOS*** API must not be interrupted by an interrupt function, which does use the ***embOS*** API. Usually this can be achieved by giving interrupt function not using ***embOS*** API a higher interrupt level.

6. STOP / WAIT Mode

In case your controller does support some kind of power saving mode, it should be possible to use it also with ***embOS***, as long as the timer keeps working and timer interrupts are processed. To enter that mode, you usually have to implement some special sequence in function OS_Idle(), which you can find in ***embOS*** module RTOSINIT.c.

7. Technical data

7.1. 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***. Using SMALL memory model, the minimum ROM requirement for the kernel itself is about 1.400 bytes.
In the table below, you can find minimum RAM size for ***embOS*** resources. Please note, that sizes depend on selected ***embOS*** library mode; table below is for a release build.

embOS resource	RAM [bytes]
Task control block	18
Resource semaphore	4
Counting semaphore	2
Mailbox	12
Software timer	10

8. Files shipped with ***embOS***

Directory	**File**	**Explanation**
INC	RTOS.H	Include file for RTOS, to be included in every "C"-file using RTOS-functions
LIB	`RTOS*.lib`	Libraries for all memory models and debug options
SRC	RtosInit.c	Initializes the hardware, can be modified if required
SRC	Main.c	Frame program to serve as a start

Any additional file shipped as example.

9. Index