embOS

Real Time Operating System

CPU & Compiler specifics for

C166/C167 core with Keil uVision2

Developer's Kit V2.38a

Document Rev. 1

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 *Keil Developer's Kit*.

1.1. How to use this manual

This manual describes all CPU and compiler specifics for embOS using C166/C167 based controllers with *Keil Developer's Kit*. 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 *Keil Developer's Kit*. 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 *Keil Developer's Kit*.

2. Using ***embOS*** with Keil Developer's Kit

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 *Keil Developer's Kit* 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 *Keil Developer's Kit* project manager, it may be used without the project manager using batch files or a make utility without any problem.

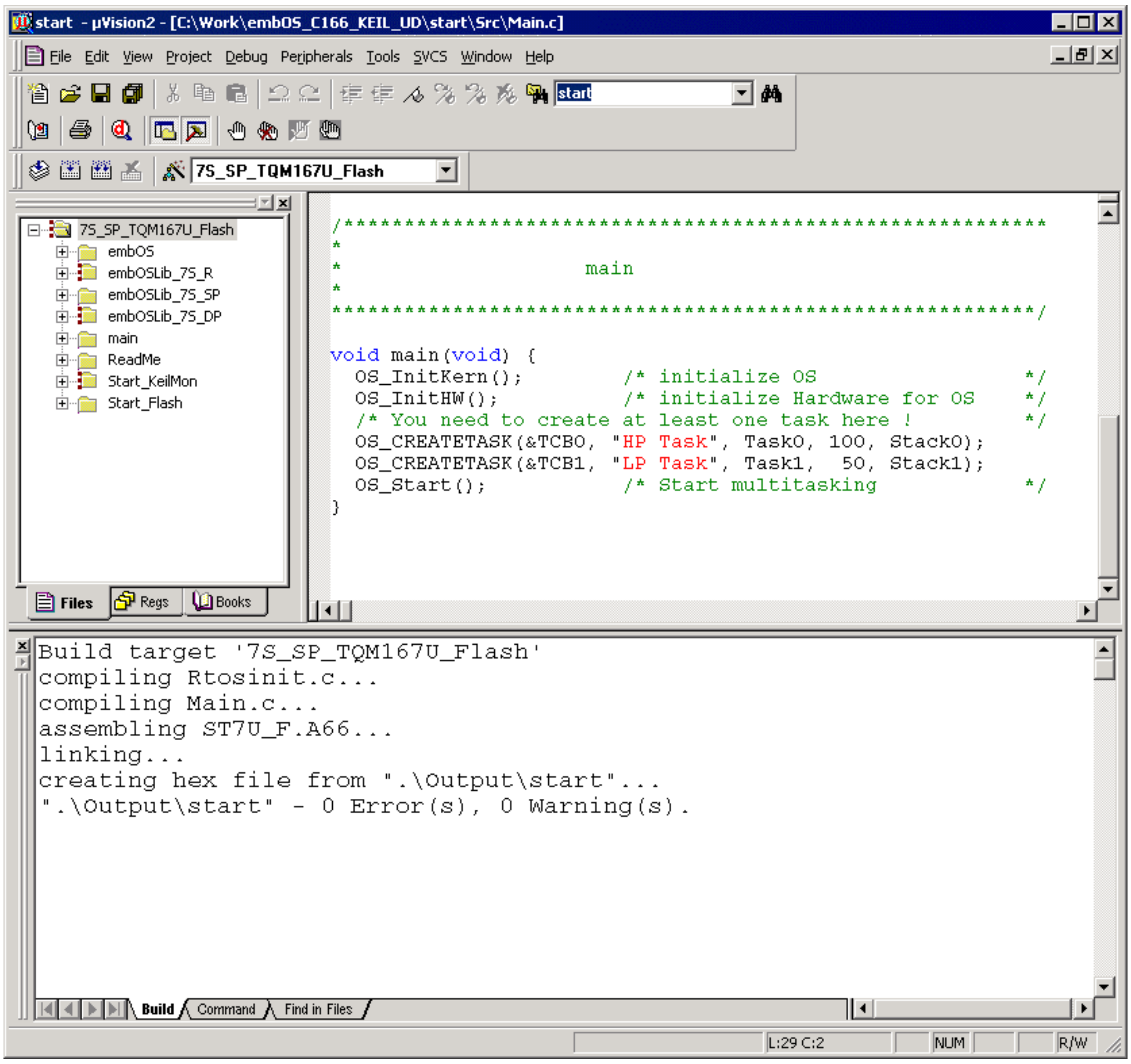
2.2. First steps

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.uv2 with *Keil Developer's Kit* project manager (e.g. by double clicking it)
Build the start project

Your screen should look like follows:



```
/*********************************************************************
*
*                    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                     */
}
```

```
Build target '7S_SP_TQM167U_Flash'
compiling Rtosinit.c...
compiling Main.c...
assembling ST7U_F.A66...
linking...
creating hex file from ".\Output\start"...
".\Output\start" - 0 Error(s), 0 Warning(s).
```

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

2.3. The sample application Main.c

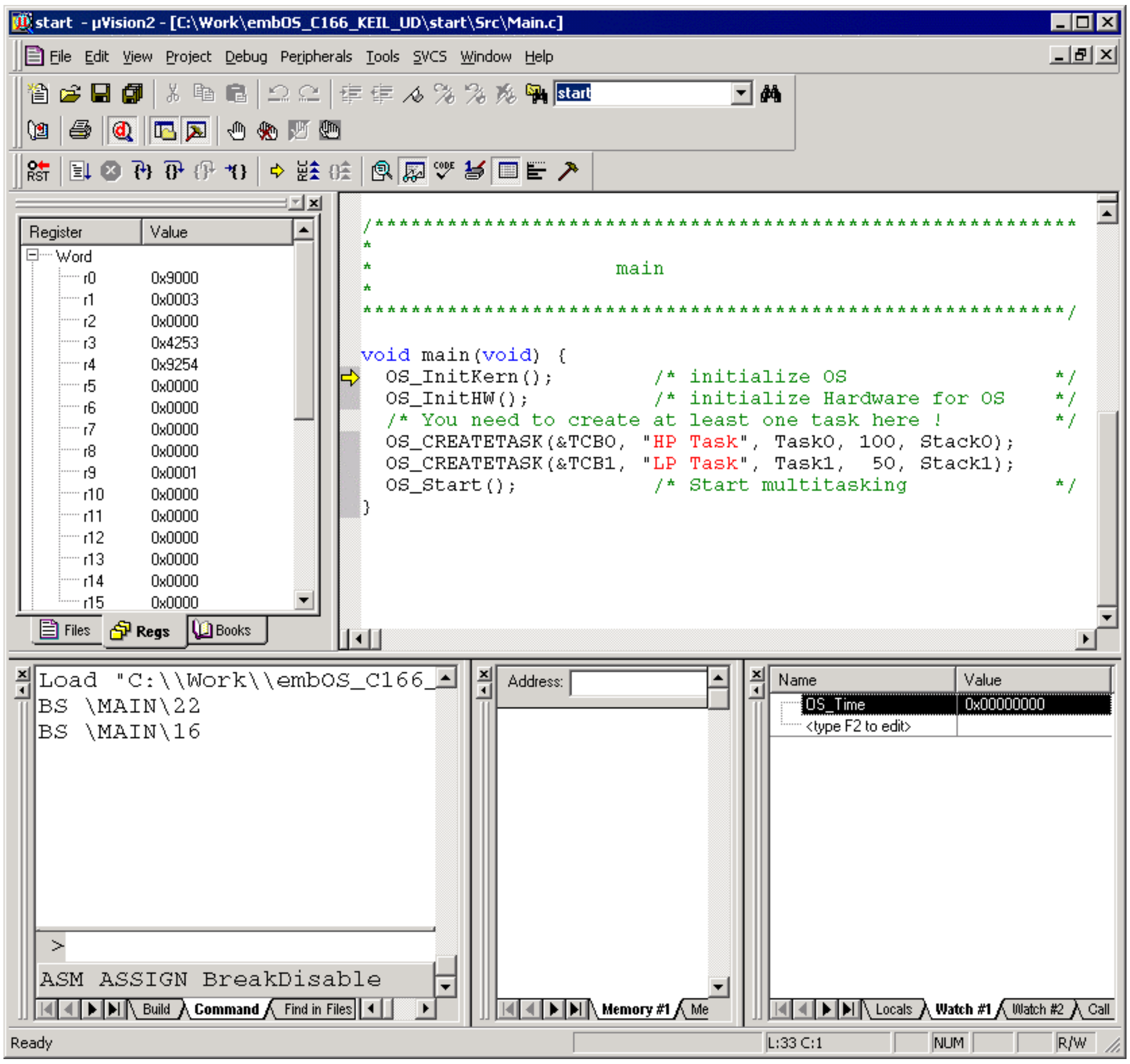
The following is a printout of the sample application main.c. It is a good starting-point for your application. (Please note that the file actually shipped with your port of ***embOS*** may look slightly different from this one)
What happens is easy to see:
After initialization of ***embOS;*** two tasks 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 Keil 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.


start - µVision2 - [C:\Work\embOS_C166_KEIL_UD\start\Src\Main.c]
File Edit View Project Debug Peripherals Tools SVCS Window Help
start
Files Regs Books
Build Command Find in Files
Address:
Memory #1 Me
Locals Watch #1 Watch #2 Call
Ready L:33 C:1 NUM R/W

Register	Value
Word	
r0	0x9000
r1	0x0003
r2	0x0000
r3	0x4253
r4	0x9254
r5	0x0000
r6	0x0000
r7	0x0000
r8	0x0000
r9	0x0001
r10	0x0000
r11	0x0000
r12	0x0000
r13	0x0000
r14	0x0000
r15	0x0000

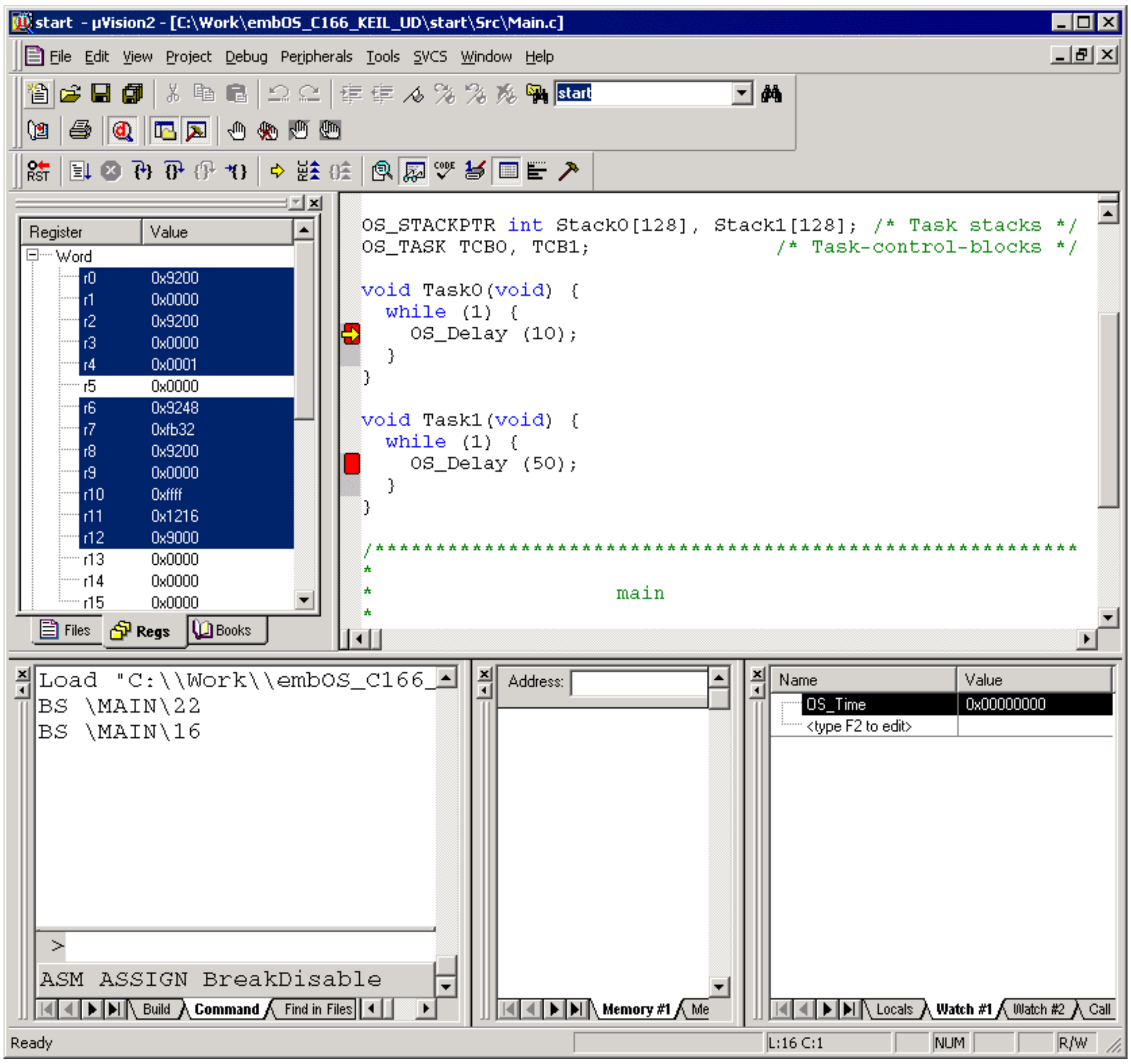
```
/*********************************************************************
*
*                    main
*
*********************************************************************/

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            */
}
```

```
Load "C:\\Work\\embOS_C166_
BS \MAIN\22
BS \MAIN\16
```

```
>
ASM ASSIGN BreakDisable
```

Name	Value
OS_Time	0x00000000
<type F2 to edit>	

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.


start - µVision2 - [C:\Work\embOS_C166_KEIL_UD\start\Src\Main.c]
File Edit View Project Debug Peripherals Tools SVCS Window Help
start
Files Regs Books
Address:
Build Command Find in Files
Memory #1 Me
Locals Watch #1 Watch #2 Call
Ready
L:16 C:1
NUM
R/W

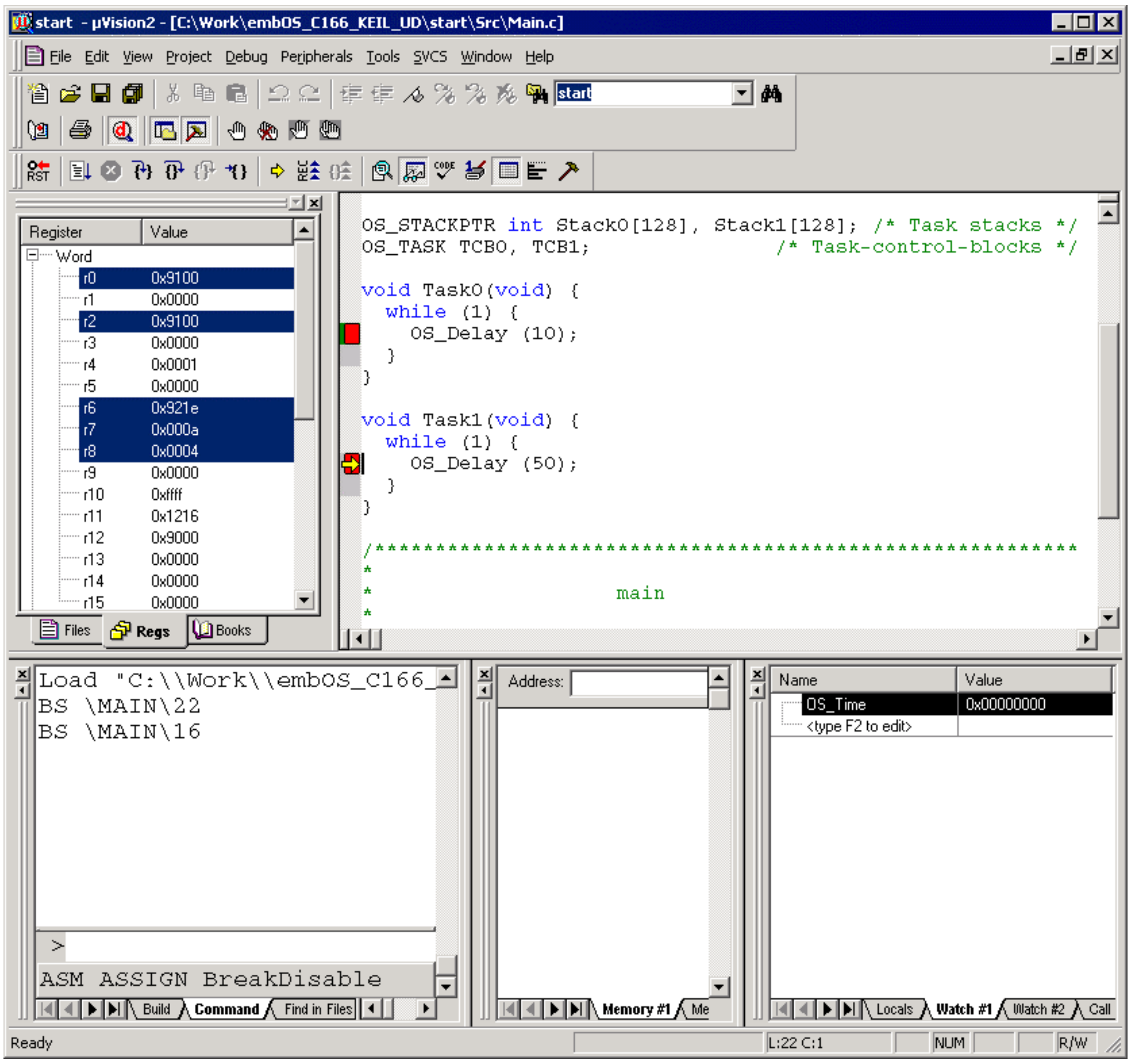
Register	Value
Word	
r0	0x9200
r1	0x0000
r2	0x9200
r3	0x0000
r4	0x0001
r5	0x0000
r6	0x9248
r7	0xfb32
r8	0x9200
r9	0x0000
r10	0xffff
r11	0x1216
r12	0x9000
r13	0x0000
r14	0x0000
r15	0x0000

```
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
*
```

```
Load "C:\\Work\\embOS_C166_
BS \MAIN\22
BS \MAIN\16

>
ASM ASSIGN BreakDisable
```

Name	Value
OS_Time	0x00000000
<type F2 to edit>	

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


start - µVision2 - [C:\Work\embOS_C166_KEIL_UD\start\Src\Main.c]
File Edit View Project Debug Peripherals Tools SVCS Window Help
start
Files Regs Books
Build Command Find in Files
Address:
Memory #1 Me
Locals Watch #1 Watch #2 Call
Ready
L:22 C:1
NUM
R/W

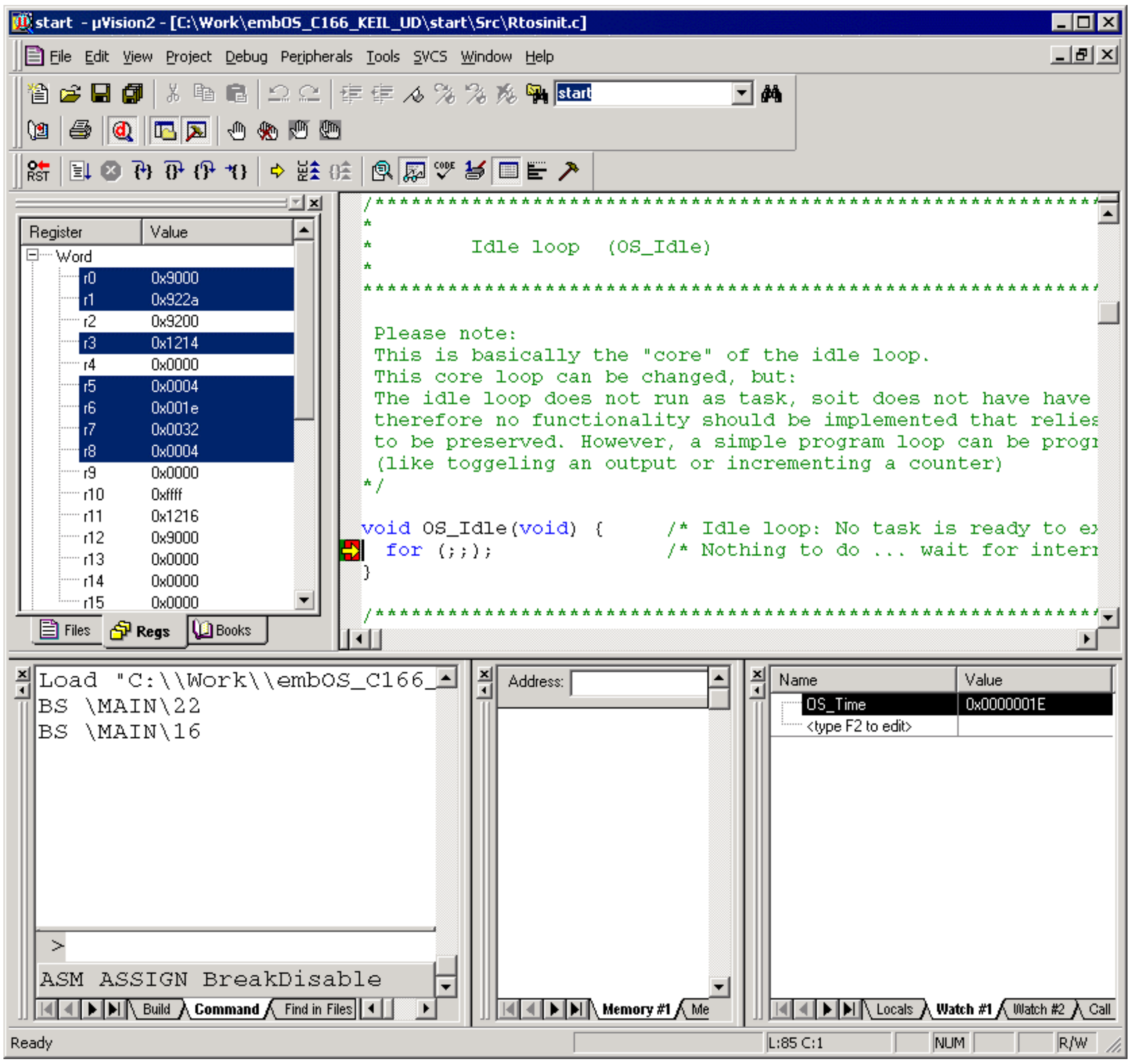
Register	Value
Word	
r0	0x9100
r1	0x0000
r2	0x9100
r3	0x0000
r4	0x0001
r5	0x0000
r6	0x921e
r7	0x000a
r8	0x0004
r9	0x0000
r10	0xffff
r11	0x1216
r12	0x9000
r13	0x0000
r14	0x0000
r15	0x0000

```
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
*
```

```
Load "C:\\Work\\embOS_C166_
BS \MAIN\22
BS \MAIN\16

>
ASM ASSIGN BreakDisable
```

Name	Value
OS_Time	0x00000000
<type F2 to edit>	

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).


start - µVision2 - [C:\Work\embOS_C166_KEIL_UD\start\Src\Rtosinit.c]
File Edit View Project Debug Peripherals Tools SVCS Window Help
start
Files Regs Books
Build Command Find in Files
Memory #1 Me
Locals Watch #1 Watch #2 Call
Ready
L:85 C:1
NUM
R/W

Register	Value
Word	
r0	0x9000
r1	0x922a
r2	0x9200
r3	0x1214
r4	0x0000
r5	0x0004
r6	0x001e
r7	0x0032
r8	0x0004
r9	0x0000
r10	0xffff
r11	0x1216
r12	0x9000
r13	0x0000
r14	0x0000
r15	0x0000

```
/***************************************************************
*
*        Idle loop  (OS_Idle)
*
****************************************************************

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

void OS_Idle(void) {      /* Idle loop: No task is ready to ex
  for (;;);               /* Nothing to do ... wait for interr
}

/***************************************************************
```

```
Load "C:\\Work\\embOS_C166_
BS \MAIN\22
BS \MAIN\16

>
ASM ASSIGN BreakDisable
```

Address:

Name	Value
OS_Time	0x0000001E
<type F2 to edit>	

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 the MOD167 control directive and take advantage of the enhanced instruction set.

3.2. Memory models

There are ***embOS*** libraries, which support SMALL, COMPACT, HCOMPACT, MEDIUM, LARGE and HLARGE memory model of the *Keil Developer's Kit.* 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	COMPACT	Release	RTOS6C_R.lib
C166	COMPACT	Stack-check	RTOS6C_S.lib
C166	COMPACT	Stack-check + Profiling	RTOS6C_SP.lib
C166	COMPACT	Debug	RTOS6C_D.lib
C166	COMPACT	Debug + Profiling	RTOS6C_DP.lib
C166	COMPACT	Debug + Trace	RTOS6C_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	COMPACT	Release	RTOS7C_R.lib
C167	COMPACT	Stack-check	RTOS7C_S.lib
C167	COMPACT	Stack-check + Profiling	RTOS7C_SP.lib
C167	COMPACT	Debug	RTOS7C_D.lib

C167	COMPACT	Debug + Profiling	RTOS7C_DP.lib
C167	COMPACT	Debug + Trace	RTOS7C_DT.lib
C167	HCOMPACT	Release	RTOS7HC_R.lib
C167	HCOMPACT	Stack-check	RTOS7HC_S.lib
C167	HCOMPACT	Stack-check + Profiling	RTOS7HC_SP.lib
C167	HCOMPACT	Debug	RTOS7HC_D.lib
C167	HCOMPACT	Debug + Profiling	RTOS7HC_DP.lib
C167	HCOMPACT	Debug + Trace	RTOS7HC_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
C167	HLARGE	Release	RTOS7HL_R.lib
C167	HLARGE	Stack-check	RTOS7HL_S.lib
C167	HLARGE	Stack-check + Profiling	RTOS7HL_SP.lib
C167	HLARGE	Debug	RTOS7HL_D.lib
C167	HLARGE	Debug + Profiling	RTOS7HL_DP.lib
C167	HLARGE	Debug + Trace	RTOS7HL_DT.lib

3.4. Assembler startup code

embOS does not require any specific startup code. The scheduler is started by calling function OS_Start(). However, ***embOS*** has to be able to reset the user stack pointer. For that reason, it is necessary to make the symbol ?C_USERSTKTOP of the default startup code public.

3.5. Register banks

The Context Pointer CP is not modified by ***embOS*** in any way. Therefore, you can still use different register banks in your application, but you must ensure, that no ISR using ***embOS*** API takes place during a different register bank is active. You must also not call any ***embOS*** function during a different register bank is active.

4. Stacks

4.1. Stack specifics of C166/C167

Using the *Keil Developer's Kit* , 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*.
By default the user stack has to be located in *near memory* area, because the *Keil* compiler generates code, which accesses the *near memory* area for automatic variables and parameter passing. The compiler option USERSTACKDPP3 is currently not supported.

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 22 bytes. Please be aware, that all task stacks have to be located in the *near memory* area; see also 4.1. This can be achieved by using a memory model with default data segment *near*, or by using memory type keyword *near* for the stack declaration.

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 *Keil* 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 *Keil user* and *system stack* can be changed in the assembler startup code; for details, please refer to *Keil Developer's Kit* documentation. For ***embOS*** the minimum *system stack* size is about 112 bytes and the minimum *user stack size* is about 4 bytes (used by OS_CREATETASK 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 scratch registers and those permanent registers, which will be modified, to the system stack.
- 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 restore scratch registers and those permanent registers, which have been modified.
- Interrupt function does end with a RETI instruction, which restores PC and PSW.

5.2. Defining interrupt handlers in "C"

Keil 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

```
void OS_ISR_rx(void) interrupt 0x2b {
  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. Fast interrupts with C166/C167

Instead of disabling interrupts when ***embOS*** does atomic operations, the interrupt level of the CPU is set to 7. Therefore all interrupts with level 8 or higher can still be processed.
These interrupts are named *fast interrupts*. You must not execute any ***embOS*** function from within a *fast interrupt* function.

5.5. 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 and which want to do task switching at the end, 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 and which wants to do task switching at its end. Usually this can be achieved by giving interrupt function not using ***embOS*** API a higher interrupt level.

The description above might be difficult to understand. Therefore, here are useful recommendations:

- All ***embOS*** interrupts, which end with `OS_LeaveInterrupt()` or `OS_LeaveNestableInterrupt()`, should have interrupt level 1.
- Fast interrupts (interrupt level >=8) must not call any ***embOS*** function.
- Any ***embOS*** interrupt with an interrupt level greater than 1 should end with `OS_LeaveInterruptNoSwitch()` or `OS_LeaveNestableInterruptNoSwitch()`.
- Any interrupt function not using the ***embOS*** API should have a higher interrupt level than those interrupt functions using the ***embOS*** API.

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	ST7U_F.A66	Low level assembler startup code for the TQ STK16XU board and an application running in FLASH w/o the Keil Monitor
SRC	ST7U_R.A66	Low level assembler startup code for the TQ STK16XU board and an application running in RAM under control of the Keil Monitor
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