embOS

Real Time Operating System

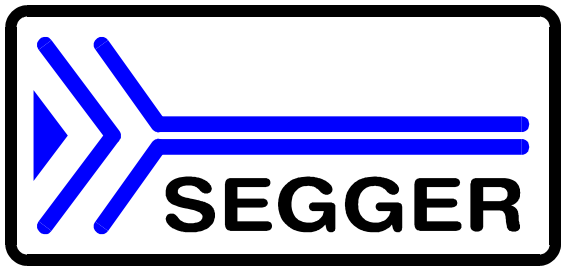
CPU & Compiler specifics for

ST7 CPUs

using Cosmic compiler

and Cosmic IdeaST7

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 STMicroelectronics ST7 series of microcontrollers using Cosmic compiler and IdeaST7 embedded workbench.

1.1. How to use this manual

This manual describes all CPU and compiler specifics for ***embOS*** using STMicroelectronics ST7 based controllers with Cosmic IdeaST7 embedded workbench. 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 Cosmic IdeaST7 embedded workbench. 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 STMicroelectronics ST7 based controllers using Cosmic ST7 compiler.

2. Using *embOS* with Cosmic IdeaST7 workbench

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 Cosmic IdeaST7 workbench to develop your application, no further installation steps are required. You will find prepared sample start projects, 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 Cosmic IdeaST7 workbench for your application development in order to become familiar with ***embOS.***

If for some reason you do not want to work with Cosmic IdeaST7 workbench, you should:

- Copy either all or only the library-file you need to your work-directory.
- Copy the CPU specific source file RtosInit.c to your work-directory.
- Also copy the ***embOS*** header-file RTOS.h 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 Cosmic IdeaST7 workbench, it may be used with batch files or a make utilities without any problem.

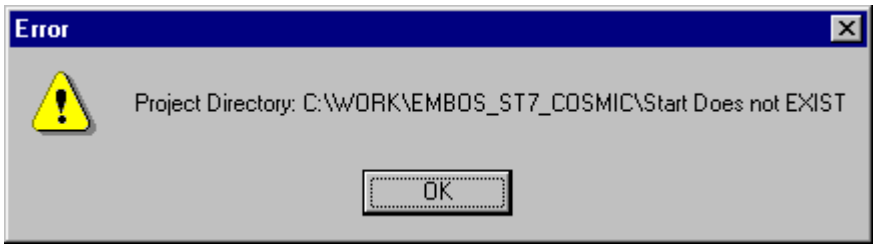
2.2. First steps

After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received ready to go sample start projects and it is a good idea to use one of these as a starting point of all of your applications, as these projects contain all compiler settings needed for ***embOS***.

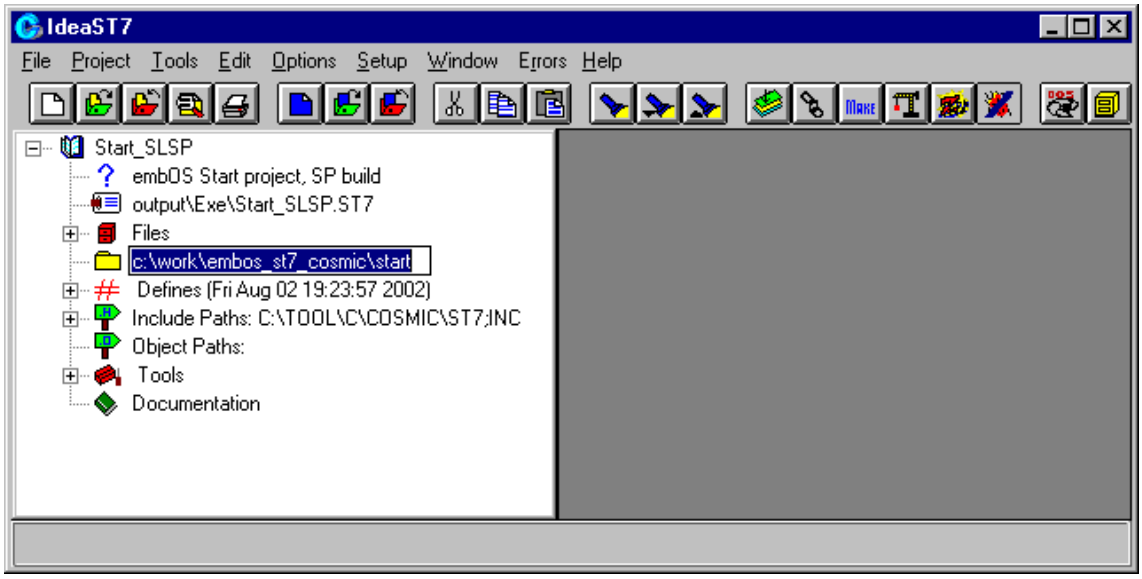
To get your new application running, you should proceed as follows:

- 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 and its sub folders.
- Start Cosmic IdeaST7
- Load one sample project from the start sub-directory in your working folder, for example ‘work\start\Start_SLSP.prj’.

You may get the following error message from IdeaST7:



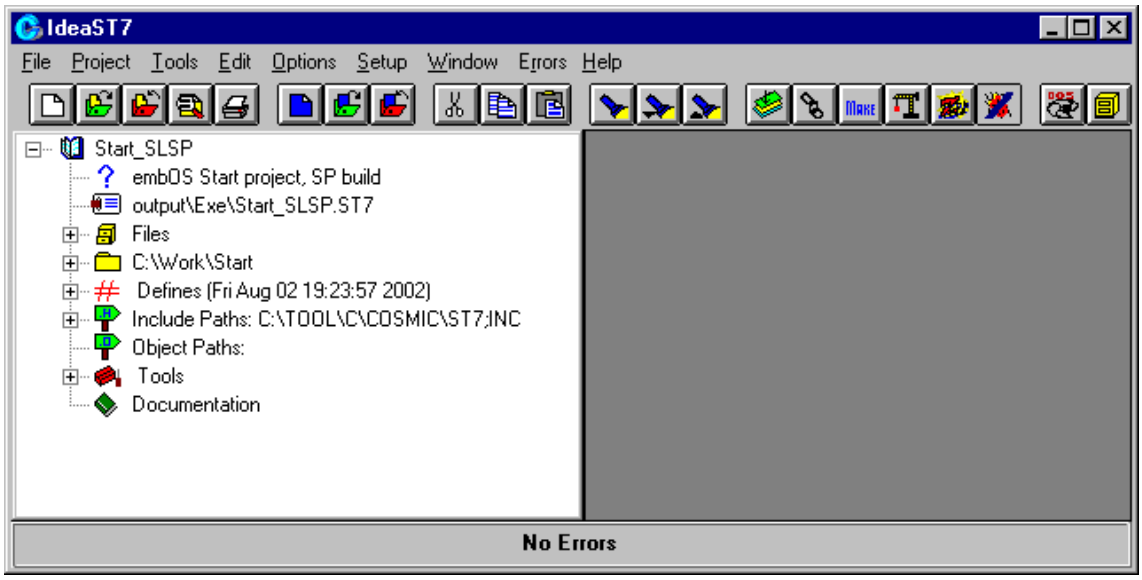
This occurs, because IdeaST7 projects contains an absolute path references to the work directory.
Press OK to close the message dialog and set the project directory by selecting the project path as shown below:



Change the working directory name to the start folders name you are actually using. (‘c:\work\start\’, if you followed our example above.)
You may alternatively set the working folder by menu ‘Setup | Working Directory’

Now Build the start project

Your screen should look like follows:



All outputs are placed in the 'output\' sub directory of the start project folder. This folder contains three subfolders for executables, list files and object files. Three executables were built:

- output\exe\Start_SLSP.mot as Motorola S-record file may be used for the target CPU.
- output\Exe\Start_SLSP.ST7 is Cosmics debug file and may be used for Cosmic's ZAP debugger / simulator.
- output\Exe\Start_SLSP.695 is an additional debug file in IEEE695 format.

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[64], Stack1[64]; /* 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_Start();            /* Start multitasking         */
}
```

2.4. Stepping through the sample application using ZAP simulator

The debug output files produced by IdeaSt7 may be used with different incircuit emulators, debuggers or simulators.
Described here is the usage of Cosmic's debugger ZAP used as simulator.

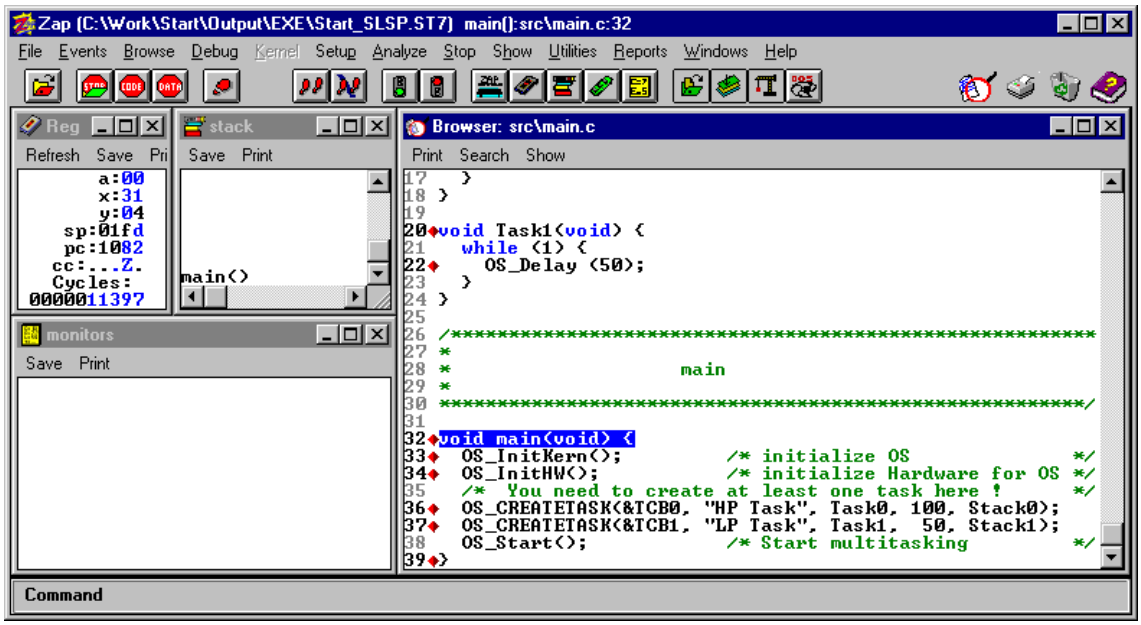
Before starting ZAP, the interrupt simulation of ***embOS*** timer interrupt should be prepared:

- Copy the interrupt definition file 'ST7.INT' found in 'Start\ZAP\' into the your program folder that contains the 'ZAPST7.exe' executable.

After starting ZAP, proceed as follows:

- Select menu function 'File | Load' to load the sample start project output file 'Start_SLSP.ST7' which is found in the 'output\exe\' subdirectory of your start project folder.
- Then set or check the search path for source files by menu 'Setup | Path' and set the source path to the source directory of your start project (for example 'c:\work\start\). Please set this path to the folder which contains the start project, not the source files.
- Open the source window by menu 'Browse | Function' and select 'main'
- Start debugging by menu 'Debug | Go Till' and choose 'main'

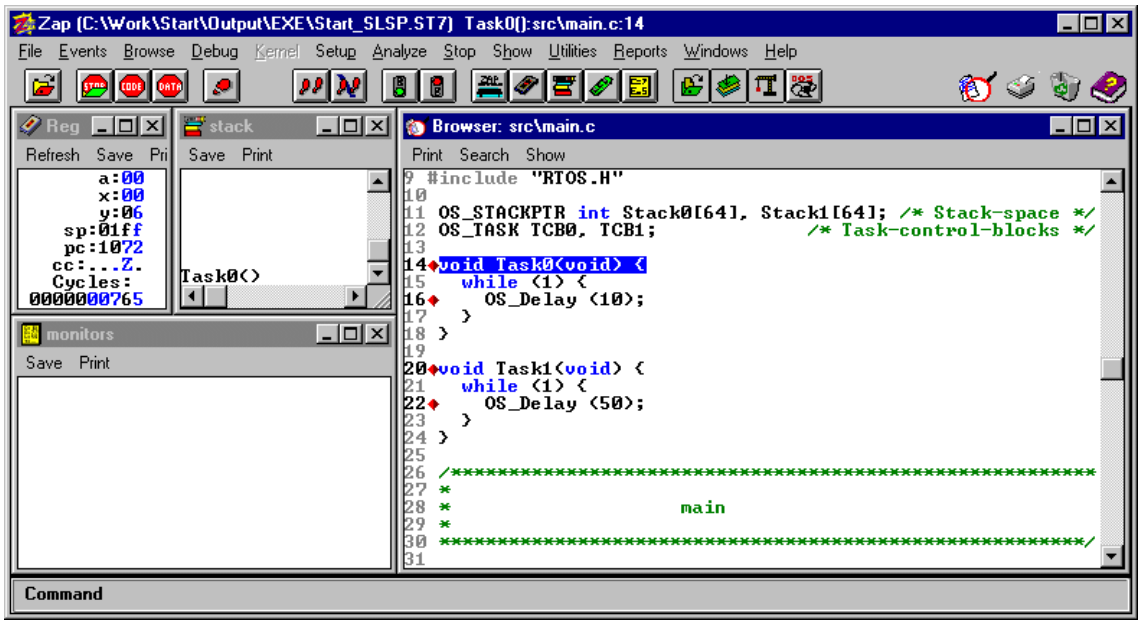
Your screen should look as follows:



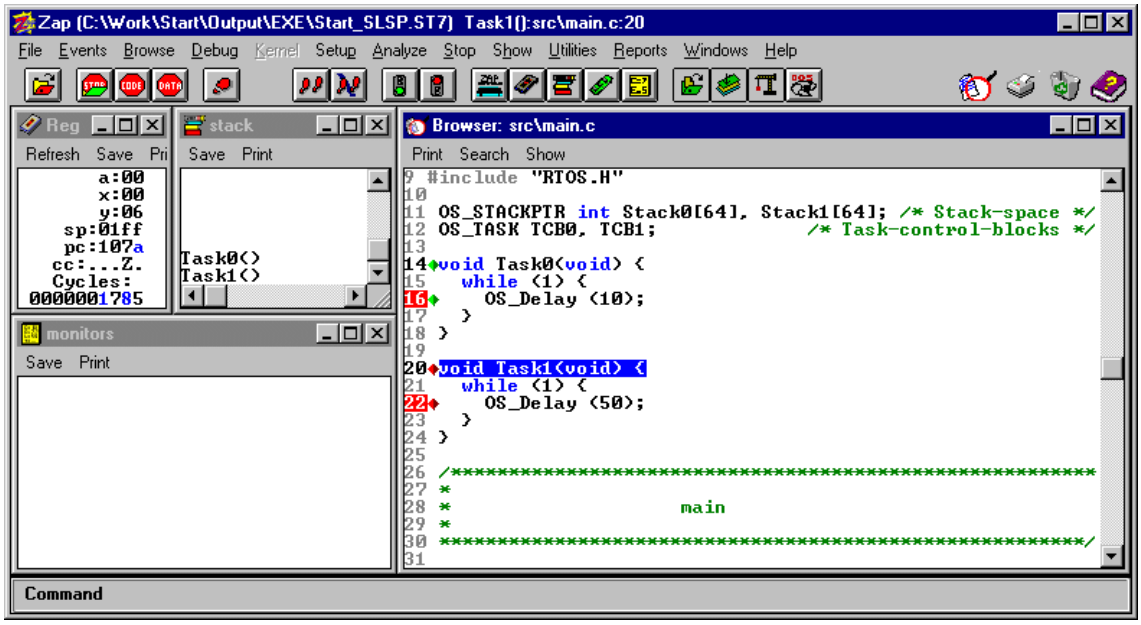
The simulator executed the startup code and stopped at main().
Now you can step through the program.

`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 interrupts to be enabled from start, you should call `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.*** You may step through it to see what is done.
`OS_COM_Init()`, called from `OS_InitHW()` is optional and initializes a UART for communication with embOSView.

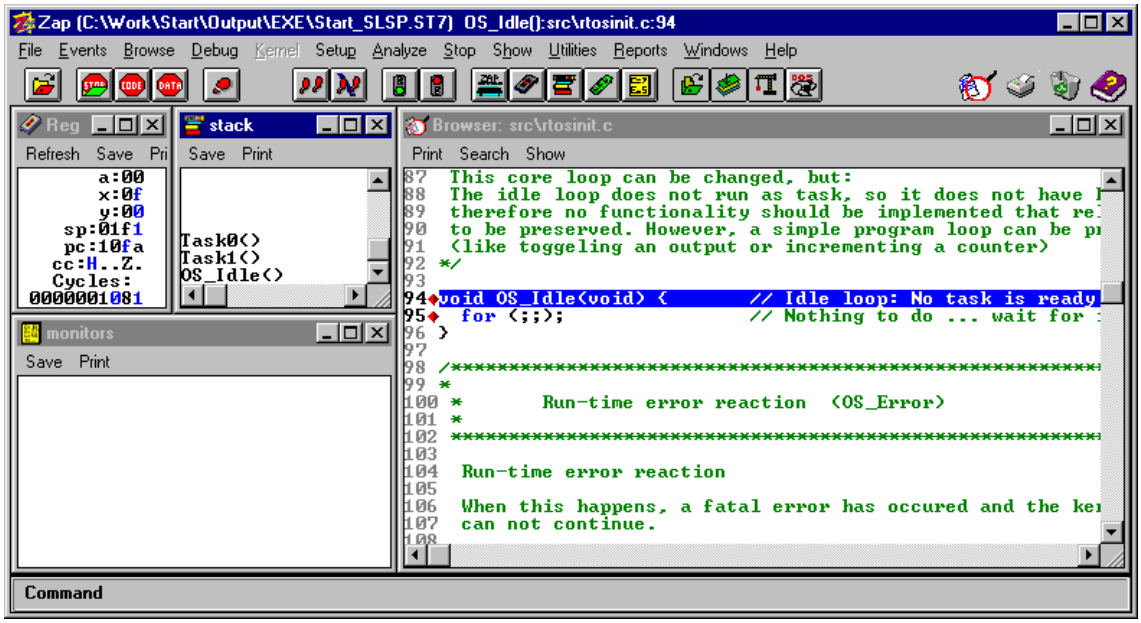
`OS_Start()` is the last line in *main*, since it starts multitasking and does not return.
When you step over `OS_Start()` using the 'Step one high level Instruction' eg menu 'Debug | Step' command twice, the next line executed is already in the highest priority task created. In our small start program, Task0() is the highest priority task and is therefore active.



If you continue stepping through the program, Task0 will run into the delay and therefore you will arrive Task1(), which is the task with the second highest priority:



Before continuing, you should set a break point in every task at the call of `OS_Delay()` as shown above.
Then continuing, there is no other task ready for execution. ***embOS*** will start the idle-loop which is part of RtosInit.c:

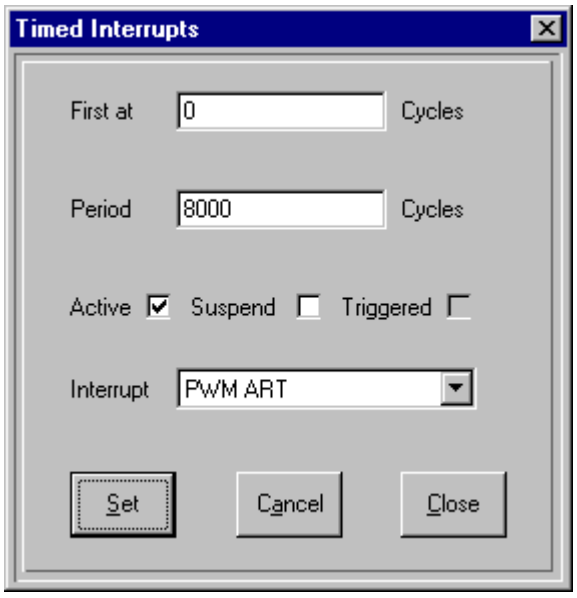


`OS_Idle()` is an endless loop which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

Now you should enable the interrupt simulation of ZAP to simulate ***embOS*** timer interrupt. As both of the tasks are delayed for a given time, ***embOS*** timer interrupt is needed to wake them up.

To start the interrupt simulation, choose menu ‘Setup | Interrupts | Time Wise’ to open the setup dialog for simulated periodical interrupts.

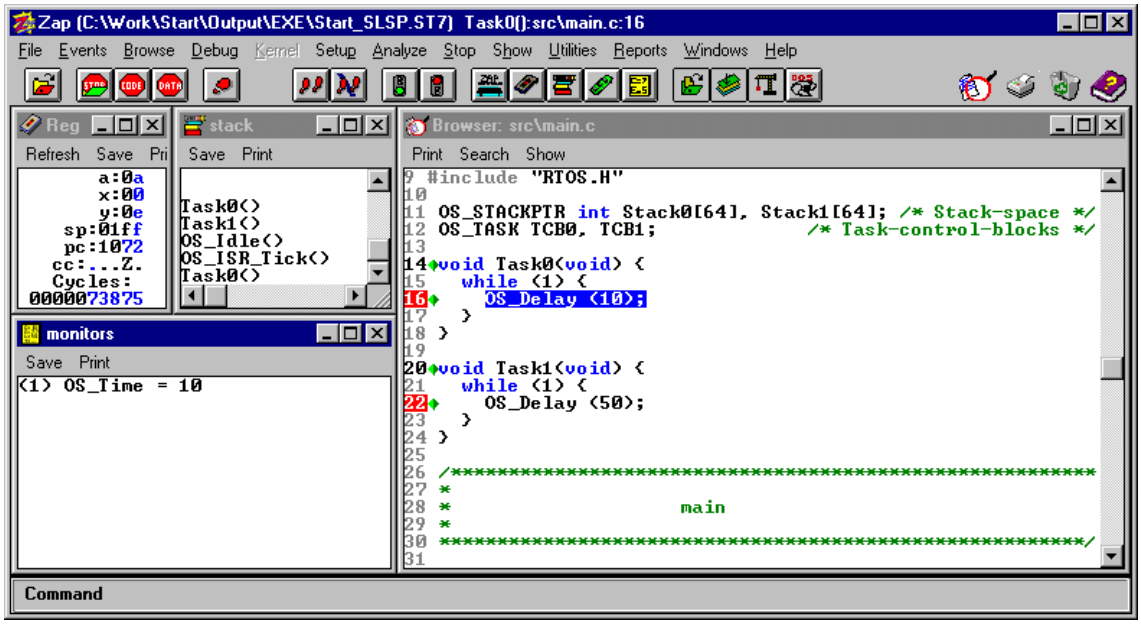
- Select ‘PWM ART’ which is the timer used by standard ***embOS*** distribution from the drop down list.
- Set a period of 8000 cycles, which would result in 1ms timer interval in real hardware.
- Check the ‘Active’ checkbox.



Finally press the ‘Set’ button and then close the dialog.

To watch the system time running, you may also monitor the ***embOS*** system time variable `OS_Time`.

Choose menu function ‘Browse | Variables | in Global List’ to open the list of all variables. Then doubleclick on `OS_Time` and choose ‘Monitor...’ from the context menu.
If you set a breakpoint in one or both of our tasks, you should now execute the “Go” command and can see that the tasks continuing execution after the given delay.
The system variable `OS_Time` reflects how much time has expired in the target system:



3. ST7 CPU specifics

3.1. Memory models

embOS as multitasking real time operating systems needs a memory model and compiler that supports reentrant code. Therefore the only memory model supported is the 'Stack Long' memory model that Cosmic's C-compiler offers.

3.2. Available libraries

embOS comes with different libraries. During development of your application you may use any of the debug type libraries, as those include error checks that are helpful during development.
Later you may change to the stack check version, which executes faster, as all other runtime error checks are disabled.
The release library executes fastest as it does not include any error checks during runtime. This one should be used finally for your target after debugging your application.

The library files to use are:

Memorymodel	Library type	Library	define
Stack long	Release	RtosSLR.lib	OS_LIBMODE_R
Stack long	Stack-check	RtosSLS.lib	OS_LIBMODE_S
Stack long	Stack-check + Profiling	RtosSLSP.lib	OS_LIBMODE_SP
Stack long	Debug	RtosSLD.lib	OS_LIBMODE_D
Stack long	Debug + Profiling	RtosSLDP.lib	OS_LIBMODE_DP
Stack long	Debug + Profiling + Trace	RtosSLDT.lib	OS_LIBMODE_DT

They are located in the LIB\ sub folder of your start project folder.

When using Cosmic IdeaST7 workbench or compiler, please check the following points:

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

3.3. Distributed project files

The distribution of ***embOS*** contains start projects for the Stack long memory model in the start subdirectory.
The project names reflect the memory model and target library used.
You should use these start projects to develop your application. Simply add new subdirectories containing your own application modules to the 'start' directory and then add your files to the project. This ensures, that all settings needed for ***embOS*** are always setup correctly.

4. Stacks

4.1. Stack address range

The ST7 has an 8-bit hardware stack-pointer and a stack memory that is located at a fixed address above zero page.
This stack is too small to be distributed among several tasks.
Therefore task stacks may be defined in any memory address range that can be used as RAM. During task switches, ***embOS*** copies the used task stack into the hardware stack memory area.
The zero page area of ST7 controllers is almost used for special function registers. Also the ***embOS*** internal variables are placed there.
You should chose the memory area above 0x200 as task stacks.

4.2. System stack

The system stack is the one that is used after reset. For ST7 controller this is the physical stack memory, starting at address 0x100. It is used for the following purposes:

- Normal stack during startup (until `OS_Start()` is called).
- ***embOS*** internal functions
- Software timer
- Stack for interrupt handler, when `OS_EnterIntStack()` is used.

Normally, the whole memory area that the stack pointer can address, should be used as system stack. (256 bytes)
When additional RAM is used for your application and you examined, that the whole stack area is not needed, you may reduce the system stack size.

NOTE: Current ***embOS*** version is designed for those CPUs that support 256 bytes of stack, even if this amount of stack is not used. This is because internal calculations only work with the upper stack memory address at 0x1FF.
If you have to use a small ST7 CPU with a maximum stack address of 0x17F, please contact us.

4.3. Task stacks

Every task uses its own stack which has to be defined when the task is created. This stack may be located in any memory area.
A good value for a minimum task stack size is about 40 bytes.

NOTE: The maximum task stack size is limited to 254 bytes.
This is because task stacks are copied onto the real hardware stack, which is limited to 256 bytes.

5. Interrupts

5.1. What happens when an interrupt occurs?

- The CPU-core receives an interrupt request
- As soon as the interrupts are enabled and the processor finished a complete instruction, the interrupt is executed
- the CPU saves actual PC, X, A and CC register on the stack
- the CPU loads the specified program counter address from the interrupt vector table thus starting the interrupt service routine.
- The CPUs interrupt priority is set to the value according to the accepted interrupt (defined in the according interrupt priority level register). Thus lower priority level interrupts are automatically disabled.
- ISR: save registers
- ISR: user-defined functionality
- ISR: restore registers
- ISR: Execute RETI command, restoring CC, A, X and PC, thus returning to the interrupted program.
- For details, please refer to the ST7 users manual.

5.2. Defining interrupt handlers in "C"

Interrupt handlers can be defined in C source code by using special prefixes. The definition of an interrupt function using ***embOS*** calls is very much the same as for a normal interrupt service routine (ISR).
If your ISR shall execute ***embOS*** system calls, you have to tell ***embOS***, that an interrupt routine is running:

- Call `OS_EnterInterrupt()`at the beginning of your ISR
- Call `OS_LeaveInterrupt()`as last function in your ISR

This disables further interrupts and prevents ***embOS*** from executing task switches from within an ISR, as those would cause a system crash. If required, a task switch is executed from `OS_LeaveInterrupt()`.

When ***embOS*** functions are used in an interrupt service routine, nested interrupts are not allowed.

Example

Interrupt-routine using ***embOS*** calls

```
@interrupt @svlreg void ART_TimerInt (void) {
  OS_EnterInterrupt();
  ARTCSR |= (0);               // reset interrupt pending(read CSR)
  OS_SignalEvent(EVENT_ART, TCBMain);
  OS_LeaveInterrupt();
}
```

@interrupt forces the compiler to add additional code to store additional memory variables used as temporary registers onto the stack when entering the interrupt routine. These memory variables are restored when leaving the routine. The routine ends with RETI instruction instead of a normal return.
@svlreg also saves (and restores) memory variables used as additional register for long variables. When ***embOS*** functions are called from an interrupt han-

dler, this option should be used, as ***embOS*** functions may use long variable calculations.
For details please refer to Cosmic C Cross Compiler users guide for ST7 microcontroller.

Interrupt vector table

Every interrupt function has to be declared in the interrupt vector table. The vector table itself may be defined in C or assembly language.
The distribution of ***embOS*** for ST7 contains a vector table definition in C source code form in the file 'Vector.c', which should be used and modified to add additional interrupt service routines used for your application.
Also, if you have to use an other timer for the ***embOS*** timer tick, this file has to be modified.
The vector file has to be part of your project. The linker command file has to link this file to a fixed address.
For details please refer to the cross compiler manual.

5.3. Restrictions for interrupts with ***embOS*** ST7

- **Nested interrupts are not allowed**. ST7 controller does not disable interrupts when the interrupt service routine is entered. Instead of this, the interrupt priority is set to the value defined for the current peripheral interrupt. Unfortunately there is no way to enable interrupts without lowering the priority. Enabling interrupts sets the priority to lowest level.
- **All Interrupts have to run on highest priority**. This ensures, that nesting of interrupts is automatically disabled by hardware. If an interrupt would run with lower priority, higher priority interrupts could interrupt this interrupt handler before the call of `OS_EnterInterrupt()`. This could cause a system crash, when the interrupting interrupt causes a task switch. The priority is set by Interrupt software priority register 0-3. The default value after reset sets all priorities to maximum. The default values should not be changed or should explicitly set to 3 (max value) when hardware for interrupt peripherals is initialized.
- **Top level interrupt must not call any *embOS* function**. As this interrupts may interrupt all others, this could occur before `OS_EnterInterrupt()` is executed in a lower priority interrupt service routine. If any ***embOS*** function causes a task switch then, the system crashes.

5.4. Interrupt-stack

Since the ST7 controller has only one hardware stack pointer, every interrupt uses additional stack space on the system stack.
When the interrupt handler causes a task switch, this task switch is performed at the end of the interrupt handler execution during `OS_LeaveInterrupt()`. The additional stack used by the interrupt handler at that point of execution is therefore also copied to the task stack of the interrupted task.

Extra stack switching during interrupts is not supported with current version of ***embOS*** for ST7 controllers.
`OS_EnterIntStack()` and `OS_LeaveIntStack()` are dummy defines with no functionallity.

Simple Example

```
@interrupt void ISR_uart_rx(void) {
  OS_EnterInterrupt();
  OS_EnterIntStack();      /* Have no effect with embOS ST7
  SignalEvent(&Task,1);    /* any functionality could be here */
  OS_LeaveIntStack();      /* Have no effect with embOS ST7
  OS_LeaveInterrupt();
}
```

6. Power saving modes

ST7 controllers support different power saving modes. These modes may be used to save power consumption during idle times.
Therefore you may activate a power saving mode command in `OS_Idle()`.
Interrupts or reset will wake up the CPU and operation continues.
Please refer to ST7 controller manual about which interrupts can exit from the selected power saving mode.

6.1. SLOW mode

Slow mode may be used as power saving mode, as the ST7 controller remains full functional at reduced clock speed. Therefore CPU may be switched to slow mode in `OS_Idle()`.
The disadvantage of using slow mode is that also the peripheral clock is reduced to the lower frequency, which affects the ***embOS*** timer. All timer dependent functions and software timers are affected and the timing is non deterministic. An other disadvantage is, that any interrupt that may end OS_idle(). All interrupt handlers should therefore be modified to switch of slow mode.

6.2. WAIT mode

WAIT mode places the MCU in a low power consumption mode by stopping the CPU. All peripherals remain active. During WAIT mode, interrupts are automatically enabled and all registers CPU registers remain unchanged.
This power saving mode is selected by calling the 'WFI' instruction which may be placed in OS_Idle().
As all peripherals remain active, the next timer interrupt will resume wait mode and restart the application.

6.3. HALT mode

HALT mode stops CPU and peripherals. Therefore the ***embOS*** timer interrupt can not be used to resume from HALT mode unless external clock is used to drive the hardware timer.
Optionally, if the real time clock is available on the ST7 controller, this could be used as source for ***embOS*** timer interrupt. This timer can not be initialized to produce 1ms resolution. ACTIVE-HALT mode is the lowest power consumption mode of the MCU that could be used in OS_Idle(). Different interrupts may be used to resume from HALT mode. Please refer to ST7 data sheets.

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***. The minimum memory requirement for the kernel itself is about 2000 bytes ROM and 27 bytes RAM.
The table below shows 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	17
Resource semaphore	4
Counting semaphore	2
Mailbox	11
Software timer	9

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

Directory	File	Explanation
Start\	Start_SLR.prj	Start project for release library type.
Start\	Start_SLSP.prj	Start project for stack check library type.
Start\INC\	RTOS.H	Include file for ***embOS***, to be included in every "C"-file using ***embOS*** functions
Start\INC\	iodefine.h	CPU specific special function register definition, used by RTOSInit.c to initialize the hardware required by ***embOS***. This may be replaced by the version required for your target CPU.
Start\LIB\	RTOS.H	Libraries for all memory models and debug options
Start\Link\	Start*.lkf	Linker command files for different targets
Start\SRC\	RtosInit.c	Initializes the hardware, can be modified if required
Start\SRC\	Main.c	Frame program to serve as a start.
Start\SRC\	Vector.c	Interrupt vector table, may be modified.
Start\ZAP\	ST7.INT	Used to setup interrupt simulation with Cosmic's ZAP simulator as shown in our examples.

Any additional file shipped as example.

9. Index