embOS

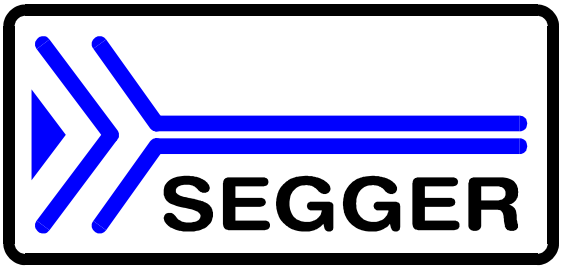
Real Time Operating System

CPU & Compiler specifics for

TOSHIBA TLCS900 CPUs

and TOSHIBA compiler

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*** T900 Real Time Operating System for TOSHIBA TLCS900 32bit series of microcontroller using TOSHIBA CC900 compiler and TOSHIBA Integrated Development Environment.

1.1. How to use this manual

This manual describes all CPU and compiler specifics for ***embOS*** using TLCS900 CPUs with TOSHIBA CC900 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 TOSHIBA Integrated Development Environment. If you have no experience using ***embOS***, you should follow this introduction, even if you do not plan to use TOSHIBA Integrated Development Environment, 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 TLCS900 series of CPUs using TOSHIBA CC900 compiler.

2. Using *embOS* with TOSHIBA Integrated Development Environment

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 TOSHIBA Integrated Development Environment to develop your application, no further installation steps are required. You will find a prepared sample start project workspace for TMP94FD53 CPU, 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 TOSHIBA Integrated Development Environment for your application development in order to become familiar with ***embOS.***

If for some reason you will not work with TOSHIBA Integrated Development Environment, you should:
Copy either all, or only the library-file that you need, to your work-directory. Also copy the RTOSInit.c file and RTOS.f header, which are integral parts of ***embOS***. 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 TOSHIBA Integrated Development Environment, it may be used without the workbench using batch files or a make utility without any problem.

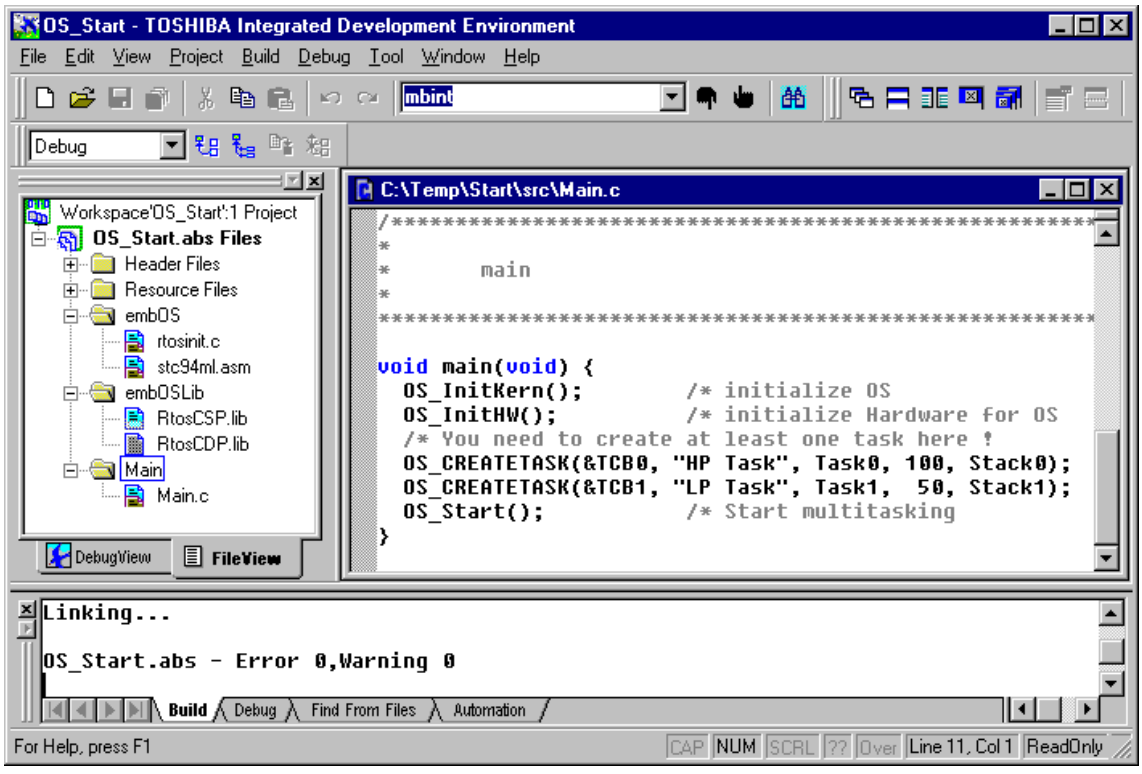
2.2. First steps

After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received a ready to go sample start workspace with a project for TMP94FD53 CPUs 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’ from your ***embOS*** distribution 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’.
- Open the start project workspace ‘OS_Start.tws’. (e.g. by double clicking it)
- Build the start project.

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



For latest information you should open the ReadMe.txt which is part of your project.

2.3. The sample application Main.c

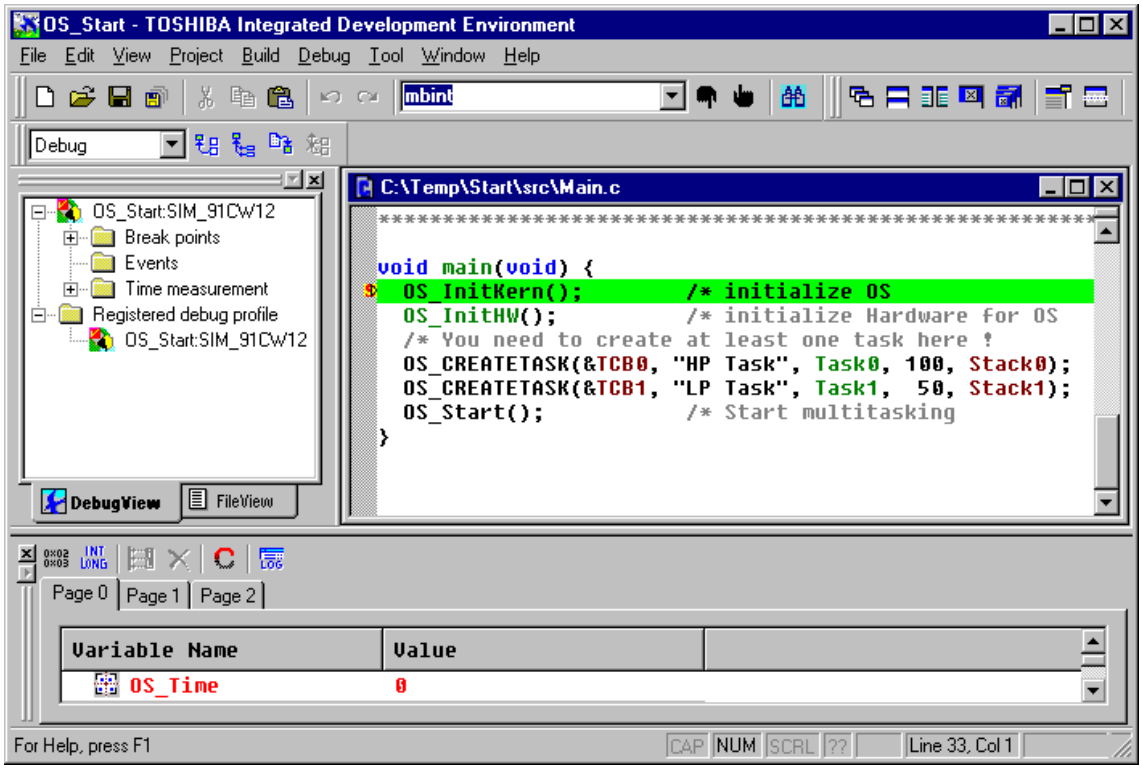
The following is a printout of the sample application main.c. It is a good starting-point for your application.
What happens is easy to see:
After initialization of ***embOS;*** two tasks are created and started
The two tasks are activated and execute until they run into the delay, then suspend for the 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
*
**********************************************************/

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           */
}
```

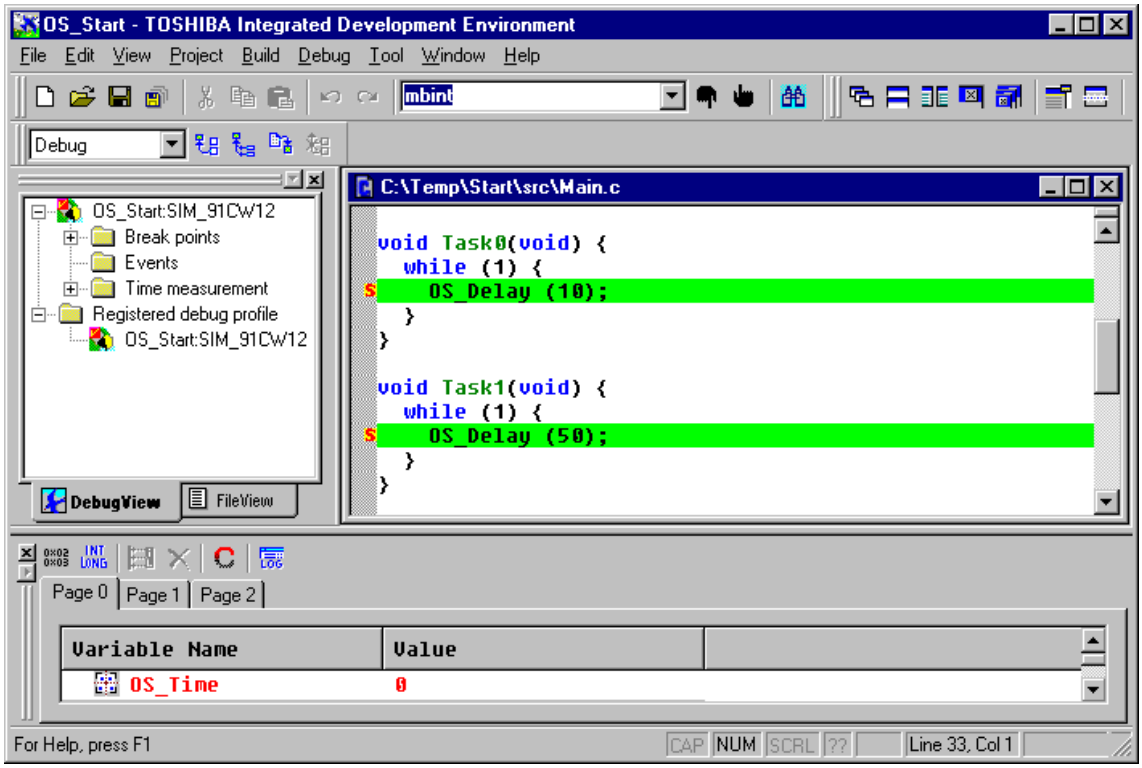
2.4. Stepping through the sample application Main.c

When starting the debugger, you will usually see the main function (very similar to the screenshot below). In some debuggers, 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 is therefore part of your application. Its primary purpose is to initialize the hardware required to generate the timer-tick-interrupt for ***embOS.*** Step through it to see what is done.
`OS_COM_Init()` in `OS_InitHW()` is optional. It is required if embOSView shall be used. In this case it should initialize the UART used for communication.

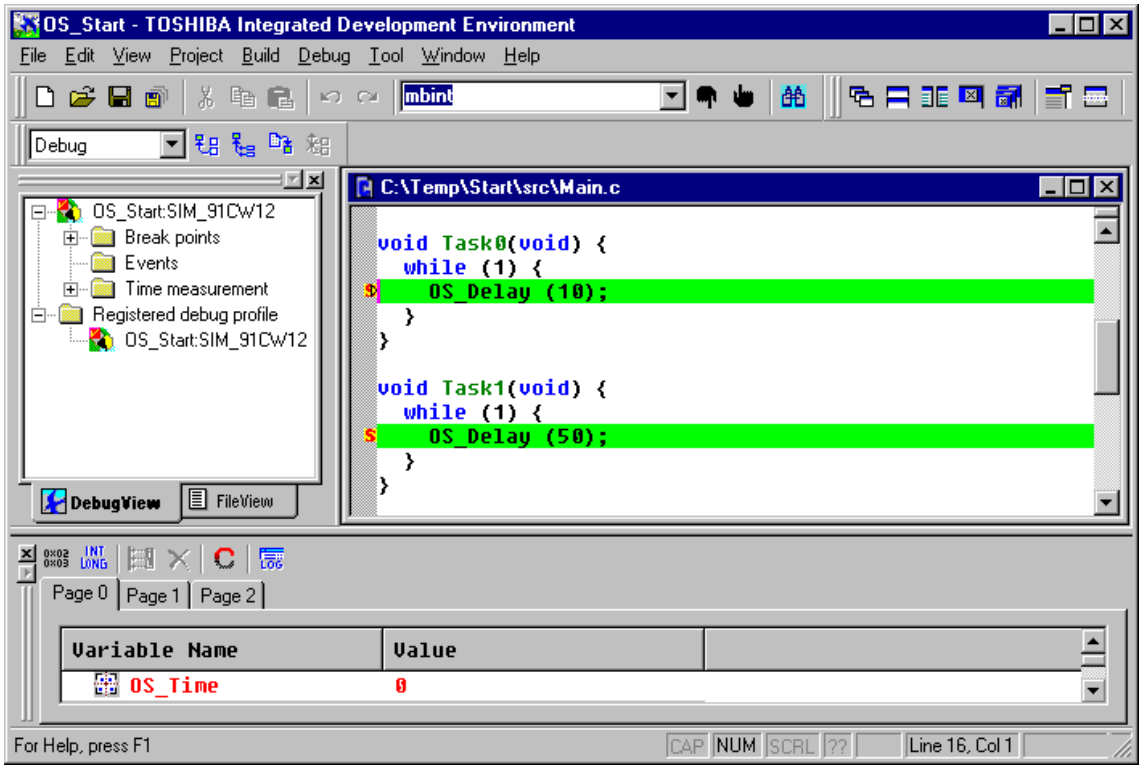
`OS_Start()` should be the last line in main, since it starts multitasking and does not return.



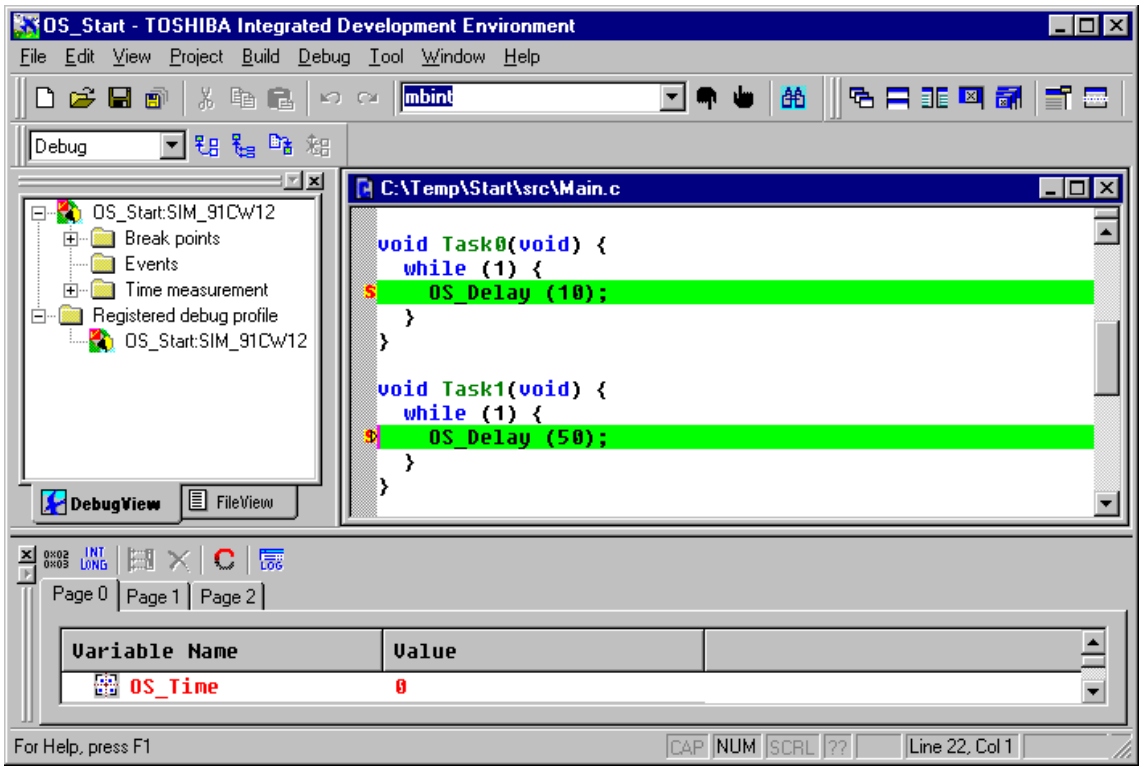
Before you step into `OS_Start()`, you should set breakpoints in the two tasks:



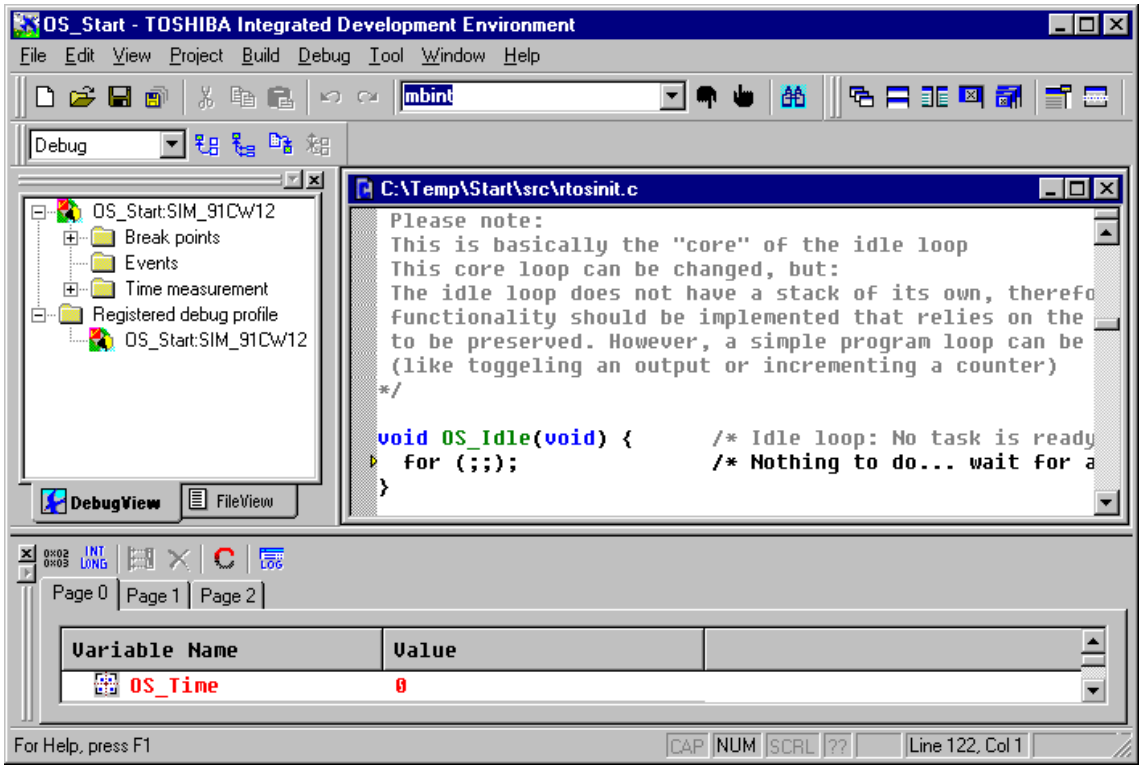
When you step over `OS_Start()`, the next line executed is already in the highest priority task created. (you may also step into `OS_Start()`, then stepping through the task switching process in disassembly mode). In our small start program, `Task0()` is the highest priority task and is therefore active.



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

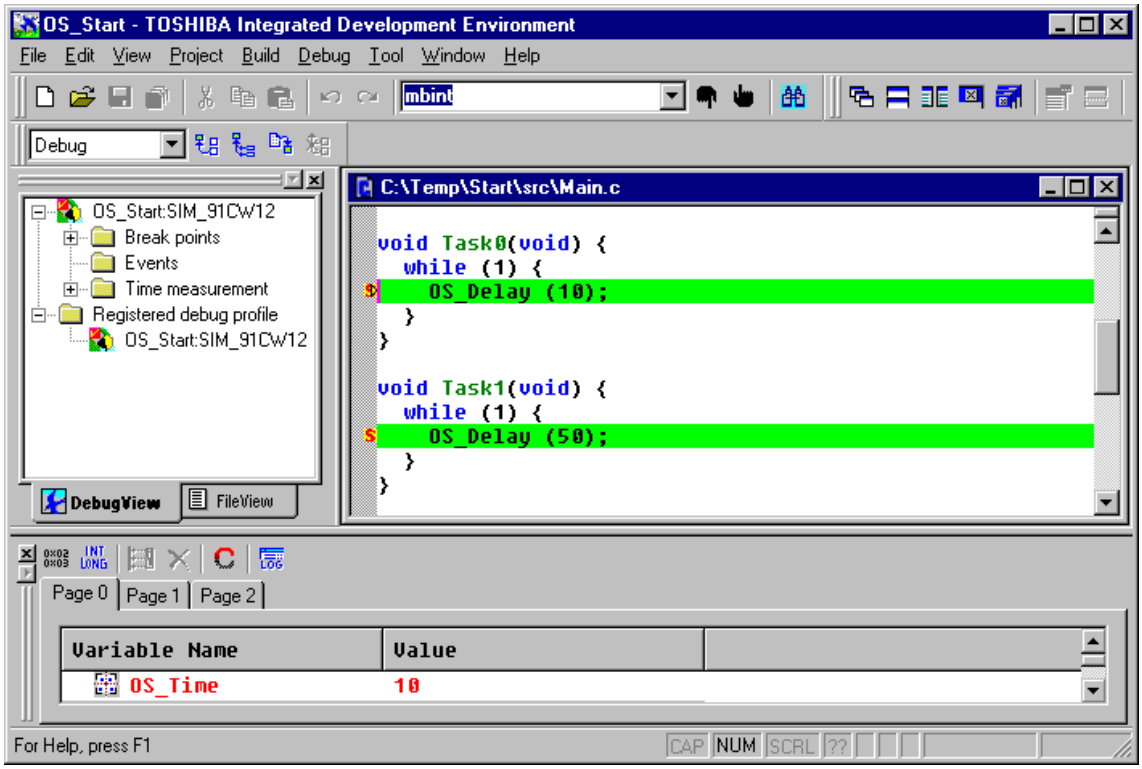


Continuing to step through the program, there is no other task ready for execution. ***embOS*** will suspend Task1 and switch to the idle-loop, which is an endless loop which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).
`OS_Idle()` is found in RTOSInit.c:



If you set a breakpoint in one or 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 to arrive at the highest priority task after its delay is expired.

The watch window shows the system variable `OS_Time`, which shows how much time has expired in the target system.



3. TLCS900 processor and compiler specifics

3.1. Memory models

embOS supports MAX mode memory model only, which means that data, program and constants may reside in any memory location.
Both calling conventions of CC900 compiler are supported:

__cdecl is the default calling convention passing parameters via stack.
__adecl is the assembler like calling convention, passing parameters via register.
The calling convention setting has to be used as global project setting. Different types of ***embOS*** libraries are required according to the calling convention used.

3.2. Available libraries

The files to use depend on global calling convention and library type used.
The library files are located in the subfolder 'Lib' of the start project folder.
The calling convention for your target application has to confirm to the library used in your application.

The naming convention for library files is as follows:

RTOS<CALLING_CONVENTION><LIBRARYTYPE>.lib

<CALLING_CONVENTION> specifies the calling convention:

- **A** for assembler like calling convention (parameter in registers)
- **C** for C style calling convention (parameter passed via stack)

<LIBRARYTYPE> specifies the type of ***embOS*** -library:

- **R** stands for Release build library.
- **S** stands for Stack check library, which performs stack checks during runtime.
- **SP** stands for Stack check and Profiling library, which performs stack checking and additional runtime (Profiling) calculations
- **D** stands for Debug library which performs error checking during runtime.
- **DP** stands for Debug and Profiling library which performs error checking and additional Profiling during runtime.
- **DT** stands for Debug and Trace library which performs error checking and additional Trace functionality during runtime.

Example:

RTOSCSP.lib is the ***embOS*** library for **C**-like calling convention with **S**tack check and **P**rofiling functionality.
It is located in the Start\lib\ subdirectory.

For TLCS900 CPUs running in MAX mode, the following libraries are available (located in the subfolder ‘Start\lib\):

Calling convention	Library type	Library	#define
__adecl	Release	RtosAR.r48	OS_LIBMODE_R
__adecl	Stack-check	RtosAS.r48	OS_LIBMODE_S
__adecl	Stack-check + Profiling	RtosASP.r48	OS_LIBMODE_SP
__adecl	Debug	RtosAD.r48	OS_LIBMODE_D
__adecl	Debug + Profiling	RtosADP.r48	OS_LIBMODE_DP
__adecl	Debug + Profiling + Trace	RtosADT.r48	OS_LIBMODE_DT
__cdecl	Release	RtosCR.r48	OS_LIBMODE_R
__cdecl	Stack-check	RtosCS.r48	OS_LIBMODE_S
__cdecl	Stack-check + Profiling	RtosCSP.r48	OS_LIBMODE_SP
__cdecl	Debug	RtosCD.r48	OS_LIBMODE_D
__cdecl	Debug + Profiling	RtosCDP.r48	OS_LIBMODE_DP
__cdecl	Debug + Profiling + Trace	RtosCDT.r48	OS_LIBMODE_DT

The appropriate define has to be set as project (compiler preprocessor) option according to the library used in your project.

4. Stacks

4.1. Task stack for TLCS900 CPUs

Every ***embOS*** task has to have its own stack. Task stacks can be located in any RAM memory location.
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.
As TLCS CPUs do not support a separate interrupt stack, interrupts also require additional stack space on every task stack.
For TLCS900 CPUs, the minimum task stack size is about 100 bytes.

4.2. System stack for TLCS900 CPUs

The system stack size required by ***embOS*** is 80 bytes. However, since the system stack is also used by the application before the start of multitasking (the call to `OS_Start()`), and because software-timers also use the system-stack, the actual stack requirements depend on the application.
A good value of system stack size is around 160 bytes
To change the system stack size, you have to modify two entries in the assembler startup code file. Normally the system stack is located behind the CPUs internal special function registers. Its size is indirectly given by the value `BaseXSP` which is used to initialize the stack pointer during startup. The size therfore is the difference between `BaseXSP` and the last address of CPUs special function registers. When `BaseXSP` is changed, always also recalculate and change `C_STACK_SIZE` which is used by ***embOS*** for stack checking. Please refer to our modified startup file “stc94ml.asm”.

4.3. Interrupt stack for TLCS900 CPUs

TLCS900 CPUs unfortunately do not deliver a separate stack pointer for interrupts.
Current version of ***embOS*** does not support an additional interrupt stack. Interrupts run on task stacks or the system stack.

4.4. Stack specifics of TLCS900 CPUs

The TLCS900 stack-pointer can address the entire memory area, stacks can be located anywhere in RAM. For performance reasons you should try to locate stacks in fast RAM.

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 interrupt priority level is below or equal to the interrupt priority level of the requesting interrupt source, the interrupt is executed
- the CPU saves PC and status register on the stack
- the CPUs interrupt mask register is set to the priority of the accepted interrupt + 1
- The CPU increments the interrupt nesting counter by 1
- 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 RETI command, restoring PC, status register and decrement interrupt nesting counter by one.

For details, please refer to the CPUs users manual.

5.2. Defining interrupt handlers in "C"

Routines defined with the keyword `__interrupt` automatically save & restore the registers they modify and return with `RETI`.
For a detailed description on how to define an interrupt routine in "C", refer to the C-Compiler's user's guide.

Example

"Simple" interrupt-routine

```
void __interrupt OS_ISR_Tick (void) {
  OS_TickHandler();
}
```

5.3. Defining interrupt vectors (interrupt vector table)

The interrupt vector table is part of the constant structure `_IntTbl[]` found in the “ini*.c” file, which is part of your application / project.
embOS uses one timer interrupt which is initialized during OS_InitHW(). The timer interrupt vector has to be added to the interrupt vector table.
When using embOSView, the interrupt vector for Rx and Tx interrupt of the selected UART have to be defined also.
Our sample start project defines interrupt vectors in TMP94FD53\ini94ml.c

5.4. Interrupt priority

Interrupts of TLCS900 CPUs can not be disabled without changing the interrupt priority.
embOS needs to disable interrupts during internal operations. This is done by macro `OS_DI()` which sets the interrupt priority to 5. OS_DI() is defined in RTOS.h.
OS_DI() keeps interrupts with higher priority enabled and results in very low interrupt latency for those interrupts.

Important rules for interrupts:

- Interrupts with priority above 5 must not call any ***embOS*** function.
- Interrupts with priority above 5 must not re-enable interrupts with lower priority.
- Interrupts with priority below 5 may call any embOS function, when OS_EnterInterrupt() / OS_LeaveInterrupt() or OS_EnterNestableInterrupt() / OS_LeaveNestableInterrupt() are used as described in generic manual.

6. STOP / IDLE / HALT Mode

Usage of the HALT instruction or switching CPU in idle mode are possibilities 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.

Stop mode deactivates internal peripheral clock and can therefore not be exited by ***embOS*** timer interrupt. Stop mode can be released by RESET, NMI or INT0 external interrupt.

7. Using a different CPU

The sample start project of ***embOS*** for TLCS900 was build for and tested with a TMP94FD53 CPU.
To use ***embOS*** with an other CPU, you may build your own new project using TOSHIBA Integrated Development Environment.
When the project is built, there are following steps to do:

- Copy all ***embOS*** libraries into any subfolder of your project.
- Copy RTOSInit.c and RTOS.h into any subfolder of your project.
- Modify RTOSInit.c to fit your CPU
- Modify the startup code
- Modify the ini*.c file
- Add RTOSInit.c and an ***embOS*** library to your project.
- Define OS_LIBMODE as compiler preprocessor option according to the library type used.

7.1. Modification of RTOSInit.c

To use a different CPU, RTOSInit.c may have to be adapted:

- Include the correct IO definition header file.
- Check and modify OS_InitHW() for embOS timer.
- Check and modify OS_GetTimeCycles() which relies on hardware timer.
- Check OS_ConvertCycles2us()
- Check and modify OS_COM_Init() if required
- Check and modify OS_COM_Send1() if required
- Check and modify OS_ISR_Rx() if required
- Check and modify OS_ISR_Tx() if required

7.2. Modification of startup code

The startup code generated by TOSHIBA Integrated Development Environment needs to be modified for use with embOS.
For stack checking, ***embOS*** needs information about stack location and stack size. Therefore the two symbols `BaseXSP` and `C_STACK_SIZE` have to be exported from startup code.
Please refer to our sample startup file “stc94ml.asm”

7.3. Modification of init code and interrupt vector table

For use with ***embOS*** the standard init code file “ini*.c” has to be modified. This file also contains “C”-source init code and the interrupt vector table.
Following modifications should be made:

- Call of __EI() before calling main has to be removed from _Initial().
- Interrupt service routines used by embOS have to be declared.
- Interrupt vector table has to be modified to be used with ***embOS***.

Refer to our sample “ini94ml.c” delivered with ***embOS***.

8. Technical data

8.1. 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 far memory model and release build library.

Short description	ROM [byte]	RAM [byte]
Kernel	approx.1600	41
Event-management	< 200	---
Mailbox management	< 550	---
Single-byte mailbox management	< 300	---
Resource-semaphore management	< 250	---
Timer-management	< 250	---
Add. Task	---	28
Add. Semaphore	---	6
Add. Mailbox	---	14
Add. Timer	---	14
Power-management	---	---

9. Files shipped with ***embOS*** for TLCS900 CPUs

Directory	File	Explanation
root	`*.pdf`	Generic API and target specific documentation
root	`Release.html`	Release notes of ***embOS*** T900
root	`embOSView.exe`	Utility for runtime analysis, described in generic documentation
Start\	`Start*.*`	Start workspace and project files
Start\Lib\	`*.Lib`	***embOS*** libraries
Start\Inc\	`RTOS.h`	To be included in any file using ***embOS*** functions
Start\Src\	`main.c`	Frame program to serve as a start
Start\Src\	`RtosInit.c`	To be compiled & linked with your program, initializes the hardware, can be modified
Start\TMP94FD53	`*.*`	Target CPU specific files.
Start\TMP94FD53	`stc94ml.asm`	CPU startup code, modified to be used with ***embOS***.
Start\TMP94FD53	`ini94ml.c`	“C”-source init code, modified to be used with ***embOS***.

10. Index