embOS

Real Time Operating System

Simulation for

Windows 32 bit system

and Visual C++

Document Rev. 1

A product of Segger Microcontroller Systeme GmbH

www.segger.com

© 1996-2004 Segger Microcontroller Systeme GmbH

Contents

1. About this document

This guide describes how to use ***embOS*** Real Time Operating System Simulation for Windows 32bit operating system and Microsoft Visual Studio tool chain.

embOS Simulation is written in pure ANSI-"C". It uses Windows 32bit API and does in no way rely on Microsoft Visual Studio tool chain.
embOS Simulation may therefore be used with other tool chains and compilers like Borland or Watcom as well.

1.1. How to use this manual

This manual describes all specifics for ***embOS*** simulation running under Windows 32bit operating system using Microsoft Visual C++ development tools. Before actually using ***embOS*** Simulation, 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*** Simulation using Visual C++ compiler. As ***embOS*** Simulation differs from ***embOS*** for target CPU, it is recommended to follow this introduction to get familiar with ***embOS*** Simulation.
Most of the other chapters in this document are intended to provide you with detailed information about functionality and fine-tuning of ***embOS*** Simulation.

2. Using ***embOS*** Simulation with Visual studio

The following chapter describes, how to use ***embOS*** Simulation running under Windows 32bit operating system using Visual studio.

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.

No further installation steps are required. You will find a prepared sample start workspace and project for Visual Studio, which you should use and modify to write your application. So follow the instructions of the next chapter ‘First steps’.

2.2. First steps

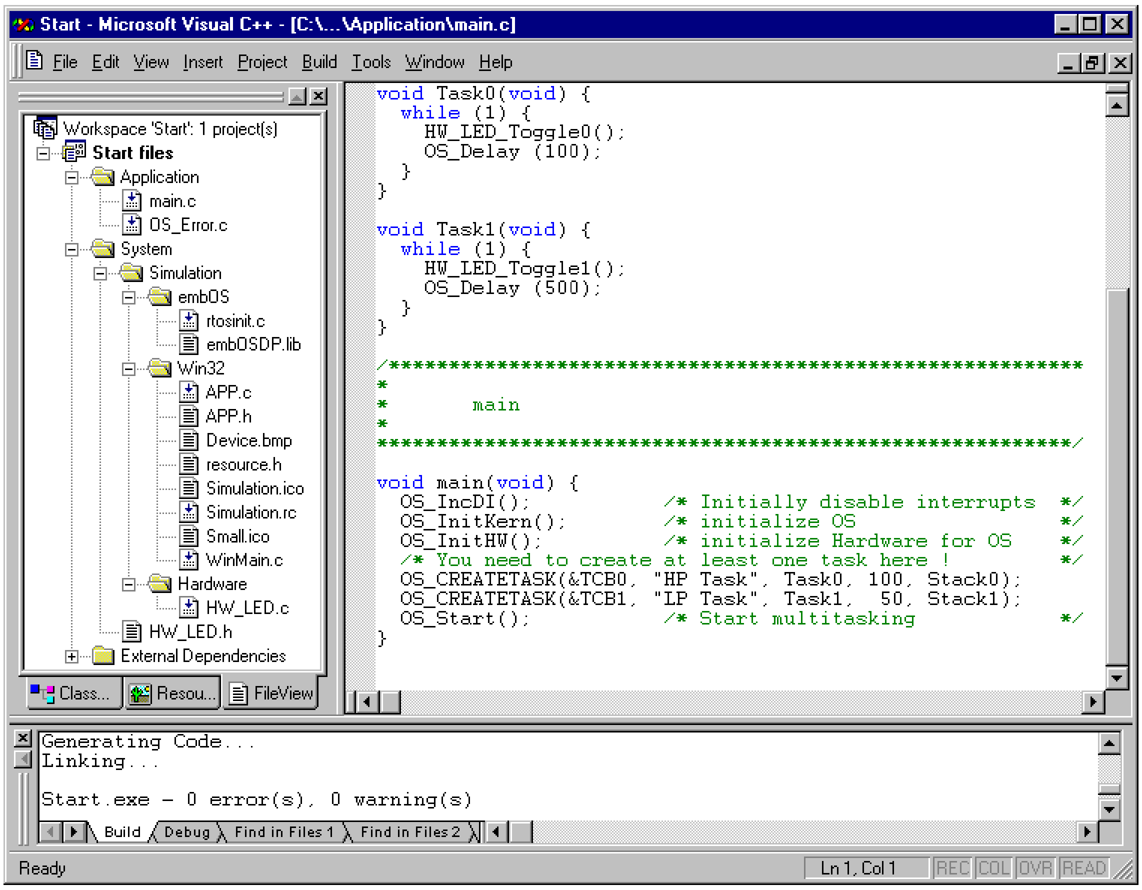
After installation of ***embOS*** Simulation (→ Installation) you are able to create and test your multitasking application. You received a ready to go sample start workspace and project for Visual Studio and it is a good idea to use this as a starting point for all of your applications.

Your ***embOS*** Simulation distribution contains one folder ‘Start’ which contains the sample start workspace and project and every additional files used to build your application.

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

- Create a work directory for your application, for example c:\work
- Copy all files and subdirectories from the ***embOS*** distribution disk into your work directory.
- Clear the read only attribute of all files in the new ‘Start’-folder in your working directory.
- Open the folder ‘Start’ in your work directory.
- Open the start workspace ‘Start.dsw’. (e.g. by double clicking it)
- Build the start project

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



```
void Task0(void) {
  while (1) {
    HW_LED_Toggle0();
    OS_Delay (100);
  }
}

void Task1(void) {
  while (1) {
    HW_LED_Toggle1();
    OS_Delay (500);
  }
}

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

void main(void) {
  OS_IncDI();             /* Initially disable interrupts  */
  OS_InitKern();          /* initialize OS                 */
  OS_InitHW();            /* initialize Hardware for OS    */
  /* You need to create at least one task here !           */
  OS_CREATETASK(&TCB0, "HP Task", Task0, 100, Stack0);
  OS_CREATETASK(&TCB1, "LP Task", Task1,  50, Stack1);
  OS_Start();             /* Start multitasking            */
}
```

```
Generating Code...
Linking...

Start.exe - 0 error(s), 0 warning(s)
```

Initially a target with debug output is selected which can be used with Visual studio debugger to step through the application.

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.

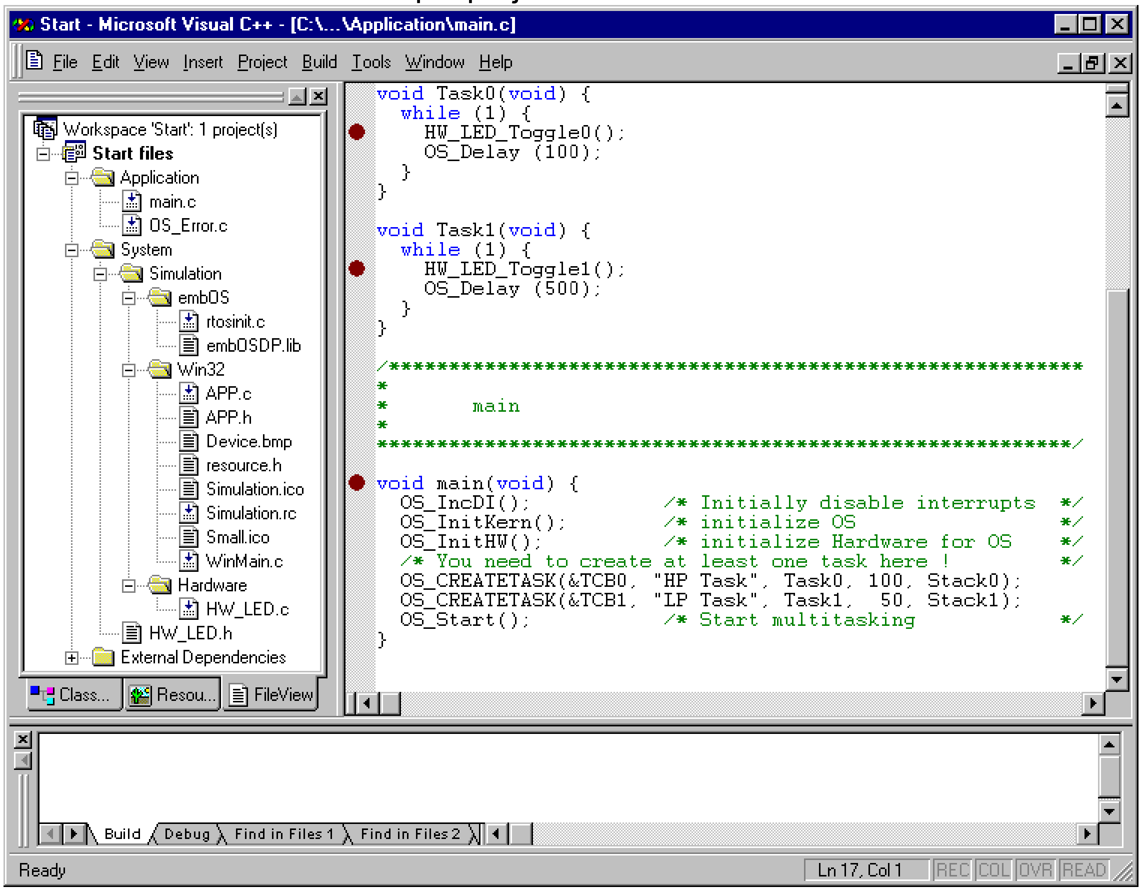
```
/**********************************************************
*          SEGGER MICROCONTROLLER SYSTEME GmbH
*   Solutions for real time microcontroller applications
**********************************************************
File    : Main.c
Purpose : Skeleton program for embOS
--------- END-OF-HEADER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

#include "RTOS.H"
#include "HW_LED.h"

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

void Task0(void) {
  while (1) {
    HW_LED_Toggle0();
    OS_Delay (100);
  }
}

void Task1(void) {
  while (1) {
    HW_LED_Toggle1();
    OS_Delay (500);
  }
}

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

void main(void) {
  OS_IncDI();           /* Initially disable interrupts  */
  OS_InitKern();        /* initialize OS                 */
  OS_InitHW();          /* initialize Hardware for OS    */
  /* You need to create at least one task here !         */
  OS_CREATETASK(&TCB0, "HP Task", Task0, 100, Stack0);
  OS_CREATETASK(&TCB1, "LP Task", Task1,  50, Stack1);
  OS_Start();           /* Start multitasking            */
}
```

3. Using debugging tools to debug the application

The ***embOS*** start project contains a configuration for Visual studio debugger which is normally selected as default configuration.
The following chapters describe a sample debug session based on our sample application main.

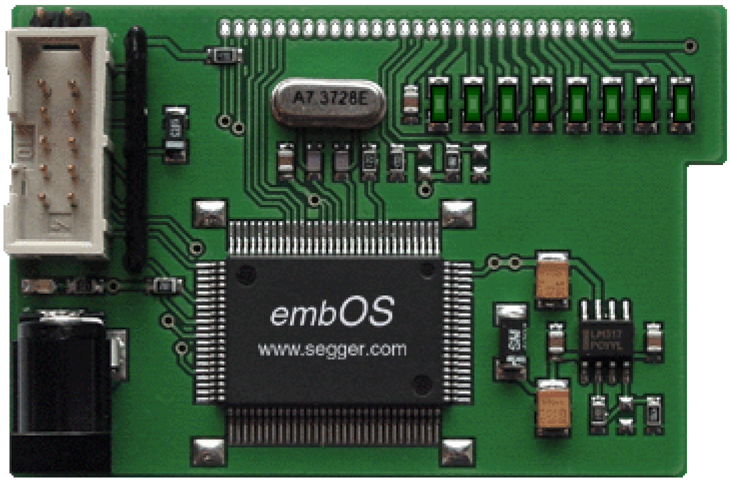
3.1. Using Visual studio debugger

Before starting the debugger, you should set breakpoints into the main function and the two tasks of our sample project in `main.c`.

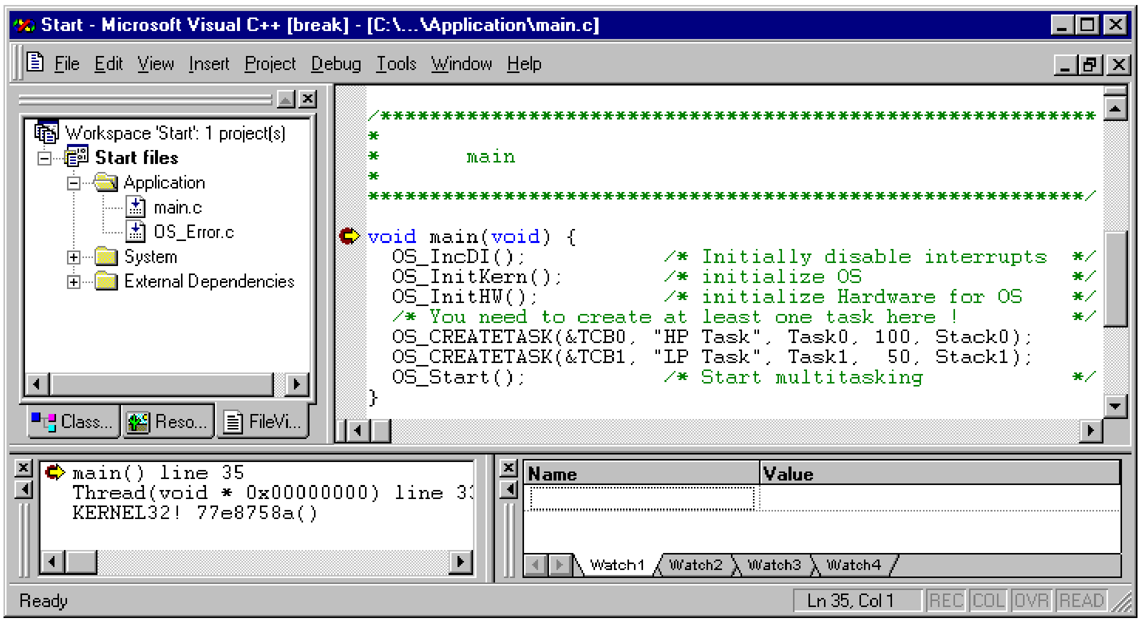


```
void Task0(void) {
  while (1) {
    HW_LED_Toggle0();
    OS_Delay (100);
  }
}

void Task1(void) {
  while (1) {
    HW_LED_Toggle1();
    OS_Delay (500);
  }
}

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

void main(void) {
  OS_IncDI();                 /* Initially disable interrupts  */
  OS_InitKern();              /* initialize OS                 */
  OS_InitHW();                /* initialize Hardware for OS    */
  /* You need to create at least one task here !              */
  OS_CREATETASK(&TCB0, "HP Task", Task0, 100, Stack0);
  OS_CREATETASK(&TCB1, "LP Task", Task1,  50, Stack1);
  OS_Start();                 /* Start multitasking            */
}
```

After setting the breakpoint at main, you may start the debugger by menu “Build | Start Debug | Go” or just press F5.
The ***embOS*** simulation environment is set up and starts a simulated device which is shown before the debugger stops at the breakpoint at main().



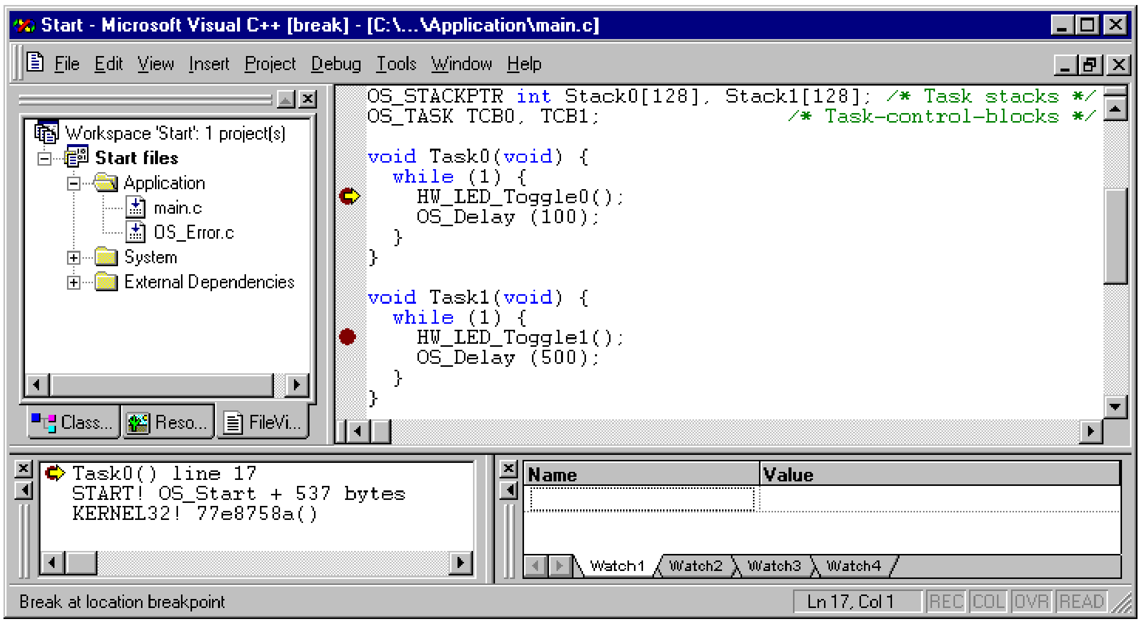
Now you can step through the program.


Start - Microsoft Visual C++ [break] - [C:\...\Application\main.c]
File Edit View Insert Project Debug Tools Window Help
Workspace 'Start': 1 project(s)
Start files
Application
main.c
OS_Error.c
System
External Dependencies
Class... Reso... FileVi...
Name Value
Watch1 Watch2 Watch3 Watch4
Ready
Ln 35, Col 1 REC COL OVR READ

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

void main(void) {
  OS_IncDI();                   /* Initially disable interrupts  */
  OS_InitKern();                /* initialize OS                 */
  OS_InitHW();                  /* initialize Hardware for OS    */
  /* You need to create at least one task here !                 */
  OS_CREATETASK(&TCB0, "HP Task", Task0, 100, Stack0);
  OS_CREATETASK(&TCB1, "LP Task", Task1,  50, Stack1);
  OS_Start();                   /* Start multitasking            */
}
```

```
main() line 35
Thread(void * 0x00000000) line 3:
KERNEL32! 77e8758a()
```

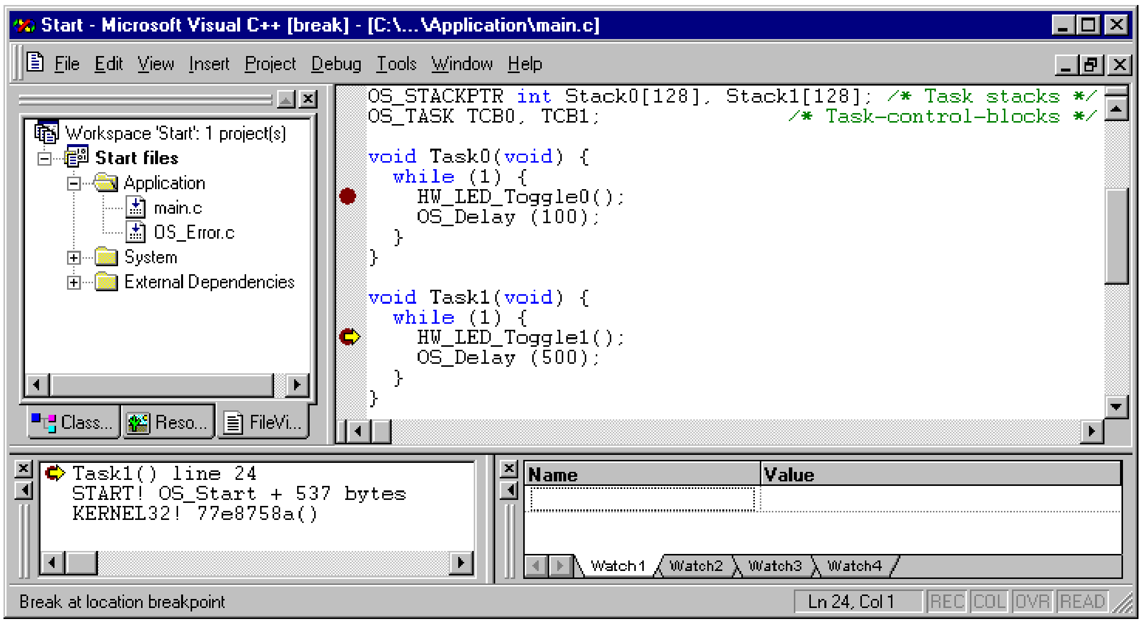
`OS_IncDI()` initially disables interrupts
`OS_InitKern()`initializes ***embOS*** –Variables. As this function is part of the ***embOS*** library, you may step into it in disassembly mode only. Interrupts are not enabled, because of the previous call of `OS_IncDI()`.
`OS_InitHW()` is part of RTOSINIT.c and therefore part of your application. Its primary purpose is to initialize the simulated interrupt required to generate the timer-tick-interrupt for ***embOS.*** Step through it to see what is done.
`OS_COM_Init()`, normally found in `OS_InitHW()` is not implemented, because UART for embOSView can not be simulated.
`OS_Start()` should be the last line in main, since it starts multitasking and does not return.
When you step over `OS_Start()` the next source line executed is Task0 which is the task with the highest priority in our start project and is therefore activated.


Start - Microsoft Visual C++ [break] - [C:\...\Application\main.c]
File Edit View Insert Project Debug Tools Window Help
Workspace 'Start': 1 project(s)
Start files
Application
main.c
OS_Error.c
System
External Dependencies
Class... Reso... FileVi...
Name Value
Watch1 Watch2 Watch3 Watch4
Break at location breakpoint
Ln 17, Col 1 REC COL OVR READ

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

void Task0(void) {
  while (1) {
    HW_LED_Toggle0();
    OS_Delay (100);
  }
}

void Task1(void) {
  while (1) {
    HW_LED_Toggle1();
    OS_Delay (500);
  }
}
```

```
Task0() line 17
START! OS_Start + 537 bytes
KERNEL32! 77e8758a()
```

If you continue stepping, The first LED of our Device will be switched on, Task0 will run into its delay and therefore, ***embOS*** will start the task with lower priority.


Start - Microsoft Visual C++ [break] - [C:\...\Application\main.c]

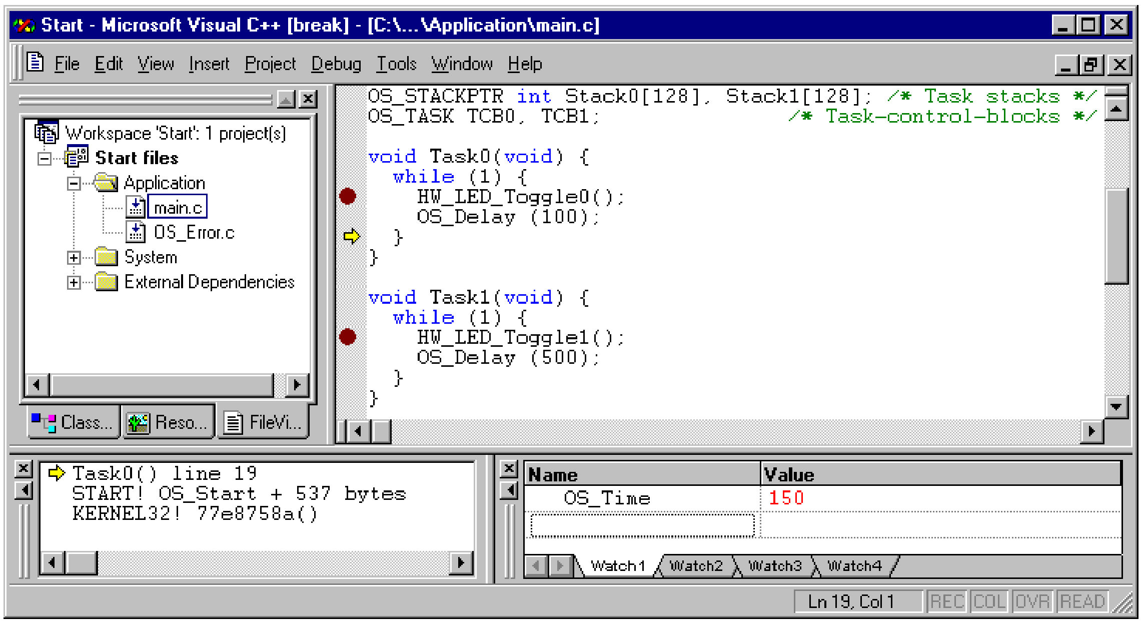
```
OS_STACKPTR int Stack0[128], Stack1[128]; /* Task stacks */
OS_TASK TCB0, TCB1;                      /* Task-control-blocks */

void Task0(void) {
  while (1) {
    HW_LED_Toggle0();
    OS_Delay (100);
  }
}

void Task1(void) {
  while (1) {
    HW_LED_Toggle1();
    OS_Delay (500);
  }
}
```

```
Task1() line 24
START! OS_Start + 537 bytes
KERNEL32! 77e8758a()
```

Break at location breakpoint — Ln 24, Col 1

Continuing to step through the program, Task1 will switch on LED1 and then run into its delay.
Please note, that the simulated device may not show the correct state of LEDs immediately. Default settings update the screen for the simulated device every 20 ms. How to modify this behavior is described later on in this manual.
As there is no other task ready for execution when Task1 runs into its delay, ***embOS*** will suspend Task1 and switch to the idle process, which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).
embOS Simulation does not contain an `OS_Idle()` function which is implemented in normal ***embOS*** ports. Idle times are spent in Windows idle process.

When you step over the OS_Delay() function of Task1, you will arrive back in Task0 after the delay of Task0 expired:


Start - Microsoft Visual C++ [break] - [C:\...\Application\main.c]

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

void Task0(void) {
  while (1) {
    HW_LED_Toggle0();
    OS_Delay (100);
  }
}

void Task1(void) {
  while (1) {
    HW_LED_Toggle1();
    OS_Delay (500);
  }
}
```

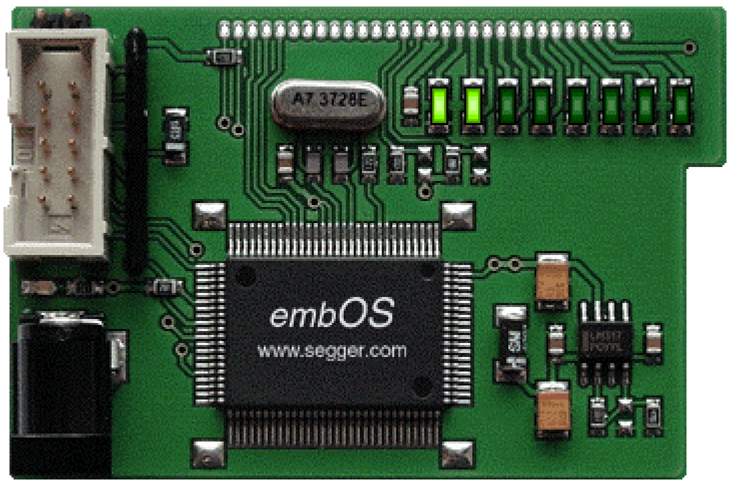
```
Task0() line 19
START! OS_Start + 537 bytes
KERNEL32! 77e8758a()
```

Name	Value
OS_Time	150

Ln 19, Col 1

As can be seen by the value of ***embOS*** timer variable `OS_Time`, shown in the watch window, Task0 continues operation after expiration of the 100 ms delay.
Please note, that delays seem to be longer than expected . When the debugger stops at a breakpoint, it takes some time until the screen is updated and the

`OS_Time` variable is examined. Therefore `OS_Time` may shows larger values than expected.
After the delay of Task0 expired, the simulated device shows both LEDs lit as can be seen in the following screenshot:



You may now disable the two breakpoints in our tasks and execute the debug “Go” command to see how the simulated device runs in real time.

3.2. Common debugging hints

For debugging your application, you should use a debug build, e.g. use the debug build libraries in your projects if possible. The debug build contains additional error check functions during runtime.
When an error is detected, the debug libraries call `OS_Error()`, which is defined in file `OS_Error.c`.
You should set a breakpoint at `OS_Error()`. The actual error code is assigned to the global variable `OS_Status`. The program then waits for this variable to be reset. This allows to get back to the program-code that caused the problem easily: Simply reset this variable to 0 and you can step back to the program sequence causing the problem. Most of the time, a look at this part of the program will make the problem clear.
The debug library is used per default in our sample start project.
How to select an other library for your projects is described later on in this manual.

4. Build your own application

To build your own application, you should always start with a copy of the sample start project and edit or add files in this copied project.
This has the advantage, that all necessary files are included and all settings for the project are already done.
You should NOT create a new project. This is of course possible, but it is more difficult and error prone and requires a solid knowledge of the Visual Studio and compiler / linker tool chain.

4.1. Required files for an ***embOS*** Simulation

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

- **RTOS.h** from sub folder "Inc\".
  This header file declares all ***embOS*** API functions and data types and has to be included in any source file using ***embOS*** functions.
- **RTOSInit.c** from subfolder "System\Simulation\embOS\".
  It contains initialization code for ***embOS*** timer interrupt handling and simulation.
- **OS_Error.c** from subfolder "Application\".
  It contains the ***embOS*** runtime error handler `OS_Error()`which is used in stack check or debug builds.
- One ***embOS*** **library** from the "System\Simulation\embOS\" subfolder. Normally ***embOS*** Simulation comes with one library built for debug and profiling support.
- **WinMain.c** from subfolder "System\Simulation\Win32\".
  It contains the initialization code for ***embOS*** Simulation environment under Windows 32bit operation system and the Windows message loop.
- **Sim.c** from subfolder "System\Simulation\Win32\".
  It contains initialization and update handling of the simulated device. This file may be modified to support your own simulated device as described later in this manual.
- **Resource files** from subfolder "System\Simulation\Win32\".
  These resource files contain commands and resources used for ***embOS*** Simulation running under Windows 32bit operating system.

Before calling any other ***embOS*** function, your main() function should call `OS_IncDI()` and then has to initialize ***embOS*** by call of `OS_InitKern()`.
The call of `OS_IncDI()` before `OS_InitKern()` inhibits enabling of interrupts during execution of `OS_InitKern()`.
After OS_InitKern(), you should call `OS_InitHW()` to initialize ***embOS*** timer interrupt.

4.2. Select a start project configuration

embOS Simulation comes with one start project which includes two configurations:

- **Start - Win32 Debug** is the configuration normally used for development and debugging.

- **Start - Win32 Release** may be used to build an executable, which shows the application and simulated device and can be started on any PC running with Windows 32bit operating system. It may be used for demonstration purposes.

4.3. Add your own code

For your own code, you may add a new folder to the project or you may add additional files to the “Application” folder.
You should then modify or replace the main.c source file in the subfolder “Application”.

4.4. Change library mode

For your real application you may wish to use a different ***embOS*** library mode. For ***embOS*** Simulation, there is no need to use an other library mode, because additional runtime error checks or additional memory requirements do not care.

4.5. Rebuilding the ***embOS*** library

An other library for ***embOS*** Simulation can only be built using the source version of ***embOS*** Simulation.

- Modify the entry `#define OS_LIBMODE_DP` in the ***embOS*** port specific file `OS_Chip.h.`
- Modify the “M.bat” batch file in the root directory of the ***embOS*** Simulation source distribution to name the library according the library mode which should be used.
- Finally start “M.bat” to produce a new “Start” folder which then contains the new library.

5. Device simulation

embOS Simulation contains additional functions to visualize and simulate a hardware device. This may be used to simulate and visualize hardware ports, LEDs or displays which would normally be controlled with functions running on your real hardware. The following chapter describes how device simulation works on our sample project and how this simulation can be modified to simulate your own hardware.

5.1. How device simulation works

During startup of the Windows application in `WinMain()`, the bitmap “Device.bmp” is set up and shown on the screen.

- The device bitmap is shown as main window of ***embOS*** Simulation.
- `SIM_Init()`, called from `WinMain()` initializes a Windows timer which periodical sends a `WM_TIMER` message to the main window.
- The window procedure of the main window handles the `WM_TIMER` message and calls the timer callback function that was assigned to the timer during creation.
- The timer callback function (found in `SIM.c`) invalidates the main window, which results in a `WM_PAINT` message sent to the main window.
- The windows procedure of the main window handles the `WM_PAINT` message and calls `SIM_Paint()` which is responsible to draw the device bitmap.

5.2. Modify periodical device update rate

The device update rate is controlled by the timer interval value which is set during creation of the update timer in `SIM_Init()`.
The default update interval is set to 20 ms.
According to your needs, you may modify the update interval. Therefore modify the following line in `SIM_Init()` found in `SIM.c`:

```
  SetTimer(hMainWindow, 0, 20, _cbTimer);
```

The third parameter of `SetTimer()` specifies the update interval im ms.
The update interval should not be set to short to avoid high CPU load. If short update reaction is required, it might be better to force immediate device update on demand as described below.

5.3. Force immediate device update

Our sample device consist of 8 LEDs which can be switched on, off or can be toggled by the sample hardware specific functions declared in our sample code file `HW_LED.c`
During the sample debug session described in chapter 3, you may have noticed, that our sample device was not updated immediately when we called the `HW_LED_Toggle()` functions from our two sample tasks.
Per default, a timer is used to update and redraw the simulated device periodically as described in previous chapter.
The function `SIM_Update()` from `SIM.c` may be called to force an immediate update of the simulated device. This function does not take parameters and does not return any value.

You may modify our sample code in HW_LED.c as shown below to force immediate device update when any LED function is called.

```
/*********************************************************************
*
*       HW_LED_Set0
*/
void LED_SetLED0(void) {
  _LEDs |= (1 << 0);
  SIM_Update();  /* Force immediate device update */
}

/*********************************************************************
*
*       HW_LED_Clr0
*/
void HW_LED_Clr0(void) {
  _LEDs &= ~(1 << 0);
  SIM_Update();  /* Force immediate device update */
}

/*********************************************************************
*
*       HW_LED_Toggle0
*/
void HW_LED_Toggle0(void) {
  _LEDs ^= (1 << 0);
  SIM_Update();  /* Force immediate device update */
}
```

You may then restart the sample application to see the different behavior.

5.4. How to use your own simulated device

Any Windows bitmap file can be used and shown as main window of ***embOS*** Simulation to visualize a simulated device.
To display the real contour of the device, the background outside the device contour may be filled with one specific color, which is treated as transparent when the bitmap is shown on the screen.
This “transparent” color should of course not exist in the device itself.

To display your device on the screen during simulation, proceed as follows:

- Create a bitmap file of your device.
- Follow the contour of your device and fill the background with one color, that does not exist in your device image to make the background transparent. Per default, we use pure red in our sample.
- Save the device as “Device.bmp” in subfolder `“System\Simulation\Win32”` of your start project.
- In `WinMain.c` check or modify the entry `_rgbTransparent` which defines the transparent color.
- To simulate visual elements of your device, write or modify the `SIM_Paint()` function in `SIM.c`.

Finally you have to write your own “hardware” file similar to our `HW_LED.c` sample file which transforms your hardware outputs to any memory variables or states which can be visualized by your `SIM_Paint()` function.

6. Stacks

The following chapter describes stack handling of ***embOS*** Simulation running under Windows 32bit operating system.

6.1. Task stacks

Every ***embOS*** task has to have its own stack. Task stacks can be located in any RAM memory location.
In ***embOS*** Simulation, every task runs as a separate thread under Windows 32bit operating system. The real “task” stack is therefore managed by Windows. Declaration and size of task stacks in your application are necessary for generic ***embOS*** functions, but do not affect the stack size of the generated Windows thread. A stack check and stack overflows can therefore not be simulated.

6.2. System stack

The system stack used during startup and ***embOS*** internal functions is controlled by Windows 32bit operating system, because simulated startup and system calls run in a Windows thread. Stack checking of system stack can therefore not be simulated.

6.3. Interrupt stack

Simulated interrupts in ***embOS*** Simulation run as Windows thread with higher priority. As the threads stacks are managed by Windows 32bit operating system, the interrupt stacks will never overflow and stack check can not be simulated.

7. Interrupts

With ***embOS*** Simulation, interrupts have to be simulated. This requires modified interrupt service routines which differ from those used in your embedded application. The following chapter describes interrupt simulation and handling in ***embOS*** Simulation running on Windows 32bit operating system.

7.1. How interrupt simulation works

With ***embOS*** Simulation, all interrupt handler functions are started as individual threads running at highest priority.
Under Windows 32bit operating system, interrupt handler functions run with priority `THREAD_PRIORITY_TIME_CRITICAL`.
Because ***embOS*** might have to disable interrupts when ***embOS*** internal operations are performed, ***embOS*** Simulation has to be able to suspend and resume interrupt handler threads. This requires, that all interrupt handler threads have to be created and installed by the special ***embOS*** Simulation API function `OS_SIM_CreateISRThread()`.

Interrupt simulation under ***embOS*** Simulation functions as follows:
- An interrupt handler is written as normal “C”-function without parameter or return value.
- The interrupt handler function is started and initialized as Windows thread running at highest priority by a call of `OS_SIM_CreateISRThread()`.
- The interrupt handler function runs as endless loop which consists a `Sleep_Ex()` function call.
- As soon as the interrupt handler calls `Sleep_Ex()`, the thread is suspended by Windows operating system.
- The interrupt is triggered by an “APC” function call which is used to resume the interrupt simulation thread suspended by `Sleep_Ex()`.
- The interrupt handler function is resumed by Windows and performs the required function.
- The interrupt handler running as endless loop calls `Sleep_Ex()` again and suspends until it is resumed again by the associated “APC” function call.

To simulate a periodical interrupt, a Windows timer object can be used to call an APC function for the simulated interrupt handler, or `Sleep_Ex()` can be called with timeout value.

7.2. Defining interrupt handlers for simulation

Interrupt handlers used in ***embOS*** Simulation can not handle the real hardware normally used in your target application. The interrupt handler functions of your target application have to be replaced by a modified version which can be used in simulation.

Example

Interrupt simulation routine for periodical interrupt generation.

```
static void _ISRTimerThread(void) {
  while (1) {
    OS_EnterInterrupt();    /* Tell embOS that interrupt code is running */
    DoTimerHandling();      /* Any functionality can be added here       */
    OS_LeaveInterrupt();    /* Tell embOS that interrupt code ends       */
    SleepEx(10, TRUE);      /* Suspend until delay expired or triggered  */
                            /* by “APC” function                         */
  }
}
```

How to create and use a timer object which periodical calls an “APC” function to signal the interrupt thread can be seen in RTOSInit.c. There we use a timer object to signal the ***embOS*** timer interrupt thread.
Please note:
`OS_EnterInterrupt()` or `OS_EnterNestableInterrupt()` has to be called before any other *embOS* function is called from the interrupt handler.

7.3. Interrupt handler installation

Any interrupt handler in ***embOS*** Simulation has to be installed by a call of `OS_SIM_CreateISRThread()`

Prototype

```
OS_U32 OS_SIM_CreateISRThread(void (*pStartAddress)(void))
```

Parameter	Meaning
`pStartAddress`	Pointer to a void function which serves as simulated interrupt handler

Return value

OS_U32 value is the handle to the created interrupt simulation thread.

Add. information

The returned thread handle may be used to create and assign a synchronization object for the created thread. An example on how to use this handle for creation of a timer object to periodical signal an interrupt can be seen in our RTOSInit.c where it is used for ***embOS*** timer interrupt simulation.

7.4. Interrupt priorities

With ***embOS*** Simulation, all simulated interrupts, installed with function `OS_SIM_CreateISRThread()` run on the same ISR thread priority, which is `THREAD_PRIORITY_TIME_CRITICAL`. The order of thread activation is scheduled by Windows 32bit operating system and can not be influenced or predicted.

8. Files shipped with *embOS* Simulation

embOS Simulation for Windows 32bit operating system is shipped with documentation in PDF format and release notes as html.
The start project, workspace, source files, ***embOS*** library and additional files required for debugging and simulation are located in the sub folder 'Start'. The distribution of ***embOS*** Simulation contains the following files:

Directory	File	Explanation
Start\	`Start.dsw`	Start project workspace for Visual studio.
Start\	`Start.dsp`	Start project for Visual studio.
Start\Inc\	`RTOS.h`	***embOS*** API header file. To be included in any file using ***embOS*** functions
Start\Application\	`main.c`	Frame program to serve as a start
Start\Application\	`OS_Error.c`	***embOS*** runtime error handler.
Start\System\Simulation\embOS\	`RtosInit.c`	Hardware setup functions used for ***embOS*** Simulation
Start\System\Simulation\embOS\	`embOSDP.lib`	***embOS*** library used for Simulation
Start\System\Simulation\Win32\	`*.*`	Windows specific files used for ***embOS*** Simulation
Start\System\Simulation\Hardware\	`*.*`	Sample files for hardware simulation used with ***embOS*** Simulation
GenOsSrc\	`*.*`	***embOS*** sources (Source version only)
CPU\	`*.*`	***embOS*** port specific sources (Source version only)
	`*.Bat`	Batch files to build ***embOS*** libraries from sources (Source version only)

The manuals are found in the root directory of the distribution.

9. Index