embOS

Real Time Operating System

CPU & Compiler specifics for

ST STM8 CPUs

and IAR compiler for STM8

Document Rev. 0

SEGGER

A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Contents

1. About this document

This guide describes how to use ***embOS*** for STM8 Real Time Operating System for the ST STM8 series of microcontroller using the IAR compiler for STM8 and the IAR Embedded Workbench.

1.1. How to use this manual

This manual describes all CPU and compiler specifics for ***embOS*** STM8 for IAR 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 contains a step-by-step introduction, how to install and use ***embOS*** using IAR Embedded Workbench. If you have no experience using ***embOS***, you should follow this introduction, even if you do not plan to use C-SPY or the IAR Embedded Workbench, 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 STM8 CPUs using the IAR compiler.

2. Using *embOS* with IAR Embedded 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 the IAR Embedded Workbench to develop your application, no further installation steps are required. You will find a prepared sample workspace including one start project, 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 IAR Embedded Workbench for your application development in order to become familiar with ***embOS.***

If for some reason you will not work with the IAR Embedded Workbench, you should:
Copy either all or only the library-file that you need to your work-directory. Also copy the hardware initialization file RTOSInit*.c and the ***embOS*** header file RTOS.h. 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 the IAR Embedded Workbench, it may be used without the workbench using batch files or a make utilities without any problem.

2.2. First steps

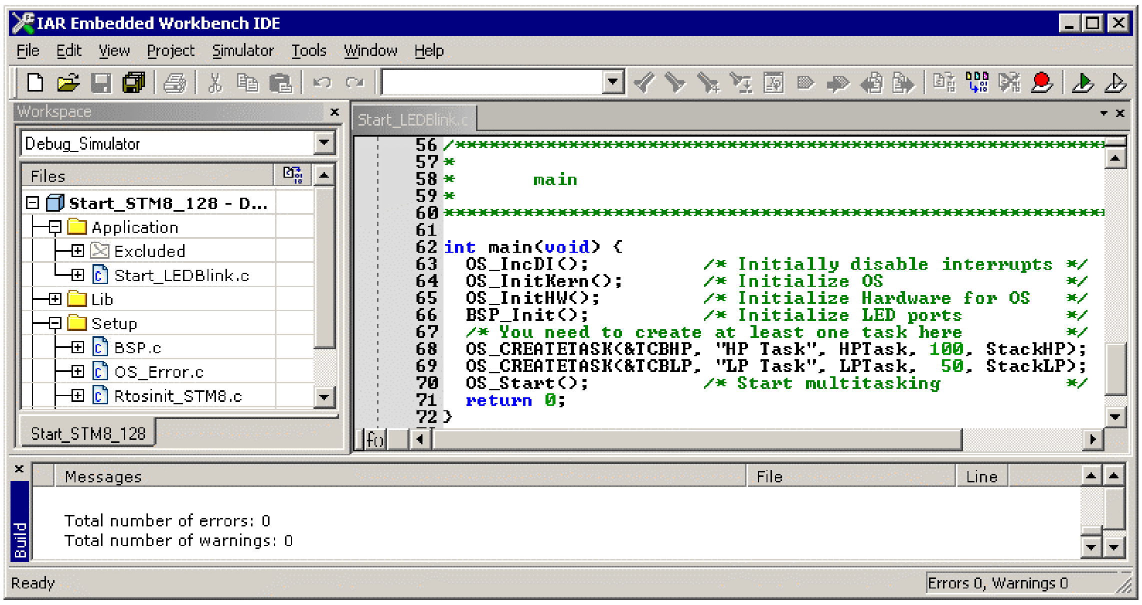
After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received ready to go sample workspaces and start project for an STM8S CPU and it is a good idea to use this project as a starting point for all of your applications.

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\BoardSuport\ST\'.
- Open one of the sample workspaces in one of the subfolders of Start\BoardSupport\ST\ *.eww'. (e.g. by double clicking it)
  The documentation is based on the Start_STM8_128.eww sample workspace for the ST STM8_128_Eval-Board.
- Select the configuration "Debug_Simulator"
- Build the start project

Further examples in this manual show the configuration for the CSpy simulator. Using the ST-Link tool and CSpy for debugging in real hardware is supported by another configuration and is similar and looks the same.

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



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

int main(void) {
  OS_IncDI();                      /* Initially disable interrupts */
  OS_InitKern();                   /* Initialize OS                */
  OS_InitHW();                     /* Initialize Hardware for OS   */
  BSP_Init();                      /* Initialize LED ports         */
  /* You need to create at least one task here                     */
  OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
  OS_Start();                      /* Start multitasking           */
  return 0;
}
```

```
Total number of errors: 0
Total number of warnings: 0
```

For additional information you should open the ReadMe.txt which is part of the sample project.

2.3. The sample application Start_LEDBlink.c

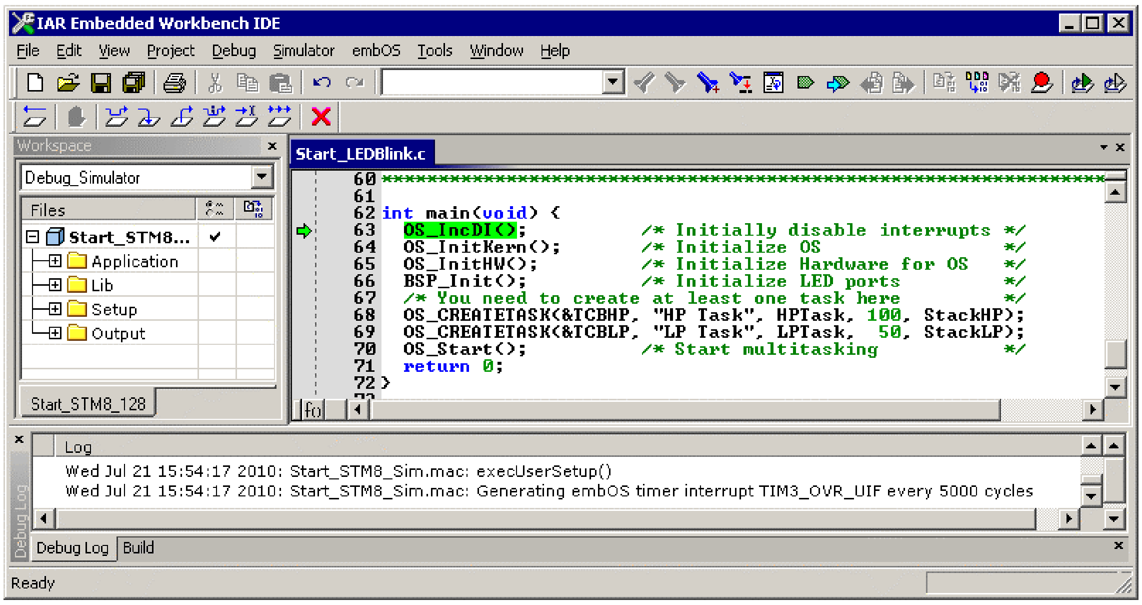
The following is a printout of the sample application Start_LEDBlink.c. It is a good starting-point for your application.
What happens is easy to see:
After initialization of ***embOS***, two tasks are created and started.
The two tasks are activated and execute until they run into the delay, then suspend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

```
/*********************************************************************
*                  SEGGER MICROCONTROLLER GmbH & CoKG
*         Solutions for real time microcontroller applications
**********************************************************************
File    : Start_LEDBlink.c
Purpose : Sample program for OS running on EVAL-boards with LEDs
--------- END-OF-HEADER --------------------------------------------*/

#include "RTOS.h"
#include "BSP.h"

OS_STACKPTR int StackHP[128], StackLP[128];          /* Task stacks */
OS_TASK TCBHP, TCBLP;                        /* Task-control-blocks */

static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_Delay (50);
  }
}

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
  }
}

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

int main(void) {
  OS_IncDI();                      /* Initially disable interrupts */
  OS_InitKern();                   /* Initialize OS                */
  OS_InitHW();                     /* Initialize Hardware for OS   */
  BSP_Init();                      /* Initialize LED ports         */
  /* You need to create at least one task before calling OS_Start() */
  OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
  OS_Start();                      /* Start multitasking           */
  return 0;
}
```

2.4. Stepping through the sample application Main.c using CSpy

When starting the debugger, you will usually see the main function (very similar to the screenshot below). If you may look at the startup code, you have to set a breakpoint at `main()`. Now you can step through the program.
`OS_IncDI()` initially disables interrupts.
`OS_InitKern()` is part of the ***embOS*** library; you can therefore only step into it in disassembly mode. It initializes some relevant OS-Variables. Because of the previous call of `OS_IncDI()`, interrupts are not enabled during execution of `OS_InitKern()`.

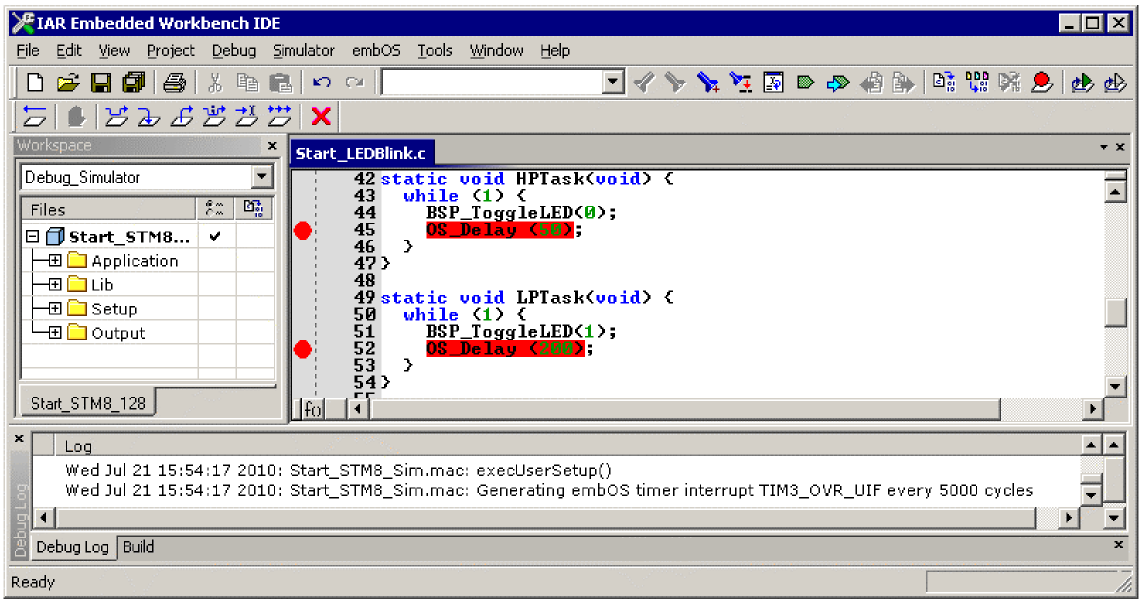
`OS_InitHW()` is part of the CPU specific RTOSInit*.c file and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the timer-tick-interrupt for ***embOS.*** Step through it to see what is done.
`OS_COM_Init()` in `OS_InitHW()` is optional. It is 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.



```
***************************************************************

int main(void) {
  OS_IncDI();                  /* Initially disable interrupts */
  OS_InitKern();               /* Initialize OS                */
  OS_InitHW();                 /* Initialize Hardware for OS   */
  BSP_Init();                  /* Initialize LED ports         */
  /* You need to create at least one task here                 */
  OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
  OS_Start();                  /* Start multitasking           */
  return 0;
}
```

```
Log
Wed Jul 21 15:54:17 2010: Start_STM8_Sim.mac: execUserSetup()
Wed Jul 21 15:54:17 2010: Start_STM8_Sim.mac: Generating embOS timer interrupt TIM3_OVR_UIF every 5000 cycles
```

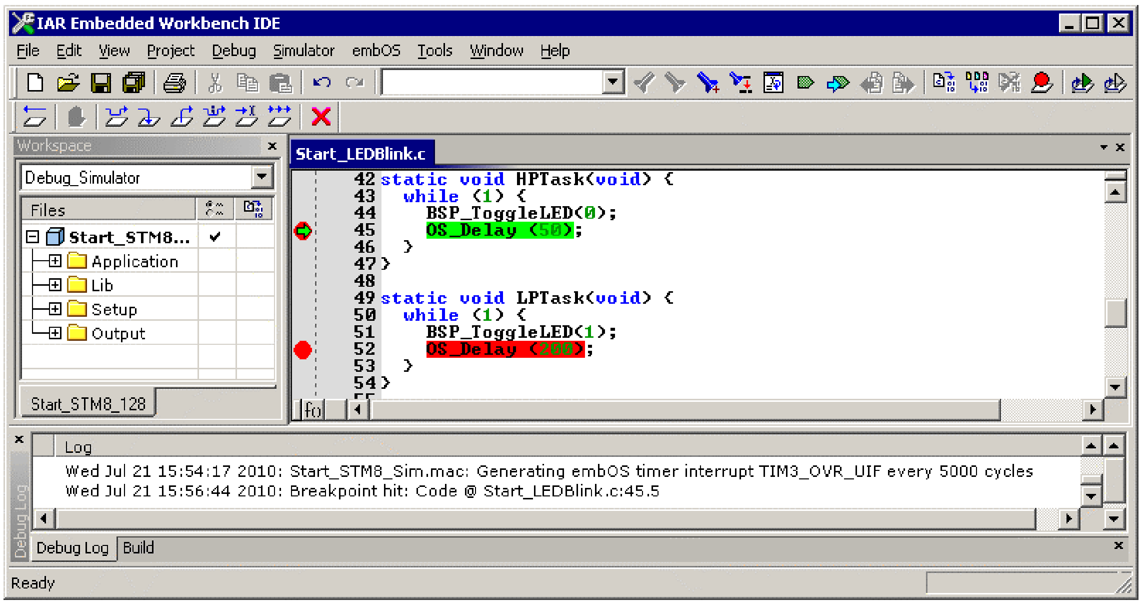
Before you step into `OS_Start()`, you should set breakpoints in the two tasks:



```
static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_Delay (50);
  }
}

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
  }
}
```

```
Log
Wed Jul 21 15:54:17 2010: Start_STM8_Sim.mac: execUserSetup()
Wed Jul 21 15:54:17 2010: Start_STM8_Sim.mac: Generating embOS timer interrupt TIM3_OVR_UIF every 5000 cycles
```

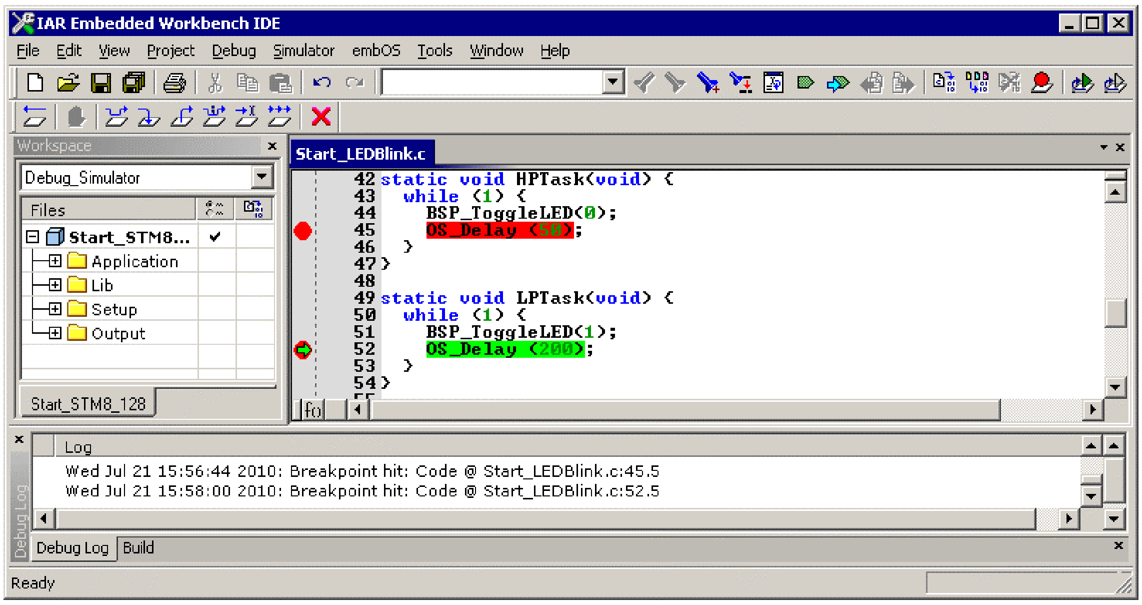
When you step over `OS_Start()`, the next line executed is already in the highest priority task created. (you may also step into `OS_Start()`, then stepping through the task switching process in disassembly mode). In our small start program, `HPTask()` is the highest priority task and is therefore active.



```
42 static void HPTask(void) {
43   while (1) {
44     BSP_ToggleLED(0);
45     OS_Delay (50);
46   }
47 }
48
49 static void LPTask(void) {
50   while (1) {
51     BSP_ToggleLED(1);
52     OS_Delay (200);
53   }
54 }
```

```
Wed Jul 21 15:54:17 2010: Start_STM8_Sim.mac: Generating embOS timer interrupt TIM3_OVR_UIF every 5000 cycles
Wed Jul 21 15:56:44 2010: Breakpoint hit: Code @ Start_LEDBlink.c:45.5
```

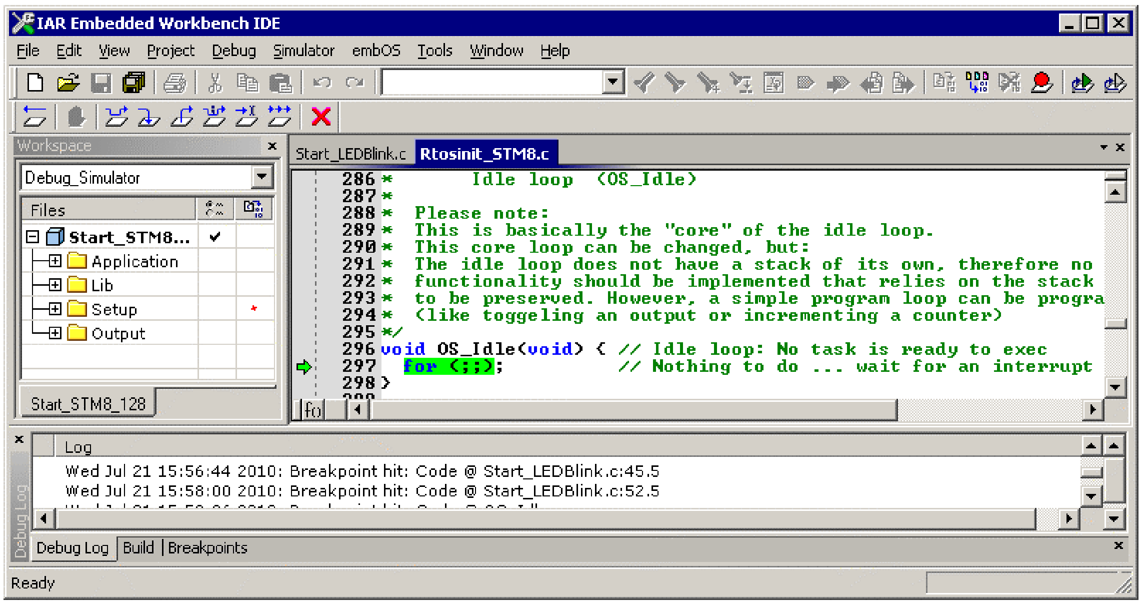
If you continue stepping, you will arrive in the task with the lower priority:



```
42 static void HPTask(void) {
43   while (1) {
44     BSP_ToggleLED(0);
45     OS_Delay (50);
46   }
47 }
48
49 static void LPTask(void) {
50   while (1) {
51     BSP_ToggleLED(1);
52     OS_Delay (200);
53   }
54 }
```

```
Wed Jul 21 15:56:44 2010: Breakpoint hit: Code @ Start_LEDBlink.c:45.5
Wed Jul 21 15:58:00 2010: Breakpoint hit: Code @ Start_LEDBlink.c:52.5
```

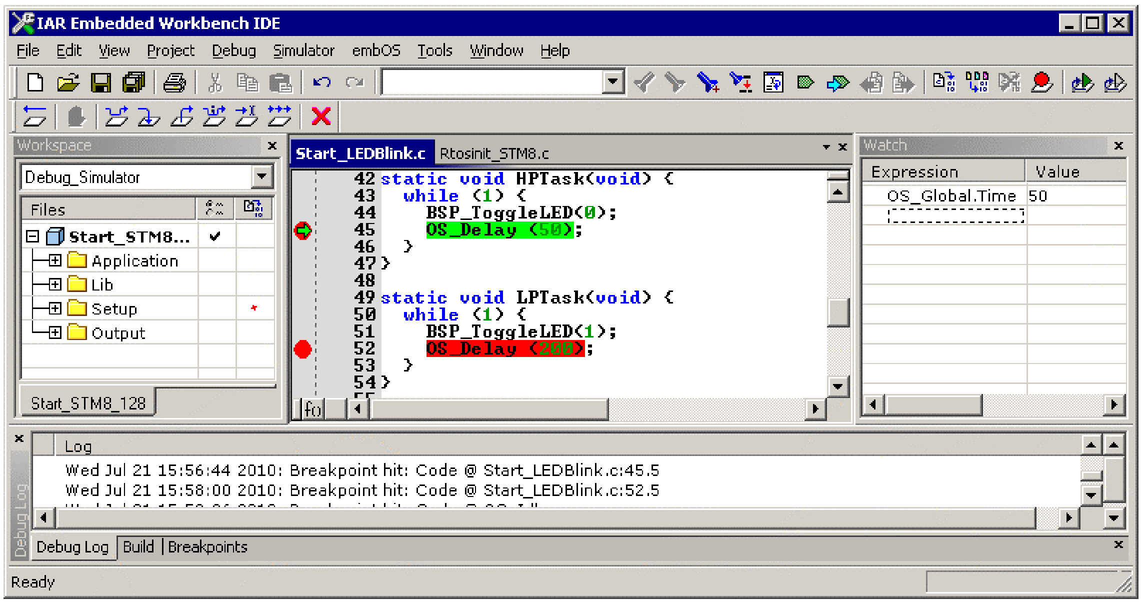
Continuing to step through the program, there is no other task ready for execution. ***embOS*** will suspend `LPTask()` and switch to the idle-loop, which is an endless loop which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).
`OS_Idle()`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



```
 *       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 have a stack of its own, therefore no
 *  functionality should be implemented that relies on the stack
 *  to be preserved. However, a simple program loop can be progra
 *  (like toggeling an output or incrementing a counter)
 */
void OS_Idle(void) {  // Idle loop: No task is ready to exec
  for (;;);           // Nothing to do ... wait for an interrupt
}
```

```
Log
Wed Jul 21 15:56:44 2010: Breakpoint hit: Code @ Start_LEDBlink.c:45.5
Wed Jul 21 15:58:00 2010: Breakpoint hit: Code @ Start_LEDBlink.c:52.5
```

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.
This can be seen at the system variable `OS_Global.Time`:



```
static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_Delay (50);
  }
}

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
  }
}
```

Expression	Value
OS_Global.Time	50

```
Log
Wed Jul 21 15:56:44 2010: Breakpoint hit: Code @ Start_LEDBlink.c:45.5
Wed Jul 21 15:58:00 2010: Breakpoint hit: Code @ Start_LEDBlink.c:52.5
```

3. Build your own application

To build your own application, you should start with the sample start project. This has the advantage, that all necessary files are included and all settings for the project are already done.

3.1. Required files for an *embOS* application

To build an application using ***embOS***, the following files from your ***embOS*** 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 a CPU specific Setup\ subfolder.
  It contains the hardware dependent initialization code for the ***embOS*** timer and optional UART for embOSView.
- One ***embOS*** **library** from the Lib\ subfolder.
- **OS_Error.c** from the CPU specific Setup\ folder.
  The error handler is used if any library other than a Release build library is used in your project.

When you decide to write your own startup code, ensure that non initialized variables are initialized with zero, according to "C" standard. This is required for some ***embOS*** internal variables.
Your `main()` function has to initialize ***embOS*** by calling `OS_InitKern()` and `OS_InitHW()` prior any other ***embOS*** functions except `OS_IncDI()` are called.

3.2. Select a start project

embOS comes with one start project for an STM8S CPU. The project includes different configurations for different output formats or debug tools. The start project for the STM8S CPU was built and tested with an STM8S208MB CPU and the STM8/128_EVAL board from ST. For other CPU variants there may be modifications required in the CPU specific RTOSInit*.c file.

3.3. Add your own code

For your own code, you may add a new group to the project.
You should then modify or replace the `main()` function in the start project.

3.4. Change the library mode

For your application you may wish to choose an other library. For debugging and program development you should use an ***embOS*** -debug library. For your final application you may wish to use an ***embOS*** -release library.
Therefore you have to select or replace the ***embOS*** library in your project or target:

- in the Lib group, exclude all libraries from build, except the one which should be used for your application.

Finally check the project options about library mode setting according library mode used, or modify the `OS_Config.h` file. Refer to chapter 4 about the library naming conventions to select the correct library and library mode specific define.

4. Project and compiler specifics

4.1. Code and data memory models, compiler options

embOS for STM8 and IAR compiler is delivered with libraries for all code models and one data model.

For STM8 CPUs, the IAR compiler offers three code models:

Code Model	Default code memory attribute	Code placement
Small	__near_func	0-0xFFFF
Medium	__far_func	0-0xFFFFFF
Large	__huge_func	0-0xFFFFFF

For STM8 CPUs, the IAR compiler offers three data models:

Data Model	Default memory attribute	Data placement
Small	__tiny	0-0xFF
Medium	__near	0-0xFFFF
Large	__far	0-0xFFFFFF

With ***embOS***, the following limitations exist:

- The small data model for STM8 is not supported, because it does not make much sense to run an RTOS application with 256 bytes of RAM.
- The large data model for STM8 is not supported.

4.2. Available libraries

The ***embOS*** library files are located in the subfolder 'Lib' of the "Start" folder.
To use ***embOS***, one library has to be included to your project. The files to use depend on additional error check possibilities wished to be used.
The naming convention for embOS libraries follows the one used by IAR for the runtime libraries:

osSTM8<code_model><data_model>_<LibMode>.a

Parameter	Meaning	Values
code_model	Selected code model	s: small code model
		m: medium code model
		l: large code model
data_model	Selected data model	m: medium data model
LibMode	Library mode	XR: Extreme Release
		R: Release
		S: Stack check
		D: Debug
		SP: Stack check + profiling
		DP: Debug + profiling
		DT: Debug + trace

Example:

osSTM8sm_SP.a is the ***embOS*** library for the small code model, medium data model and **S**tack check and **P**rofiling functionality.
It is located in the Start\Lib\ subdirectory.

For STM8 CPUs, the following ***embOS*** libraries are available:

Library type	Library	Code / data model	#define
Extreme Release	osSTM8sm_XR.a	small / medium	OS_LIBMODE_XR
Release	osSTM8sm_R.a	small / medium	OS_LIBMODE_R
Stack-check	osSTM8sm_S.a	small / medium	OS_LIBMODE_S
Stack-check+Profiling	osSTM8sm_SP.a	small / medium	OS_LIBMODE_SP
Debug	osSTM8sm_D.a	small / medium	OS_LIBMODE_D
Debug+Profiling	osSTM8sm_DP.a	small / medium	OS_LIBMODE_DP
Debug+Profiling +Trace	osSTM8sm_DT.a	small / medium	OS_LIBMODE_DT
Extreme Release	osSTM8mm_XR.a	medium / medium	OS_LIBMODE_XR
Release	osSTM8mm _R.a	medium / medium	OS_LIBMODE_R
Stack-check	osSTM8mm _S.a	medium / medium	OS_LIBMODE_S
Stack-check+Profiling	osSTM8mm _SP.a	medium / medium	OS_LIBMODE_SP
Debug	osSTM8mm _D.a	medium / medium	OS_LIBMODE_D
Debug+Profiling	osSTM8mm _DP.a	medium / medium	OS_LIBMODE_DP
Debug+Profiling +Trace	osSTM8mm _DT.a	medium / medium	OS_LIBMODE_DT
Extreme Release	osSTM8lm_XR.a	large / medium	OS_LIBMODE_XR
Release	osSTM8lm _R.a	large / medium	OS_LIBMODE_R
Stack-check	osSTM8lm _S.a	large / medium	OS_LIBMODE_S
Stack-check+Profiling	osSTM8lm _SP.a	large / medium	OS_LIBMODE_SP
Debug	osSTM8lm _D.a	large / medium	OS_LIBMODE_D
Debug+Profiling	osSTM8lm _DP.a	large / medium	OS_LIBMODE_DP
Debug+Profiling +Trace	osSTM8lm _DT.a	large / medium	OS_LIBMODE_DT

For STM8 CPUs, only the medium data model is available.

Ensure that the define, according to the library type used, is set as compiler option in your project, or is selected in the `OS_Config.h` file according the value of the DEBUG define.
Check “Project | Options | C/C++ compiler | Preprocessor | Defined symbols”.

5. Stacks

5.1. Task stack for STM8 CPUs

Every ***embOS*** task has to have its own stack. Task stacks can be located in any “near” RAM memory location.
The stack has to be addressable by the CPUs stack pointer which is 16bits wide, so the stack may reside in any RAM location between 0x0 to 0xFFFF.
The required task stack size is the sum of the stack size needed from all routines called by the task, plus a basic task stack size.
The basic task stack size is the size of memory required to store the registers and virtual registers of the CPU plus the stack size required by ***embOS*** -routines.
For the STM8 CPU, the minimum task stack size is about 30 bytes.
As the STM8 CPU does not support its own interrupt stack, interrupts also run on task stacks. We recommend at least a minimum task stack size of 100 bytes.
Using nested interrupts will increase the required stack size. You may use the embOSView tool to analyze the total amount of task stack used in your application.

5.2. System stack for STM8 CPUs

The system stack size required by ***embOS*** is about 30 bytes (60 bytes in. profiling builds) 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 and ***embOS*** internal scheduling functions also use the system-stack, as well as interrupts, which may also run on the system stack, the actual stack requirements depend on the application and interrupt handling.
The size of the system stack is given in the link-file or project options as size of the CSTACK.
We recommend a minimum of 128 bytes.

5.3. Interrupt stack for STM8 CPUs

Unfortunately the STM8 CPUs do not support a separate interrupt stack pointer. Interrupts use the stack of the running application. Therefore interrupts occupy additional stack space on every task and on the system stack. The current version of ***embOS*** does not support a separate interrupt stack.

5.4. Stack specifics of the STM8 family

The STM8 family of microcontroller can address up to 64KB of memory as stack. Because the stack-pointer register is 16bits wide.

6. Interrupts

6.1. What happens when an interrupt occurs?

- The CPU-core receives an interrupt request
- If the priority of the interrupt is higher than the current processor priority, the interrupt is accepted.
- The CPU saves the PC, all registers including flags on the stack
- The CPU sets its own interrupt priority in the condition code register to the priority of the requested interrupt, thus disabling all further interrupts of lower priority.
- The CPU jumps to the interrupt service routine (ISR) whose address is found in the in the vector table.
- ISR : Save scratch registers
- ISR : User-defined functionality
- ISR : Restore scratch Registers
- ISR: Execute RETI command, restoring the PC and all registers, Flags and continue interrupted program.
- For details, refer to the ST STM8 reference and programming manuals.

6.2. Defining interrupt handlers in "C"

Routines defined with the keyword `__interrupt` automatically save & restore the scratch registers and all registers they modify and return with RETI.
The interrupt vector number has to be given as additional parameter by a `#pragma` directive prior the interrupt handler function.
For a detailed description on how to define an interrupt routine in "C", refer to the IAR C/C++ Development guide for EWSTM8.

"Simple" interrupt-routine

```
#prgama vector=12
__interrupt void IntHandlerTimer(void) {
  IntCnt++;
}
```

6.3. Interrupt handling with ***embOS*** for STM8

Interrupt processing with ***embOS*** for STM8 requires some precautions and additional code in the interrupt handler functions.

6.3.1. Interrupt priorities with ***embOS*** for STM8

The current version 3.82h of ***embOS*** for STM8 does NOT support the priority controlled interrupt nesting of the STM8 CPUs.

- All interrupt handler functions have to run on the highest interrupt priority.
- The interrupt priority control registers for peripherals have to remain in their default state, or have to be initialized to an interrupt priority of 3, which is the highest priority.
- Interrupt nesting is therefore disabled per default.
  Re-Enabling interrupts inside an interrupt handler should be avoided and has to handled carefully.

6.3.2. Interrupt handler with *embOS* for STM8

Using ***embOS*** functions in an interrupt handler requires additional ***embOS*** functions which inform ***embOS*** that an interrupt handler is running and inform ***embOS*** when the interrupt handler ends.
This is required to avoid task switches from within an interrupt handler. Task switches can only be performed when the interrupt handler finished.
Therefore, every interrupt handler calling subroutines, or using ***embOS*** functions, has to start with a call of `OS_EnterInterrupt()` and has to end with a call of `OS_LeaveInterrupt()`.

Example of an interrupt handler calling embOS functions

```
#pragma vector=17
static __interrupt void OS_IsrTickHandler(void) {
  OS_TIM_SR1 &= ~(1uL << 0);  // Clear interrupt pending condition...
  OS_EnterInterrupt();        // Inform embOS that interrupt is running
  OS_HandleTick();
  OS_LeaveInterrupt();        // Inform embOS that interrupt ended, perform
}                             // task switch if required
```

`OS_EnterInterrupt()` informs ***embOS*** that an interrupt handler is running and blocks task switches.
`OS_LeaveInterrupt()` informs ***embOS*** that the interrupt handler ended and re-enables task switches. If a task switch is pending it is performed directly from `OS_LeaveInterrupt()`.
Note that most peripherals do not reset the interrupt pending condition automatically when the interrupt service routine is called. This has to be done during the execution of the interrupt service routine. To avoid loosing interrupts, this should be done as early as possible. Resetting the interrupt pending flag may be performed before the call of `OS_EnterInterrupt()`.

6.3.3. Nested Interrupt handler with *embOS* for STM8

As ***embOS*** version 3.82h for STM8 does not support the priority controlled interrupt nesting of the STM8 CPU, all interrupt handlers have to run on the highest interrupt priority level and automatic nesting by different priorities is not available.
However, interrupt nesting can be used if required, but has to be used very carefully, as all interrupts are re-enabled when interrupts are enabled in an interrupt service routine.
This may cause recursion which has to be avoided.
Writing a nestable interrupt handler using ***embOS*** requires a call of the ***embOS*** function `OS_EnterNestableInterrupt()` as entry function and has to end with the ***embOS*** function call `OS_LeaveNestableInterrupt()`.

Example of a nestable interrupt handler calling embOS functions

```
#pragma vector=17
static __interrupt void OS_IsrTickHandler(void) {
  OS_TIM_SR1 &= ~(1uL << 0);    // Clear interrupt pending condition...
  OS_EnterNestableInterrupt(); // Inform embOS that interrupt is running,
  OS_HandleTick();             // and re-enable interrupts
  OS_LeaveNestableInterrupt(); // Inform embOS that interrupt ended, perform
}                              // task switch if required
```

Warning: As `OS_EnterNestableInterrupt()` re-enables all interrupt levels, the interrupt pending condition has to be reset, or the specific interrupt source has to be disabled, before calling `OS_EnterNestableInterrupt()`.

`OS_EnterNestableInterrupt()` informs ***embOS*** that an interrupt handler is running and blocks task switches. Then it re-enable all interrupts.
`OS_LeaveNestableInterrupt()` informs ***embOS*** that the interrupt handler ended and re-enables task switches. If a task switch is pending it is performed directly from `OS_LeaveNetstableInterrupt()`.

6.4. Interrupt-stack

Since STM8 CPUs do not provide a separate stack pointer for interrupts, every interrupt occupies additional stack space on the current stack. This may be the system stack, or a task stack of a running task that is interrupted. The additional amount of necessary stack for all interrupts has to be reserved on all task stacks.
The current version of ***embOS*** for STM8 does not support extra interrupt stack-switching in an interrupt routine.
The routines `OS_EnterIntStack()` and `OS_LeaveIntStack()` are supplied for source compatibility to other processors only and have no functionality.

7. Low-Power Modes

Usage of Low-Power modes is one possibility 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 to enter a Low-Power mode.
Please do not enter a Low-Power mode which stops the ***embOS*** timer, as this would stop time scheduled task activations.

8. Technical data

8.1. Memory requirements

These values are neither precise nor guaranteed but they give a good idea of the memory-requirements. They vary depending on the current version of ***embOS***. The values in the table are for the medium memory model and release build library.

Short description	ROM [byte]	RAM [byte]
Kernel	approx.1800	42
Event-management	< 200	---
Mailbox management	< 550	---
Single-byte mailbox management	< 300	---
Resource-semaphore management	< 250	---
Timer-management	< 250	---
Add. Task	---	22
Add. Counting semaphore	---	4
Add. Resource semaphore	---	7
Add. Mailbox	---	11
Add. Timer	---	9
Power-management	---	---

9. Files shipped with *embOS* STM8 IAR

Directory	File	Explanation
root	`*.pdf`	Generic API and target specific documentation
root	`Release.html`	Release notes of ***embOS*** STM8
root	`embOSView.exe`	Utility for runtime analysis, described in generic documentation
Start\BoardSupport	`Start_*.eww`	CPU or eval board specific start workspace
Start\BoardSupport	`Start_*.ewp`	CPU or eval board specific start projects
Start\BoardSupport	`Start_*.ewd`	Setup for CSpy debugger
Start\BoardSupport	`RtosInit_*.c`	Target CPU specific hardware initialization; can be modified
Start\Inc\	`RTOS.h`	To be included in any file using ***embOS*** functions
Start\Lib\	`os*.a`	***embOS*** libraries
CPU\	`*.*`	CPU specific ***embOS*** sources, delivered with the source version only.
GenOsSrc\	`*.*`	***embOS*** generic sources, delivered with the source version only.

Other sample source files and additional documentation or tools may be delivered with ***embOS***.

10. Index