embOS

Real Time Operating System

Software Version 3.10d

CPU independent

User's & reference manual

Document revision 2

A product of SEGGER Microcontroller Systeme GmbH

Disclaimer

Copyright notice

Trademarks

Contact / registration

Please register the software. This way we can make sure you will receive updates or notifications of updates as soon as they become available. For registration please fill in and send us the registration card,

Contact address

SEGGER Microcontroller Systeme GmbH

Email : support@segger.com
Internet: http://www.segger.com/

Contents

1. About this document

This guide describes the functionality and user API of ***embOS*** Real Time Operating System.

1.1. Assumptions

This guide assumes that you already have a solid knowledge of the following:

- The software-tools used to build your application (assembler, linker, "C"-compiler)
- The C-language
- The target processor
- DOS-command-line

If you feel your knowledge of C is not good enough, we recommend *The C Programming Language* by Kernighan and Richie, which describes the standard in C-programming and in newer editions also covers ANSI C.

1.2. How to use this manual

This Manual explains all the functions and macros that ***embOS*** offers. However, it does cover the entire subject of real-time-programming. It assumes you have a working knowledge of the C-Language, knowledge of assembly programming is not required.
The intention of this manual is to give you a CPU & compiler independent introduction of ***embOS*** and to be a reference for all ***embOS*** API functions.
For a quick and easy startup with ***embOS***, please check out chapter 2 in the *CPU & Compiler Specifics* manual of ***embOS*** documentation, which includes a step-by-step introduction about using ***embOS***.

1.3. Typographic Conventions for Syntax

This manual uses the following typographic conventions for syntax:

Regular size Arial for normal text

Regular size courier for text that you enter at the command-prompt and for what you see on your display

`Regular size courier for RTOS-functions mentioned in the text`

```
Reduced size courier in a frame for
program examples
```

Boldface Arial for very important sections

Italic text for keywords

2. Introduction to ***embOS***

2.1. What is ***embOS*** ?

embOS is a priority-controlled Multitasking-System, designed to be used as embedded operating system for the development of real-time applications for a variety of microcontrollers.
embOS is a high performance tool that has been optimized for minimum memory consumption in both RAM and ROM, high speed and versatility.

2.2. Features

In the development process of ***embOS***, the limited resources of microcontrollers have always been kept in mind. The internal structure of the RTOS has been optimized in a variety of applications with different customers over a period of more than 5 years to fit the needs of the industry. Fully source-compatible RTOS are available for a variety of microcontrollers, making it an effort well worth the time to learn how to structure real-time programs with real-time-operating systems.
embOS is highly modular. This means that only those functions that are needed are linked, keeping the ROM-size very small. (Minimum is little more than1 kByte ROM and about 30 bytes of RAM (plus memory for stacks))
A couple of files are supplied in source-code-form to make sure that you do not lose any flexibility by using ***embOS*** and that you can customize the system to fully fit your needs.
The tasks that are created by the programmer can easily and safely communicate with each other using a complete palette of communication mechanisms like semaphores, mailboxes and events.

Some features of ***embOS*** are:

- Preemptive scheduling
  Guarantees that of all tasks in READY-state the one with the highest priority executes, except for situation where priority-inversion applies.
- Round robin scheduling for tasks with identical priorities
- Preemptions can be disabled for entire tasks or sections of a program
- up to 255 Priorities
  Every task can have an individual priority ⇒ The response of tasks can be precisely defined according to the requirements of the application
- Unlimited no. of tasks
  No. of tasks is limited by the amount of available memory only
- Unlimited no. of semaphores
  No. of semaphores is limited by the amount of available memory only
- 2 types of semaphores : Resource-, counting
- Unlimited no. of mailboxes
  No. of mailboxes is limited by the amount of available memory only
- Size and number of messages can be freely defined when initializing mailbox
- Unlimited no. of software-timers
  No. of software-timers is limited by the amount of available memory only
- 8-bit events for every task
- Time resolution can be freely selected (default 1ms)
- Easily accessible time variable
- Power management : Unused calculation-time can automatically be spent in halt-mode ⇒ power-consumption is minimized
- Full interrupt support
  Interrupts can call any function except those that require waiting for data or create, delete or change the priority of a task.
  Interrupts can wake-up or suspend tasks and directly communicate with tasks using all available communication-instances (mailboxes, semaphores, events)
- Very short interrupt-disable-time ⇒ short interrupt-latency-time
- Nested interrupts are permitted
- ***embOS*** has its own interrupt-stack, usage is optional
- Frame-application for easy start
- Debug-version performs run-time checks simplifying development
- Profiling and stack check may be implemented by choosing specified libraries.
- Monitoring during run time via UART available (embOSView).
- Very fast, efficient yet small code
- Minimum RAM usage
- Core written in assembly language
- Interfaces "C" and / or assembly
- Initialization of microcontroller hardware as sources

3. Basic concepts

3.1. Tasks

In this context, a task is a program running on the CPU-core of a microcontroller. Without a multitasking-kernel (without RTOS), only one task can be executed by the CPU. This is called a single-task-system. A real-time operating system allows execution of multiple tasks on a single CPU. All tasks execute as if they would completely "own" the entire CPU. The tasks are "scheduled"; the RTOS can activate and deactivate every task.

3.2. Multitasking: cooperative - preemptive

There are different ways the calculation-power of the CPU can be distributed among the tasks.

Cooperative Multitasking

This scheduling-system expects cooperation of all tasks. Tasks can only be suspended if they call a function of the operating system. If they do not, the system "hangs", meaning that the other tasks have no chance of being executed by the CPU.

Preemptive multitasking

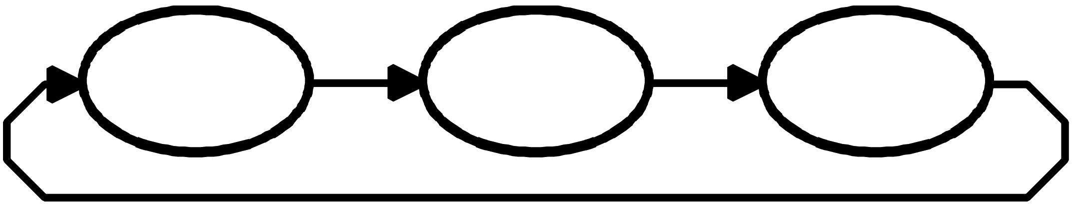
Real-time systems can be accomplished with preemptive multitasking only. A real-time operating system needs a regular timer-interrupt in order to be able to interrupt tasks at defined times and to perform task-switches if necessary.

3.3. Scheduling

There are different algorithms that determine which task to execute, called "scheduler". All schedulers have one thing in common:
They distinguish between tasks that are ready to be executed (In the READY state) and the other tasks, that are suspended for a reason (Delay, waiting for mailbox, waiting for semaphore, waiting for event etc.). The scheduler selects one of the tasks that are ready and activates it: It executes the program of this task.
This is what all schedulers have in common; the main difference is in how they distribute the computation time between the tasks in READY state.

Round-robin scheduling algorithm

In this case, the scheduler has a list of tasks and - when deactivating the active task - activates the next task that is in the READY state. Round-Robin works with either preemptive or cooperative multitasking. Round-Robin works well if you do not need to guarantee response-time and if the response time is not an issue of importance or if all tasks have the same priority. Round-robin scheduling can be symbolized as follows:



All tasks are on the same level, the possession of the CPU changes periodically after a predefined execution time. This time is called Timeslice and may be defined individually for every task.

Priority controlled scheduling algorithm

In real-world applications, the different tasks require different response times. For example in an application that controls a motor, the keyboard and a display, the motor usually requires faster reaction than keyboard and display. While the display is being updated, the motor needs to be controlled. This makes preemptive multitasking a must. Round-Robin might work, but since it can not guarantee a certain reaction time, an improved algorithm should be used: Every task is assigned a priority; the order of execution depends on this priority. The rule is very simple to put in words:
The Scheduler activates the task that has the highest priority of all tasks in READY-state.

This means that every time a task with higher priority than the active task gets ready, it immediately becomes the active task.

However, the scheduler can be switched off in sections of a program where task-switches are prohibited. (→ Critical region)

embOS uses a priority controlled scheduling algorithm with Round-Robin between tasks of identical priority. One hint at this point: Round-Robin scheduling is a nice feature because you do not have to think about which task is more important than an other one. Tasks with identical priority can not block each other for longer periods of time. But Round-Robin scheduling also costs time by constantly switching between tasks of identical priority if two or more tasks of identical priority are ready and no task of higher priority is ready. It is more efficient to assign different priorities to different tasks because this avoids unnecessary task switches.

Priority inversion

The rule to go by for the scheduler is:
Activate the task that has the highest priority of all tasks in READY-state

But what happens if the high-priority task is blocked because it is waiting for a resource owned by a low-priority task? According to the above rule, it would wait until the low-priority-task gets active again and releases the resource.

The other rule is: No rule without exception.

In order to avoid this kind of situation, the low-priority tasks that is blocking the high-priority task gets assigned the higher priority of the high-priority task until it releases the resource and it therefore no longer blocks the high-priority task.
This is known as priority inversion.

3.4. Communication between tasks

In a multi-task program (multithreaded program) multiple tasks work completely separated from each other. But since all of these tasks work in the same application, they probably have to communicate and exchange data or have to be synchronized. It also has to be made sure that resources are not used by different tasks at the same time.

Global variables

The easiest way to do this is to use global variables. In certain situations, it can make sense for tasks to communicate via global variables, but most of the time using global variables has various disadvantages.
For example if you want synchronize a task to start when the value of a global variable changes, you have to poll this variable, wasting precious calculation time & power, and your reaction time is depending on how often you poll.

Communication mechanisms

When multiple tasks work with one an other, a lot of times they have to

- exchange data,
- synchronize to an other task
- make sure that a resource is used by no more than one task at a time

For these purposes ***embOS*** offers mailboxes, semaphores and events.

Mailboxes

A mailbox is basically a data-buffer, that is managed by the RTOS and that works without conflicts and problems even if multiple tasks and interrupts try to access the mailbox simultaneously. ***embOS*** also automatically activates tasks that are waiting for a message in a mailbox the moment they receive new data and - if necessary - automatically switches to this task.

Semaphores

Two types of semaphores are used to synchronize tasks and to manage resource. Most commonly used are resource semaphores. For details and samples, check out the section on semaphores and look for samples on our website.

Events

A task can wait for a particular event without using any calculation time. The idea is as simple as convincing: There is no sense in polling if we can simply activate the task the moment the event that the task is waiting for occurs. This saves a lot of calculation power and makes sure the task can respond to the event without delay. Typical applications for events are where a task waits for data, a pressed key, a received command or character or the pulse of an external real-time clock.
For details, refer to the section → Events

3.5. How task-switching works

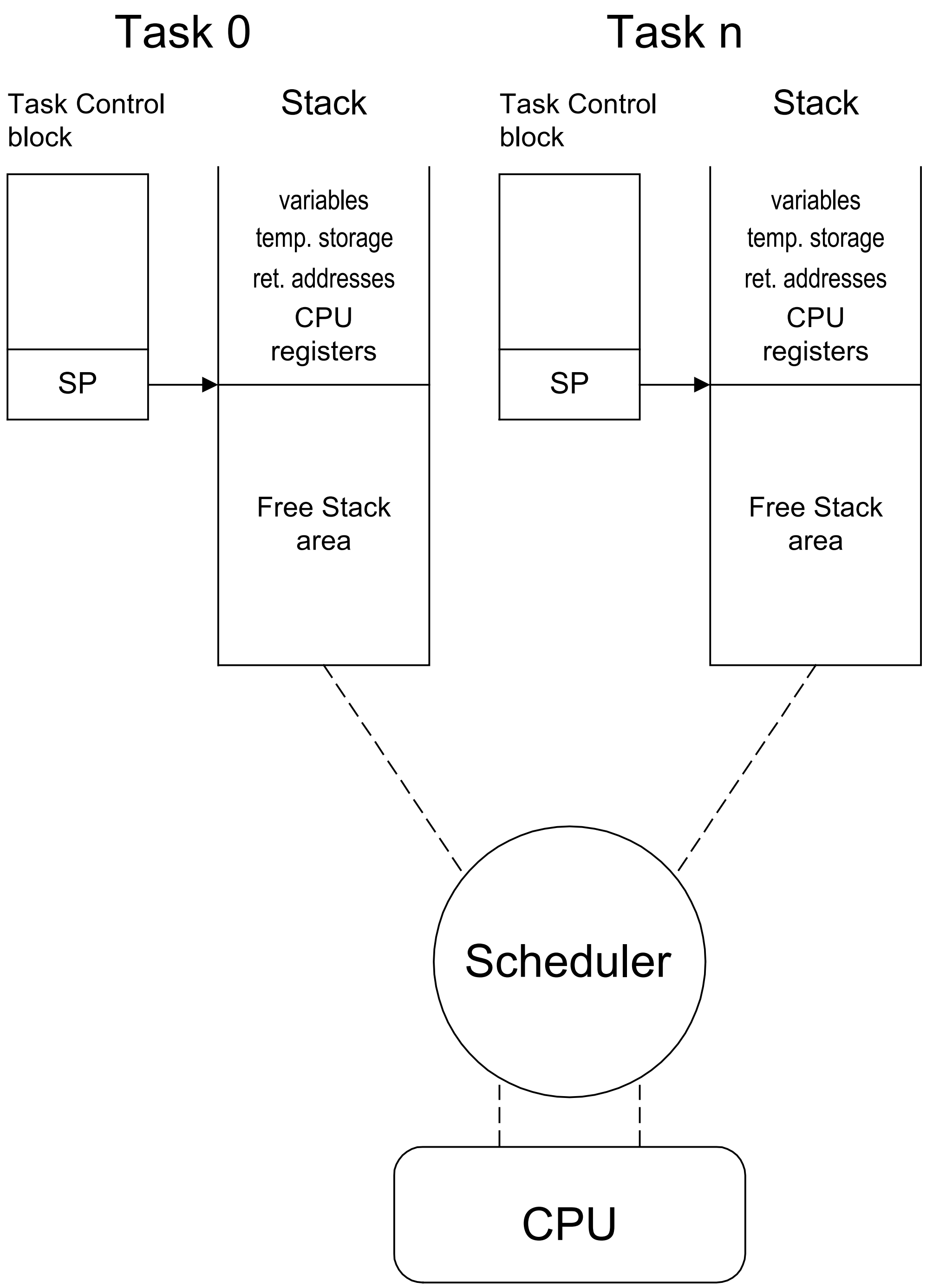
A real-time multitasking system lets multiple tasks run like multiple single-task-programs quasi-simultaneous on a single CPU.
A task consists of three parts in the multitasking-world:

- The program-code, which usually resides in ROM (though it does not have to!)
- A stack, residing in a RAM-area that can be accessed by the stack pointer
- A task-control-block, residing in RAM

The task-control-block (TCB) contains status information of the task: the stack-pointer, priority, current status (Ready, waiting and reason for suspension) and other management data. This TCB is accessed by the RTOS only.
The stack has the same function as in a single-task-system:
Storage of local variables, parameters, return addresses and temporary storage of intermediate calculation results and register values.

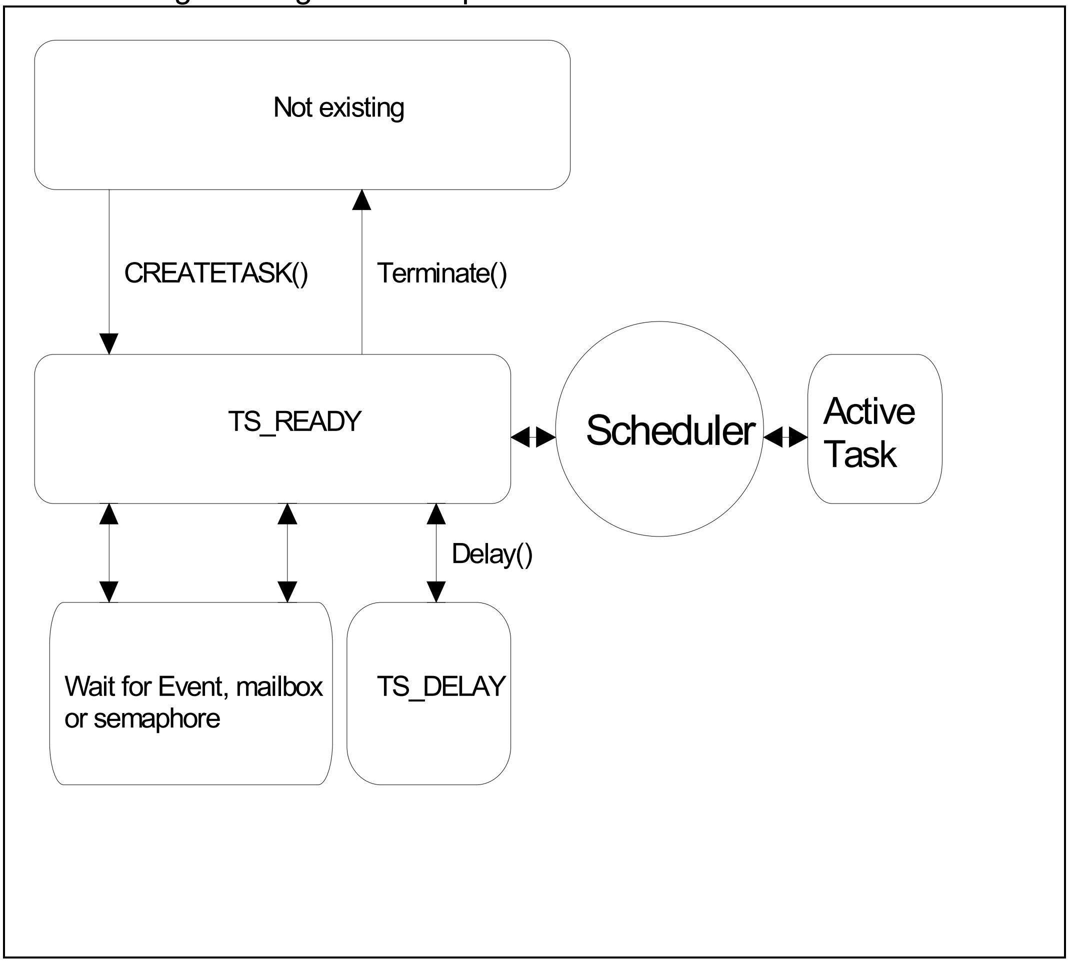
3.6. Switching stacks

The following little drawing demonstrates the process of switching from one stack to an other.
The scheduler deactivates the current task by saving the processor registers on the current stack.
It then selects the active task by loading the stack pointer and the processor-registers from the values stored on this stack.



3.7. Change of task status

When a task is created, it is automatically put in the READY state (TS_READY). As soon as there is no task with higher priority in the same state, this task is activated. This task will stay active until a task with higher priority becomes READY or the task is deactivated or it waits for a mailbox, semaphore, event or expiration of a delay.
The following drawing shows all possible task-states and the transitions.



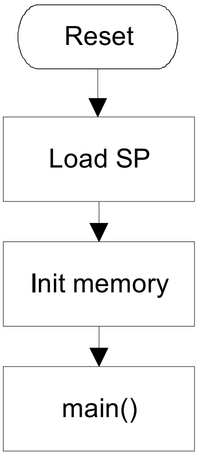
3.8. What happens after reset

On Reset, the special-function registers are set to their respective values.
After Reset, program execution starts.
The PC-register is set to the start address defined by the start-vector or start address (depending on CPU). This start address is usually in a Startup-module shipped with the C-compiler (and sometimes part of the standard library)

The startup code does the following:

- Load the SP (Stack-Pointer(s)) with the(ir) default values, which is (for most CPUs) the end of the defined stack-segment(s)
- Initialize all data segments to their respective value
- call "main" routine

The process can be shown as a flowchart as follows:



3.9. How the OS gains control

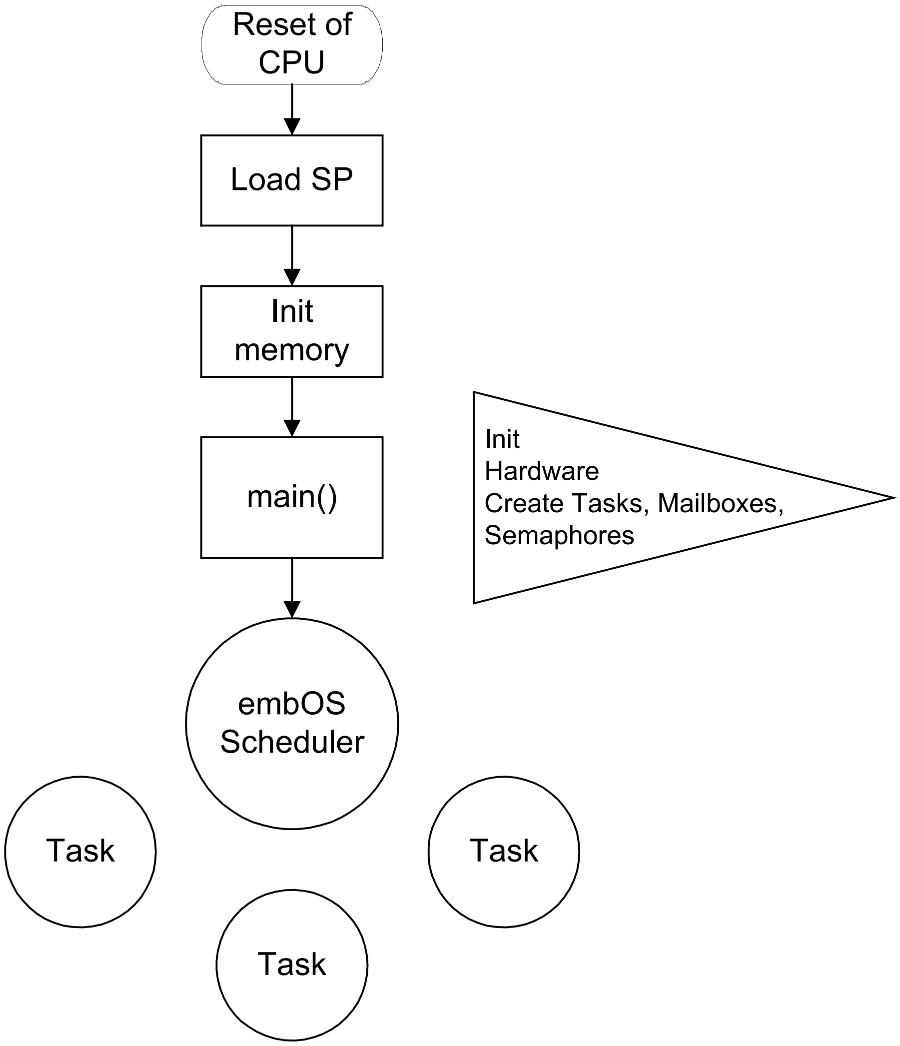
In a single-task-program, the `main` routine is part of the user-program which takes control right after the Cstartup.
Normally ***embOS*** works with the standard Cstartup-module without any change. If there are any changes required, those changes are documented in the startup file which is shipped with ***embOS***.
`main()` is still part of your application program. Basically main creates one or more tasks and then starts multitasking by calling `OS_Start()`. From here on, the scheduler controls which task is executed.
`main()` will not be interrupted by any of the created tasks, because these tasks are executed only after the call to `OS_Start()`. It is therefore usually good practice to create all or most of your tasks here, as well as control structures such as Mailboxes and Semaphores. A good practice is to write software in form of modules which are –up to a point - reusable. These modules usually have an initialization routine, which would create the task(s) and or control structures required for this module. A typical main() looks similar to the following example:

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

void main(void) {
  OS_InitKern();        /* initialize OS (should be first !)          */
  OS_InitHW();          /* initialize Hardware for OS (in RtosInit.c) */
  /* Call Init routines of all program modules which in turn will create
  the tasks they need ... (Order of creation may be important) */
  MODULE1_Init();
  MODULE2_Init();
  MODULE3_Init();
  MODULE4_Init();
  MODULE5_Init();
  OS_Start();           /* Start multitasking */
}
```

With the call to `OS_Start()`, the scheduler starts the highest-priority task.
Please note, that `OS_Start()` does not return.

The following flowchart illustrates the starting procedure:



3.10. Different builds of ***embOS***

embOS comes in different builds (Different versions of the libraries). The reason for different builds is that requirements vary during development. While developing software, the performance (and resource usage) is not as important as in the final version which usually goes as release version into the product. But during development even small programming errors should be caught by use of assertions. These assertions are compiled into the debug version of the ***embOS*** libraries and make the code a bit bigger (about 50%) and also slightly slower than the release or stack check version used for the final product. This concept gives you the best of both worlds: A compact and very efficient build for your final product (release or stack check versions of the libraries) and a safer, but slower and bigger version for development which will catch most of the common programming errors. Of course you may also use the release version of ***embOS*** during development, but it will not catch these application programming errors.

3.10.1. Profiling

embOS supports profiling in profiling builds. Profiling makes precise information available about the execution time of individual tasks.
You may always use the profiling libraries, but they induce certain overhead (Bigger task control blocks, add. ROM (app. 200 bytes) and add. run time overhead). This overhead is usually acceptable, but for best performance you may want to use non-profiling builds of ***embOS*** if you do not use this feature.

3.10.2. List of libraries

In your application program, you need to let the compiler know which build of ***embOS*** you are using. This is done by defining a single identifier prior to including RTOS.h.

Build	Define	Explanation
R: Release	OS_LIBMODE_R	Smallest, fastest build
S: Stack check	OS_LIBMODE_S	Same as release, plus stack checking
SP: Stack check plus Profiling	OS_LIBMODE_SP	Same as stack checking plus profiling
D: Debug	OS_LIBMODE_D	Maximum run-time checking
DP: Debug plus Profiling	OS_LIBMODE_DP	Maximum run-time checking plus Profiling
DT: Trace, including Debug, Profiling	OS_LIBMODE_DT	Tracing API calls, maximum run-time checking plus Profiling

4. Configuration for your target system (RTOSINIT.c)

You do not have to configure anything in order to get started with ***embOS.*** The start project supplied will execute on your system. Small changes in the configuration will be necessary at a later point for system frequency or for the UART used for communication with embOSView (optional).
The file RTOSINIT.c is provided in source-code form and can be modified in order to match your target-hardware needs. You compile and link it with your application program.

4.1. Routines in RTOSInit.c

	Explanation
OS_InitHW()	***embOS*** needs a timer-interrupt to determine when to activate tasks that wait for the expiration of a delay, when to call a software-timer and to keep the time-variable up to date. The hardware timer that needs to be initialized for a small program with ***embOS*** is initialized in the function OS_InitHW().
OS_Error()	Is called by ***embOS*** when a fatal error has been detected
OS_Idle()	The idle loop is always executed whenever no other task (and no interrupt service routine) is ready for execution.
OS_GetTime_Cycles()	Reads the timestamp in cycles. Cyclelength depends on the system. This function is used for system information sent to embOSView.
OS_ConvertCycles2us()	Converts Cycles into us. (Used with profiling only)
OS_COM_Init()	Initializes communication for embOSView (Used with embOSView only)
OS_ISR_Tick()	The ***embOS*** timer interrupt handler. When using a different timer, always check the specified interrupt vector
OS_ISR_rx()	Rx Interrupt service handler for embOSView (Used with embOSView only)
OS_ISR_rx()	Tx Interrupt service handler for embOSView (Used with embOSView only)
OS_COM_Send1(…)	Send 1 byte via UART (Used with embOSView only, DO NOT call this function from your application)

4.2. Configuration defines

For most embedded systems, configuration is done by simply changing the following defines:

define	Explanation
OS_FSYS	System Frequency (in Hz) Example: 20000000 for 20MHz
OS_UART	Selection of UART to be used for embOSView -1 will disable communication
OS_BAUDRATE	Selection of baudrate for communication with embOSView

4.3. How to change settings

The only file which needs to be changed is RTOSInit.c, This file contains all hardware specific routines. There is only one exception: Some ports of ***embOS*** require an additional interrupt vector table file.

4.3.1. Setting the system frequency OS_FSYS

Relevant defines

OS_FSYS

Relevant routines

OS_ConvertCycles2us() (Only for profiling)

For most systems it should be sufficient to change the OS_FSYS define at the top of RTOSINIT.c. When using profiling, certain values may require a change in `OS_ConvertCycles2us()`. Please check out the contents of RTOSINIT.c for more detailed information about in which cases this is necessary and what needs to be done.

4.3.2. Using a different timer to generate the tick-interrupts for ***embOS***

Relevant routines:

OS_ InitHW()

4.3.3. Using a different UART or baudrate for embOSView

Relevant defines

OS_UART
OS_BAUDRATE

Relevant routines:

OS_COM_Init()
OS_COM_Send1()
OS_ISR_rx()
OS_ISR_tx()

In some cases, this is done by simply changing the define OS_UART on top of the RTOSInit.c. Please check out the contents of this file for more detailed information on which UARTS are supported for your CPU.

4.3.4. Changing the tick frequency

embOS usually generates 1 interrupt per ms. This is done by a timer initialized in `OS_InitHW()`.

OS_FSYS defines the clock frequency of your system in Hz.

The value of OS_FSYS is taken to calculate the desired reload counter value for the system timer for 1000 interrupts/sec.
The timer itself is initialized in the routine `OS_InitHW()`, which is found in `RTOSINIT.C`. If you have to use a different timer for your application, you must modify `OS_InitHW()` to initialize the appropriate timer. For details about initialization, please read the comments in RTOSInit.c.

However, different (lower or higher) interrupt-rates are possible.
If you chose an interrupt-frequency different from 1kHz, the value of the time variable OS_Time will no longer be equivalent to multiples of 1 ms.
However, if you use a multiple of 1 ms as tick time, the basic time unit can be made 1 ms by using the (optional) configuration macro OS_CONFIG(..).
The basic time unit does not have to be 1 ms, it might just as well be 100us or 10 ms or any other value. For most applications 1 ms is a convenient value.
For details, refer to → OS_CONFIG.

4.4. OS_CONFIG

OS_CONFIG can be used to configure ***embOS*** in situations, where the basic timer interrupt interval is a multiple of 1ms and the time values for delays still should use 1 ms as time base.
OS_CONFIG tells ***embOS*** how many clock ticks expire per ***embOS*** -timer interrupt and what the system-frequency is.

<u>Examples for OS_CONFIG</u>

1) The following will lead to increment the time variable OS_Time by 1 per RTOS-timer-interrupt:

```
OS_CONFIG(8000000,8000);      /* Configure OS : System-frequency, ticks/int */
```

As this is the default for ***embOS***, usage of OS_CONFIG is not required.

2) The following will lead to increment the time variable OS_Time by 2 per ***embOS*** -timer-interrupt.

```
OS_CONFIG(8000000,16000);     /* Configure OS : System-frequency, ticks/int */
```

If for example the basic timer was initialized to 500Hz, which would result in an ***embOS*** timer interrupt every 2ms, a call of `OS_Delay(10)` would result in a delay of 20ms, because all timing values are interpreted as timer ticks. A call of OS_CONFIG with the parameter shown in example 2 will then result in a delay of 10ms when calling `OS_Delay(10)`.

5. Task routines

A task that should run under ***embOS*** needs a task control block, a stack and just a normal routine, written in C. The following rules apply to task routines:

- The task routine can not take parameters
- The task routine is never called directly from your application
- The task routine does not return
- The task routine should be implemented as endless loop, or has to terminate itself.
- The task routine is started from the scheduler, after the task was created and OS_Start() was called.

```
/*  Example of a task routine as endless loop    */
void Task1(void) {
  while(1) {
    DoSomething()               /* Do something  */
    OS_Delay(1);    /* Give other tasks a chance */
  }
}
```

```
/*  Example of a task routine that terminates       */
void Task2(void) {
  char DoSomeMore;
  do {
    DoSomeMore = DoSomethingElse() /* Do something  */
    OS_Delay(1);       /* Give other tasks a chance */
  } while(DoSomeMore);
  OS_Terminate(0);     /* Terminate yourself        */
}
```

There are different ways to create a task: ***embOS*** offers a simple macro that makes it easy to create a task and is fully sufficient in most cases. However, if you are dynamically creating and deleting tasks, a routine is available allowing "fine-tuning" of all parameters. For most applications, at least initially, using the macro as in the sample start project works fine.

5.1. OS_CREATETASK

Description

Creates a task.

Prototype

```
void  OS_CREATETASK(OS_TASK* pTask,
                    char*    pName,
                    void*    pRoutine,
                    char     Priority,
                    void*    pStack);
```

Parameter	Meaning
pTask	Pointer to a data structure of type OS_TASK which will be used as task control block (and reference) for this task.
pName	Pointer to the Name of the task. Can be NULL (or 0) if not used.
pRoutine	Pointer to a routine that should run as task
Priority	Priority of the task. Has to be in the range : 0< Priority <=255 Higher values indicate higher priorities.
pStack	Pointer to an area of memory in RAM that will serve as stack area for the task. The size of this block of memory determines the size of the stack-area for this task.

Return value

Void.

Add. information

`OS_CREATETASK` is a macro calling an OS -library function.
It creates a task and makes it ready for execution by putting it in the READY state.
The newly created task will be activated by the scheduler as soon as there is no other task with higher priority in READY state. (→Scheduler)
If there is an other task with the same priority, the new task will be put right before that.
`OS_CREATETASK` can be called at any time, either from main during initialization, or from any other task.
The recommended strategy is to create all tasks during initialization in main in order to keep the structure of your tasks easy to understand.
This macro is normally used to create a task instead of the function call below, because it has less parameters and is therefore easier to use.
The absolute value of the `Priority` is of no importance, only the value in comparison to the priorities of other tasks.
The macro OS_CREATETASK determines the size of the stack automatically using *sizeof.* This is possible only if the memory area has been defined at compile-time.

Important:

The stack that you define has to reside in an area that the CPU can actually use as stack, since the CPU can not use the entire memory-area as stack.

Example

```
char UserStack[150];    /* Stack-space */
OS_TASK UserTCB;        /* Task-control-blocks */

void UserTask(void) {
  while (1) {
    Delay (100);
  }
}

void InitTask(void) {
  OS_CREATETASK(&UserTCB, "UserTask", UserTask, 100, UserStack);  /* Create
Task0 */
}
```

5.2. OS_CreateTask

Description

Creates a task.

Prototype

```
void OS_CreateTask (OS_TASK*      pTask,
                    char*         pName,
                    unsigned char Priority,
                    voidRoutine*  pRoutine,
                    void*         pStack,
                    unsigned      StackSize,
                    unsigned      TimeSlice);
```

Parameter	Meaning
`pTask`	Pointer to a data structure of type OS_TASK which will be used as task control block (and reference) for this task.
`pName`	Pointer to the Name of the task. Can be NULL if not used.
`Priority`	Priority of the task. Has to be in the range : 0< Priority <=255 Higher values indicate higher priorities.
`pRoutine`	Pointer to a routine that should run as task
`pStack`	Pointer to an area of memory in RAM that will serve as stack area for the task. The size of this block of memory determines the size of the stack-area for this task.
`StackSize`	Size of the stack
`TimeSlice`	Time slice value for round robin scheduling. Has an effect only if other tasks are running at the same priority. TimeSlice denotes the time in timer ticks, that the task will run until it suspends; thus enabling an other task with the same priority. This parameter has no effect for some ports of ***embOS*** for efficiency reasons.

Return value

Void.

Add. information

Creates a task. All parameters of the task can be specified. The task can be dynamically created because the stack size is not calculated automatically. Works the same way as described under OS_CREATETASK.

Important:

The stack that you define has to reside in an area that the CPU can actually use as stack. Most CPUs can not use the entire memory-area as stack.

Example

```
/*
*  demo-program to illustrate the use of OS_CreateTask
*/
char StackMain[100], StackClock[50];
OS_TASK TaskMain,TaskClock;
OS_SEMA SemaLCD;

void Clock(void) {
  while(1) {
    /* code to update the clock */
  }
}

void Main(void) {
  while (1) {
    /* your code  */
  }
}

void InitTask(void) {
  OS_CreateTask(&TaskMain,  NULL, 50, Main,  StackMain, sizeof(StackMain), 2);
  OS_CreateTask(&TaskClock, NULL, 100, Clock,StackClock,sizeof(StackClock),2);
}
```

5.3. OS_Delay: Suspend for fixed time

Description

The calling task will be put to the TS_DELAY-state for a period of time.

Prototype

```
void OS_Delay(int ms);
```

Parameter	Meaning
ms	Time interval to delay. Has to be in the following range : $0 < ms < 2^{15}-1$ = 0x7FFF = 32767 for 8/16 bit CPUs $0 < ms < 2^{31}-1$ = 0x7FFFFFFF for 32 bit CPUs

Return value

Void.

Add. information

By calling the delay-routine, the task will stay in this state until the time specified has expired.
`ms` specifies the precise interval during which the task has to be suspended given in basic time intervals (usually 1/1000 sec). The actual delay (in basic time intervals) will be in the following range :

ms-1 <= Delay <= ms
depending on when the Interrupt for the Scheduler will occur.

After the expiration of a delay, the task is made ready again and activated according to the rules of the scheduler.
A delay can be ended prematurely by an other task or an interrupt-handler calling `OS_WakeTask`.

Example

```
void Hello() {
  printf("Hello");
  printf("The next line will be executed in 5 seconds");
  OS_Delay (5000);
  printf("Delay is over");
}
```

5.4. OS_DelayUntil: Suspend until

Description

Similar to the `Delay`-*routine*.

Prototype

`void OS_DelayUntil(int t);`

Parameter	Meaning
`t`	Time to delay until. Has to be in the following range : $0 < t\text{-}OS\_Time < 2^{15}-1$ 0x7FFF = 32767 for 8/16 bit CPUs $0 < t\text{-}OS\_Time < 2^{31}-1$ 0x7FFFFFFF for 32 bit CPUs

Return value

Void.

Add. information

`OS_DelayUntil` delays until the value of the time-variable `OS_Time` has reached a certain value. It is very useful if you have to avoid accumulating delays.

Example

```
int sec,min;

void TaskShowTime() {
  int t0 = TimeMS;
  while (1) {
    ShowTime();                    /* Routine to display time */
    OS_DelayUntil (t0+=1000);
    if (sec<59) sec++;
    else {
      sec=0;
      min++;
    }
  }
}
```

In the example above, the use of `OS_Delay` could lead to accumulating delays and would cause the simple "clock" to be slow.

5.5. OS_SetPriority: Change priority of a task

Description

Assigns the Priority specified by `Priority` to the specified task.

Prototype

```
void OS_SetPriority(OS_TASK * pt, char Priority);
```

Parameter	Meaning
`Pt`	Pointer to a data structure of type OS_TASK
`Priority`	Priority of the task. Has to be in the range : 0< Priority <=255

Return value

Void.

Add. information

Can be called at anytime from any task or software-timer. Calling this function might lead to an immediate task-switch.

Important:

This function may not be called from within an interrupt-handler.

5.6. OS_GetPriority: Retrieve priority of a task

Description

Returns the priority of a specified task.

Prototype

```
unsigned char OS_GetPriority(OS_TASK* pt);
```

Parameter	Meaning
`pt`	Pointer to a data structure of type OS_TASK If pt is the NULL pointer, the function returns the priority of the current running task.

Return value

Priority of specified task as unsigned char.
range 1 .. 255

Add. information

If pt does not specify a valid task, the debug version of ***embOS*** calls `OS_Error()`.
The release version of ***embOS*** can not check validity of pt and may therefore return invalid values if pt does not specify a valid task.

5.7. OS_SetTimeSlice: Change timeslice of a task

Description

Assigns the Timeslice value specified by `TimeSlice` to the specified task.

Prototype

```
unsigned char OS_SetTimeSlice(OS_TASK * pt,
                              unsigned char TimeSlice);
```

Parameter	Meaning
pt	Pointer to a data structure of type OS_TASK
TimeSlice	New timeslice value for the task Has to be in the range : 1<= TimeSlice <=255

Return value

unsigned char: Previous timeslice value of the task.

Add. information

Can be called at any time from any task or software timer. Setting the timeslice value only affects on the tasks running in round robin mode. This means, an other task with the same priority must exist. The new timeslice value is interpreted as reload value. It is used after the next activation of the task. It does not affect the remaining timeslice of a running task.

5.8. OS_Terminate: Terminate a task

Description

Ends a task.

Prototype

```
void OS_Terminate(OS_TASK* pTask);
```

Parameter	Meaning
pTask	Pointer to a data structure of type OS_TASK used for the task that shall be terminated. If `pTask` is the NULL pointer, the current task terminates.

Return value

Void.

Add. information

It should be made sure that the task does not use any resources at that point. The specified task will terminate immediately; the memory used for stack and task-control-block can be reassigned.

Important:

This function may not be called from within an interrupt-handler.

5.9. OS_WakeTask

Description

End Delay of a task immediately.

Prototype

```
void OS_WakeTask(OS_TASK* pTask);
```

Parameter	Meaning
`pTask`	Pointer to a data structure of type OS_TASK which will be used as task control block (and reference) for this task.

Return value

Void.

Add. information

Puts the specified task, that has been suspended for a certain amount of time with `OS_Delay` or `OS_DelayUntil` and is therefore in the state *TS_DELAY,* back to the state *TS_READY* (ready for execution). The specified task will be activated immediately if it has a higher priority than the priority of the task that had the highest priority before. If the specified task is not in the state *TS_DELAY* (because it has already been activated or the delay has already expired or for some other reason), the command is ignored.

5.10. OS_IsTask

Description

Checks whether a task control block actually belongs to a valid task.

Prototype

```
char OS_IsTask(OS_TASK* pTask);
```

Parameter	Meaning
pTask	Pointer to a data structure of type OS_TASK which will be used as task control block (and reference) for a task.

Return value

character value
0: TCB actually not used by any task
1: TCB is used by a task.

Add. information

This function checks, if the requested task is still in the internal task list. If the task was terminated, it is removed from the internal task list. This function may be useful to check, whether the task control block and stack for the task may be reused for an other task in applications that create and terminate tasks dynamically.

5.11. OS_GetTaskID

Description

Returns the ID of the task that is actually running.

Prototype

```
OS_TASKID OS_GetTaskID(void);
```

Return value

OS_TASKID: A pointer to the task control block. A value of 0 (NULL) indicates, that no task is executing.

Add. information

This function may be used to check, which task is executing. This may be helpful, if reaction of any function depends on actual running task.

5.12. OS_GetpCurrentTask

Description

Returns a pointer of the task control block structure of the running task.

Prototype

```
OS_TASK* OS_GetpCurrentTask(void);
```

Return value

OS_TASK*: A pointer to the task control block of the running task.

Add. information

This function may be used to check which task is executing. This may be helpful, if reaction of any function depends on actual running task.

6. Software Timer

A basically unlimited number of software-timers can be defined. A software-timer is an object defined with OS_CREATETIMER. A timer calls a user-specified routine after a specified delay.
Timers can be stopped, started and retriggered very similar to hardware timers. When defining the timer, you specify any routine that is to be called after the expiration of the delay that you specify. Timer routines are similar to interrupt routines; they have a priority higher than the priority of all tasks. For that reason they should be kept short just like interrupt routines.
Software-timers are called by ***embOS*** with interrupts enabled, so they can be interrupted by any hardware interrupt.
Generally timers run in single-shot-mode, which means, they expire only once and call their callback routine only once. By calling `OS_RetriggerTimer()` from within the callback-routine, the timer is restarted with its initial delay time and therefore works just as a free running timer.
The state of timers can be checked by the functions `OS_GetTimerStatus()`, `OS_GetTimerValue()` and `OS_GetTimerPeriod()`

6.1. OS_CREATETIMER

Description

A macro that creates and starts a software-timer.

Prototype

```
void OS_CREATETIMER(OS_TIMER*        pTimer,
                    OS_TIMERROUTINE* Callback,
                    unsigned int Timeout);
```

Parameter	Meaning
pTimer	Pointer to the OS_TIMER data structure containing the data of the timer
Callback	Pointer to the callback routine to be called from RTOS after expiration of the delay
Timeout	Initial timeout in basic embOS time units (nominal ms). Minimum 1 Maximum 32767

Return value

Void.

Add. information

The timers are being kept track of in the form of a linked list that is managed by ***embOS.*** Once the timeout is expired, the callback routine will be called immediately (unless the task is in a critical region or has interrupts disabled!).
This macro uses the functions `OS_CreateTimer()` and `OS_StartTimer()`. It is supplied for backward compatibility; In newer programs these routines should be called directly instead.
`OS_TIMERROUTINE` is defined in Rtos.h:

```
typedef void OS_TIMERROUTINE(void);
```

Source of the macro (in RTOS.h)

```
#define OS_CREATETIMER(pTimer,c,d)  \
        OS_CreateTimer(pTimer,c,d); \
        OS_StartTimer(pTimer);
```

Example

```
OS_TIMER TIMER100;

void Timer100(void) {
  LED = LED ? 0 : 1;             /* toggle LED  */
  OS_RetriggerTimer(&TIMER100);  /* make timer periodical */
}

void InitTask(void) {
  /* Create and start Timer100  */
  OS_CREATETIMER(&TIMER100, Timer100, 100);
}
```

6.2. OS_CreateTimer

Description

Creates a software-timer. (But does not start it)

Prototype

```
void OS_CreateTimer(OS_TIMER*        pTimer,
                    OS_TIMERROUTINE* Callback,
                    unsigned int Timeout);
```

Parameter	Meaning
pTimer	Pointer to the OS_TIMER data structure containing the data of the timer
Callback	Pointer to the callback routine to be called from RTOS after expiration of the delay
Timeout	Initial Timeout in basic embOS time units (nominal ms). Minimum 1 Maximum 32767

Return value

Void.

Add. information

The timers are being kept track of in the form of a linked list that is managed by ***embOS.*** Once the timeout is expired, the callback routine will be called immediately (unless the task is in a critical region or has interrupts disabled!).
The timer is not automatically started. This has to be done explicitly by a call of `OS_StartTimer()` or `OS_RetriggerTimer()`.
`OS_TIMERROUTINE` is defined in Rtos.h:

```
typedef void OS_TIMERROUTINE(void);
```

Example

```
OS_TIMER TIMER100;

void Timer100(void) {
  LED = LED ? 0 : 1;             /* toggle LED  */
  OS_RetriggerTimer(&TIMER100);  /* make timer periodical */
}

void InitTask(void) {
  /* Create Timer100, start it elsewhere */
  OS_CreateTimer(&TIMER100, Timer100, 100);
}
```

6.3. OS_StartTimer

Description

Starts the specified timer.

Prototype

```
void OS_StartTimer(OS_TIMER* pTimer);
```

Parameter	Meaning
`pTimer`	Pointer to the `OS_TIMER` data structure containing the data of the timer

Return value

Void.

Add. information

`OS_StartTimer()` is used for the following reasons:

- Start a timer which was created by `OS_CreateTimer()`. The timer will start with its initial timer value.
- Restart a timer which was stopped by calling `OS_StopTimer()`. In this case, the timer will continue with the remaining time value, which was preserved by stopping the timer.

This function has no affect on running timers.
Also this function has no effect on timers that are not running, but are expired. Use OS_RetriggerTimer() to rerstart those timers.

6.4. OS_StopTimer

Description

Stops the specified timer.

Prototype :

```
void OS_StopTimer(OS_TIMER* pTimer);
```

Parameter	Meaning
`pTimer`	Pointer to the `OS_TIMER` data structure containing the data of the timer

Return Value

Void

Add. information

The actual value of the timer (the time until expiration) is kept until `OS_StartTimer()` lets the timer continue.

6.5. OS_RetriggerTimer

Description

Restarts the specified timer with its initial time value.

Prototype

```
void OS_RetriggerTimer(OS_TIMER* pTimer);
```

Parameter	Meaning
pTimer	Pointer to the `OS_TIMER` data structure containing the data of the timer

Return value

Void.

Add. information

`OS_RetriggerTimer()` restarts the timer using the initial time value programmed at creation of the timer.

Example

```
OS_TIMER TIMERCursor;
BOOL CursorOn;

void TimerCursor(void) {
  if (CursorOn) ToggleCursor();    /* invert character at cursor-position */
  OS_RetriggerTimer(&TIMERCursor); /* make timer periodical */
}

void InitTask(void) {
  /* Create and start TimerCursor  */
  OS_CREATETIMER(&TIMERCursor, TimerCursor, 500);
}
```

6.6. OS_SetTimerPeriod

Description

Sets a new timer reload value for the specified timer.

Prototype

```
void OS_SetTimerPeriod(OS_TIMER* pTimer,
                               unsigned int Period);
```

Parameter	Meaning
pTimer	Pointer to the `OS_TIMER` data structure containing the data of the timer
Period	Timer period in basic embOS time units (nominal ms). (1 <= Delay <= 32767)

Return value

Void.

Add. information

`OS_SetTimerPeriod()` sets the initial time value of the specified timer. The period is the reload value of the timer, which is set as initial value, when the timer is retriggered by OS_RetriggerTimer().

Example

```
OS_TIMER TIMERPulse;
BOOL CursorOn;

void TimerPulse(void) {
  if TogglePulseOutput();          /* Toggle output */
  OS_RetriggerTimer(&TIMERCursor); /* make timer periodical */
}

void InitTask(void) {
  /* Create and start Pulse Timer with first pulse = 500ms */
  OS_CREATETIMER(&TIMERPulse, TimerPulse, 500);
  /* Set timer period to 200 ms for further pulses */
  OS_SetTimerPeriod(&TIMERPulse, 200);
}
```

6.7. OS_DeleteTimer

Description

Stops and deletes the specified timer.

Prototype :

```
void OS_DeleteTimer(OS_TIMER* pTimer);
```

Parameter	Meaning
pTimer	Pointer to the `OS_TIMER` data structure containing the data of the timer

Return Value

Void

Add. information

The timer is stopped and therefore removed out of the linked list of running timers. In debug builds of ***embOS*** the timer is also marked as invalid.

6.8. OS_GetTimerPeriod

Description

Returns the actual reload value of the specified timer.

Prototype

```
unsigned int OS_GetTimerPeriod(OS_TIMER* pTimer);
```

Parameter	Meaning
`pTimer`	Pointer to the `OS_TIMER` data structure containing the data of the timer

Return value

Unsigned integer between 1 and 32767, which is the allowed range of timer values.

Add. information

The period is the reload value of the timer, which is used as initial value, when the timer is retriggered by `OS_RetriggerTimer()`.

6.9. OS_GetTimerValue

Description

Returns the actual remaining timer value of the specified timer.

Prototype

```
unsigned int OS_GetTimerValue(OS_TIMER* pTimer);
```

Parameter	Meaning
`pTimer`	Pointer to the `OS_TIMER` data structure containing the data of the timer

Return value

Unsigned integer between 1 and 32767, which is the allowed range of timer values.

Add. information

The timer value is the remaining time until the timer expires and calls its callback function.

6.10. OS_GetTimerStatus

Description

Returns the actual timer status of the specified timer.

Prototype

```
unsigned char OS_GetTimerStatus(OS_TIMER* pTimer);
```

Parameter	Meaning
`pTimer`	Pointer to the `OS_TIMER` data structure containing the data of the timer

Return value

Unsigned char, denoting whether the specified timer is running or not.
0: Timer is stopped
!=0: Timer is running

Add. information

None.

6.11. OS_GetpCurrentTimer

Description

Returns a pointer to the timer structure of the timer that actually expired.

Prototype

```
OS_TIMER* OS_GetpCurrentTimer(void);
```

Return value

OS_TIMER* points to the control structure of a timer.

Add. information

The return value of `OS_GetpCurrentTimer()` is valid during execution of a timer callback function, otherwise it is undetermined.
If only one callback function should be used for multiple timers, this function can be used to examine the timer that expired.

```
#include "RTOS.H"

/********************************************************
*
*       Types
*/

typedef struct {       Timer object with own user data
  OS_TIMER Timer;
  void*    pUser;
} TIMER_EX;

/********************************************************
*
*       Variables
*/

TIMER_EX Timer_User;
int a;

/********************************************************
*
*       Local Functions
*/

void CreateTimer(TIMER_EX* timer, OS_TIMERROUTINE* Callback, OS_UINT Timeout,
                 void* pUser) {
  timer->pUser = pUser;
  OS_CreateTimer((OS_TIMER*) timer, Callback, Timeout);
}

void cb(void) { /* timer callback function for multiple timers  */
  TIMER_EX* p = (TIMER_EX*)OS_GetpCurrentTimer();
  void* pUser = p->pUser;         /* Examine user data */

  OS_RetriggerTimer(&p->Timer);   /* retrigger timer   */
}

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

int main(void) {
  OS_InitKern();        /* initialize OS */
  OS_InitHW();          /* initialize Hardware for OS */
  CreateTimer(&Timer_User, cb, 100, &a);
  OS_Start();           /* Start multitasking */
  return 0;
```

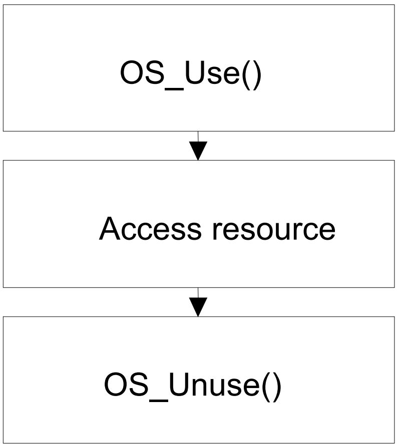
7. Resource semaphores

Resource semaphores are the type of semaphores that are most widely used. Resource semaphores are used to manage resources by avoiding conflicts caused by simultaneous use of a resource. The resource managed can be of any kind: a part of the program that is not reentrant, a piece of hardware like the display, a flash prom that can only be written to by a single task at a time, a motor in a CNC-control that can only be controlled by one task at a time and a lots more.

The basic procedure is the following:

Any task that uses the resource, first claims it calling the `OS_Use` or `OS_Request` routines of ***embOS***. If the resource is available, the program execution of the task continues, but the resource is blocked for other tasks. When the task releases the resource, it does that by calling the `OS_Unuse` routine of ***embOS***. If a second task tries to use the same resource while it is used by the first task, this task is suspended until the first task releases the resource. However, if the first task that uses the resource calls `OS_Use` again for that resource, it is not suspended because the resource is blocked only for other tasks.

The following little diagram illustrates the process of using a resource:



A resource semaphore contains a counter that keeps track of how many times the resource has been claimed by calling `OS_Request` or `OS_Use` by that task. It is released when that counter reaches 0, which means the `OS_Unuse` routine has to be called exactly the same number of times as the `OS_Use` routine. If `OS_Unuse` is not called as many times as `OS_Use` / `OS_Request`, the resource remains blocked for other tasks.

On the other side a task can not release a resource that it does not own by calling `OS_Unuse`. In the debug version, a call of `OS_Unuse` for a semaphore that is not owned by this task will result in a call to the error handler `OS_Error`.

(→ Debugging)

7.1. Example for use of Resource semaphore

Here 2 tasks access an LC display completely independent from each other. The problem is that one task may not interrupt the other task while it is writing to the LCD because in this case the first task would position the cursor, could get interrupted, the second task repositions the cursor and the first task writes to the wrong place in the LCD' s memory. So every time before the LCD is accessed by a task, the resource (the LCD) is claimed by calling `OS_Use` (and is automatically waited for if the resource is blocked). After the LCD has been written to, the resource is released by a call to `OS_Unuse`.

```
/*
*  demo program to illustrate the use of resource semaphores
*/
char StackMain[100], StackClock[50];
OS_TASK TaskMain,TaskClock;
OS_SEMA SemaLCD;


void Clock(void) {
  char t=-1;
  char s[] = "00:00";
  while(1) {
    while (TimeSec==t) Delay(10);
    t= TimeSec;
    s[4] = TimeSec%10+'0';
    s[3] = TimeSec/10+'0';
    s[1] = TimeMin%10+'0';
    s[0] = TimeMin/10+'0';
    OS_Use(&SemaLCD);        /* make sure nobody else uses LCD */
      LCD_Write(10,0,s);
    OS_Unuse(&SemaLCD);             /* release LCD  */
  }
}

void Main(void) {
  signed char pos ;
  LCD_Write(0,0,"Software tools by Segger !    ") ;
  OS_Delay(2000);
  while (1) {
    for ( pos=14 ; pos >=0 ; pos-- ) {
      OS_Use(&SemaLCD);             /* make sure nobody else uses LCD  */
      LCD_Write(pos,1,"train ");    /* draw train   */
      OS_Unuse(&SemaLCD);           /* release LCD  */
      OS_Delay(500);
    }
    OS_Use(&SemaLCD);        /* make sure nobody else uses LCD  */
    LCD_Write(0,1,"     ") ;
    OS_Unuse(&SemaLCD);             /* release LCD  */
  }
}

void InitTask(void) {
  OS_CREATETASK(&TaskMain,  0, Main,   50, StackMain);
  OS_CREATETASK(&TaskClock, 0, Clock, 100, StackClock);
  OS_CREATERSEMA(&SemaLCD);  /* Creates resource semaphore  */
}
```

In most applications, the routines that access a resource should automatically call `OS_Use` and `OS_Unuse` so when using the resource you do not have to worry about it and can use it just like in a single task system. The following is an example for how to implement the resource semaphore usage into the routines that actually access the display:

```
/*
*  simple example when accessing single line dot matrix LCD
*/

OS_RSEMA RDisp; /* define resource semaphore  */

void UseDisp() {      /* simple routine to be called before using display */
  OS_Use(&RDisp);
}

void UnuseDisp() {    /* simple routine to be called after using display  */
  OS_Unuse(&RDisp);
}

void DispCharAt(char c, char x) {
  UseDisp();
  LCDGoto(x, y);
  LCDWrite1(ASCII2LCD(c));
  UnuseDisp();
}

void DISPInit(void) {
  OS_CREATERSEMA(&RDisp);
}
```

7.2. OS_CREATERSEMA

Description

Creates a resource semaphore.

Prototype

```
void OS_CREATERSEMA(OS_RSEMA* pRSema);
```

Parameter	Meaning
pRSema	Pointer to the data structure for a resource semaphore

Return value

Void

Add. information

After creation, the resource is not blocked; the value of the counter is 0.

7.3. OS_Use: Using a Resource

Description

Claims the resource and blocks it for other tasks.

Prototype

```
int OS_Use(OS_RSEMA* pRSema);
```

Parameter	Meaning
pRSema	Pointer to the data structure for a resource semaphore

Return value

Returns the counter value of the semaphore.
A return value larger than 1 means, the resource was already locked by the calling task.

Add. information

If a resource is already blocked by an other task, the task is suspended until the resource is available again.

The following happens:
Case a)

- The resource is not in use:
  If the resource is not used by a task, which means the counter of the semaphore is 0, the resource will be blocked for other tasks by incrementing the counter and writing a unique code for the task that uses it into the semaphore.

Case b)

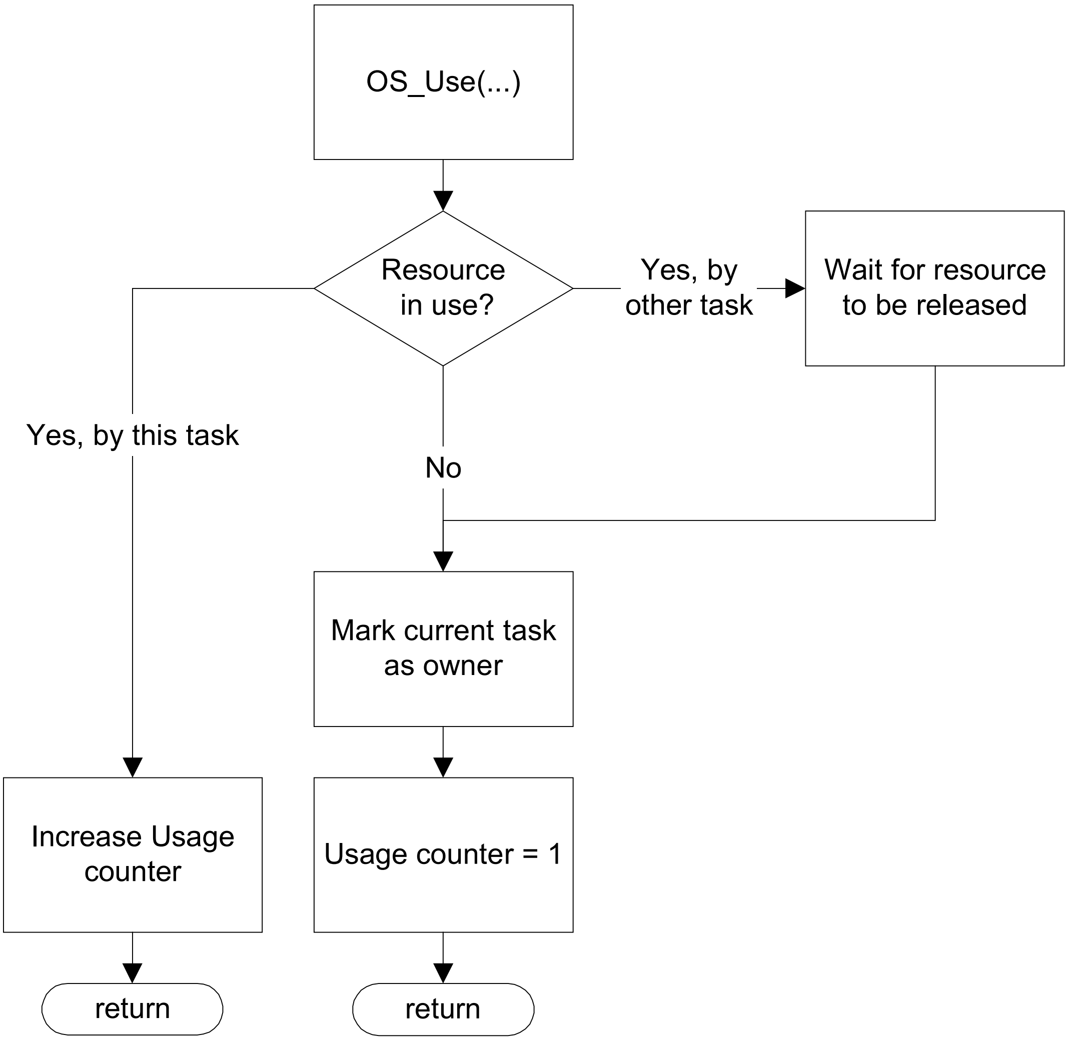
- The resource is used by this task:
  The counter of the semaphore is simply incremented. The program continues without a break.

Case c)

- The resource is already used by an other task:
  The execution of this task is halted until the resource semaphore is released. In the meantime if the task blocked by the resource semaphore has a higher priority than the task blocking the semaphore the blocking task is assigned the priority of the task requesting the resource semaphore. This is called priority inversion. Priority inversion can only temporarily increase the priority of a task, never reduce it.

An unlimited number of tasks can wait for a resource semaphore. According to the rules of the scheduler, of all the tasks waiting for the resource, the task with the highest priority will get access to the resource and can continue program execution.

The following diagram illustrates the function of the OS_Use routine



7.4. OS_Unuse: Release Resource

Description

Releases the semaphore currently in use by the task.

Prototype

```
void OS_Unuse(OS_RSEMA * pRSema);
```

Parameter	Meaning
`pRSema`	Pointer to the data structure for a resource semaphore

Return value

Void.

Add, information

`OS_Unuse()` may be used on a resource semaphore only after that semaphore has been used by calling `OS_Use()` or `OS_Request()`. `OS_Unuse()` decrements the usage counter of the semaphore which may never become negative. If this counter becomes negative, the debug version will call the ***embOS*** error handler.

7.5. OS_Request

Description

Requests the specified semaphore, blocks it for other tasks if it is available. Continues execution in any case.

Prototype

```
char OS_Request(OS_RSEMA* pRSema);
```

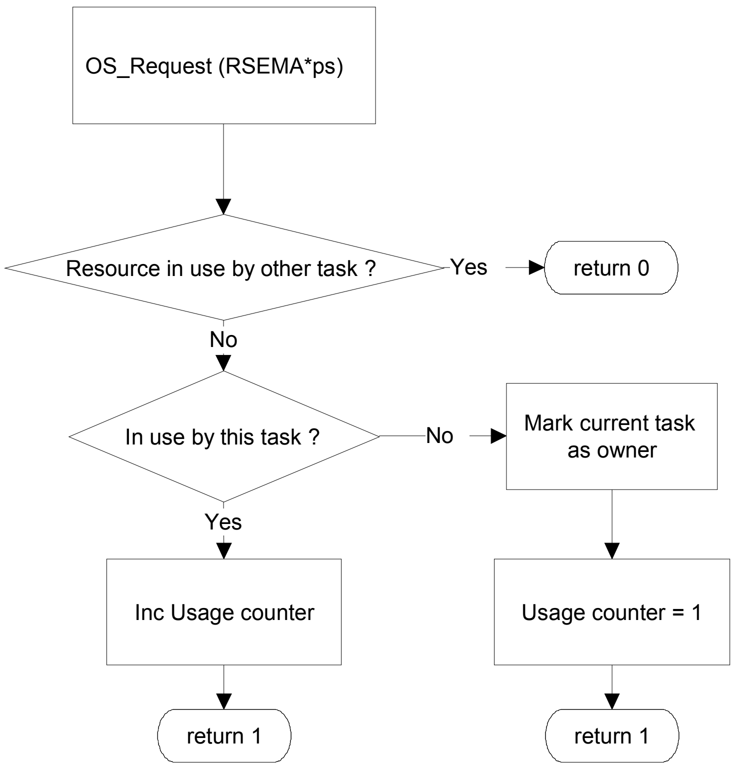
Parameter	Meaning
pRSema	Pointer to the data structure for a resource semaphore

Return value

1 Resource available, in use now
0 Resource was not available

Add. Information

The following diagram illustrates how *OS_Request* works:



Example

```
if (!OS_Request(&RSEMA_LCD) ) {
  LED_LCDBUSY = 1;           /* indicate that task is waiting for     */
                              /* resource                              */
  OS_Use(&RSEMA_LCD);        /* wait for resource                     */
  LED_LCDBUSY = 0;           /* indicate task is no longer waiting*/
}
  DispTime();                /* Access the resource LCD               */
  OS_Unuse(&RSEMA_LCD);            /* resource LCD is no longer needed      */
```

7.6. OS_GetSemaValue

Description

Returns the value of the usage counter of the specified resource semaphore.

Prototype

```
int OS_GetSemaValue(OS_SEMA* pSema);
```

Parameter	Meaning
pSema	Pointer to the data structure for a resource semaphore

Return value

Returns the counter of the semaphore. 0 means the resource is available.

Add. information

None.

7.7. OS_GetResourceOwner

Description

Returns a pointer to the task that is currently using (blocking) the resource.

Prototype

```
TASK* OS_GetResourceOwner(OS_RSEMA* pSema);
```

Parameter	Meaning
pSema	Pointer to the data structure for a resource semaphore

Return value

If the resource is available, the NULL pointer is returned.

Add. information

None.

8. Counting Semaphores

Counting semaphores are counters that are managed by ***embOS***. They are not as widely used as resource semaphores, events or mailboxes, but they can be very useful some times. They are used in situations where a task needs to wait for something that can be signaled one or more times. The semaphores can be accessed from any point, any task, any interrupt in any way.

8.1. Example for OS_SignalCSema and OS_WaitCSema

```
char Stack0[96], Stack1[64];  /* stack-space  */
OS_TASK TCB0, TCB1;           /* Data-area for tasks (task-control-blocks)  */
OS_CSEMA SEMALCD;

void Task0(void) {
Loop:
  Disp("Task0 will wait for task 1 to signal");
  OS_WaitCSema(&SEMALCD);
  Disp("Task1 has signaled !!");
  OS_Delay(100);
  goto Loop;
}

void Task1(void) {
Loop:
  OS_Delay(5000);
  OS_SignalCSema(&SEMALCD);
  goto Loop;
}


void InitTask(void) {
  OS_CREATETASK(&TCB0, NullTask0, 100, Stack0);  /* Create Task0      */
  OS_CREATETASK(&TCB1, NullTask1,  50, Stack1);  /* Create Task1      */
  OS_CREATECSEMA(&SEMALCD);                      /* Create Semaphore  */
}
```

8.2. OS_CREATECSEMA

Description

Creates a counting semaphore with an initial count value of zero.

Prototype

```
void OS_CREATECSEMA (OS_CSEMA* pCSema);
```

Parameter	Meaning
pCSema	Pointer to a data structure of type OS_CSEMA

Return value

void.

Add. information

In order to create a counting Semaphore, a data structure of the type *CSEMA* has to be defined in memory and initialized using *OS_CREATECSEMA()*.
The value of a semaphore after creation using this macro is always zero.
If for any reason you have to create a semaphore with an initial counting value above zero, you have to use the function *OS_CreateCSema()*.

8.3. OS_CreateCSema

Description

Creates a counting semaphore with a specified initial count value.

Prototype

```
int OS_CreateCSema(OS_CSEMA* pCSema,
                   unsigned char InitValue);
```

Parameter	Meaning
pCSema	Pointer to the data structure of a counting semaphore
InitValue	Initial count value of the semaphore 0 <= InitValue <= 255

Return value

void.

Add. information

In order to create a counting Semaphore, a data structure of the type *CSEMA* has to be defined in memory and initialized using *OS_CreateCSema()*.
If the value of the semaphore after creation should be zero, the macro *OS_CREATECSEMA()* should be used.

8.4. OS_SignalCSema: Incrementing

Description

Increments the counter of the semaphore

Prototype

```
void OS_SignalCSema(OS_CSEMA * pCSema);
```

Parameter	Meaning
pCSema	Pointer to the data structure of a counting semaphore

Return value

Void.

Add. information

OS_SignalCSema() signals an event to a semaphore by incrementing the counter of the semaphore. If one or more tasks are waiting for an event to be signaled to this semaphore, the task that has the highest priority will become the active task.
The counter can have a maximum value of 255. The application should make sure that this limit will not be exceeded.

8.5. OS_WaitCSema: Decrementing

Description

Decrementing the semaphore counter

Prototype

```
void OS_WaitCSema(OS_CSEMA* pCSema);
```

Parameter	Meaning
pCSema	Pointer to the data structure of a counting semaphore

Return value

Void

Add. information

If the counter of the semaphore is not 0, the counter is decremented and program execution continues. If the counter is 0, *WaitCSema* waits until the counter is incremented by an other task, a timer or an interrupt-handler via a call to *OS_SignalCSema()*. The counter is then decremented and program execution continues.
An unlimited number of tasks can wait for a semaphore. According to the rules of the scheduler, of all the tasks waiting for the semaphore, the task with the highest priority will continue program-execution.

8.6. OS_WaitCSemaTimed: Decrementing with timeout

Description

Decrementing the semaphore counter, if semaphore is available within the specified time.

Prototype

```
int OS_WaitCSemaTimed(OS_CSEMA* pCSema,
                      int TimeOut);
```

Parameter	Meaning
pCSema	Pointer to the data structure of a counting semaphore
TimeOut	Maximum time until semaphore should be available

Return value

int
0: Failed, semaphore not available within timeout time
1: OK, semaphore is available

Add. information

If the counter of the semaphore is not 0, the counter is decremented and program execution continues. If the counter is 0, *OS_WaitCSemaTimed()* waits until the semaphore is signaled by an other task, a timer or an interrupt-handler via a call to *OS_SignalCSema()* within the specified timeout time.
The counter is then decremented and program execution continues.
If the semaphore was not signaled within the specified time, the program execution continues, but receives a return value of zero.
An unlimited number of tasks can wait for a semaphore. According to the rules of the scheduler, of all the tasks waiting for the semaphore, the task with the highest priority will continue program-execution.

8.7. OS_GetCSemaValue

Description

Returns count-value

Prototype

```
int OS_GetCSemaValue(OS_SEMA* pCSema);
```

Return value

Count-value of the semaphore.

Add. information

None

8.8. OS_DeleteCSema

Description

Deletes the specified semaphore. The memory of that semaphore can be re-used for other purposes.

Prototype

```
void OS_DeleteCSema(OS_CSEMA* pCSema);
```

Parameter	Meaning
pCSema	Pointer to the data structure of a counting semaphore

Return value

Void

Add. information

Before deleting a semaphore, make sure that no task is waiting at that semaphore and no task will signal that semaphore later.
The debug version will reflect an error, if a deleted semaphore is signaled.

9. Mailboxes

9.1. Why mailboxes ?

In the preceding chapter the task synchronization by use of semaphores has been described. Unfortunately, semaphores can not transfer data from one task to an other. If we need to transfer data from one task to an other via a buffer for example, we could use a resource semaphore every time before we access the buffer. This would make the program less efficient. An other major disadvantage would be that we can not access this buffer from an interrupt handler since the interrupt handler is not allowed to wait for the resource semaphore. One way out would be the usage of global variables. In this case we have to disable interrupts every time and everywhere we access these variables. This is possible, but it is a path full of pitfalls. Most of all, we have one disadvantage: It is not easy for a task to wait for a character to be placed in the buffer without polling the global variable that contains the number of characters in the buffer. Again, there is a way: The task could be notified by an event signaled to the task every time a character is placed in the buffer.
Complicated you think ?
That is why there is an easier way to do this with a real time OS:
The use of mailboxes.

9.2. Basics

A *mailbox* is a buffer that is managed by the real time operating system. The buffer behaves like a normal buffer: you can put something (called a message) in and retrieve it later. Mailboxes usually work as FIFO: first in, first out. So a message that is put in first will usually be retrieved first. Message might sound abstract. But really message means just "item of data". It will become clear in the following typical applications explained in the following chapter.

9.3. Typical applications

A keyboard buffer

In most programs, you use either a task, a software timer or an interrupt handler to check the keyboard. When you detect that a key has been pressed, you put that key in a mailbox that is used as keyboard buffer. The message is then retrieved by the task that handles keyboard input. The message in this case will be typically a single byte that holds the key code, the message size is 1 byte. The advantages: The management of the keyboard buffer is very efficient, you do not have to worry about it since it is reliable, proven code and you have a type ahead buffer at no extra cost. On top of that, a task can easily wait for a key to be pressed without having to poll the buffer. It simply calls the `OS_GetMail` routine for that mailbox. The number of keys that can be stored in the type ahead buffer depends on the size of the mailbox buffer only, which you define when creating the mailbox.

A buffer for serial I/O

In most cases, serial I/O is done with the help of interrupt handlers. The communication to these interrupt handlers is very easy using mailboxes. Both your task programs and your interrupt handlers store or retrieve data to/from the same mailboxes.
For interrupt driven sending: The task places character(s) in the mailbox using `OS_PutMail` or `OS_PutMailCond`, the interrupt handler that is activated when a new character can be send retrieves this character with `OS_GetMailCond`.
For interrupt driven receiving: The interrupt handler that is activated when a new character is received puts it into the mailbox using `OS_PutMailCond`, the task receives it using `OS_GetMail` or `OS_GetMailCond`.
Again, the message size will be 1 character.

A buffer for commands sent to a task

Assume you have one task that controls a motor as you might have in applications that control a machine. An easy way to give commands to this task on how to control the motor would be to define a structure for commands. The message size will then be the size of this structure.

9.4. Number of and size of mailboxes, type of mail

The number of mailboxes is limited by the amount of available memory only.

Message size:	1 <= x <= 127 byte.
Number of messages	1 <= x <= 32767.

These limitations have been placed on mailboxes in order to guarantee efficient coding and to keep the management very efficient.
However, these limitations normally are not a problem. If they are in your case, please give us feedback and we will try to find a solution.
To handle message sizes above 127 bytes you might use queues.

9.5. OS_CREATEMB: Creating a mailbox

Description

Creates a new mailbox.

Prototype

```
void OS_CREATEMB(OS_MAILBOX* pMB,
                 char        sizeofMsg,
                 char        maxnofMsg,
                 void*       pMsg);
```

Parameter	Meaning
pMB	Pointer to a data structure of type OS_MAILBOX reserved for the management of the mailbox
sizeofMsg	Size of a message in bytes
maxnofMsg	Max. no. of messages
pMsg	Pointer to a memory area used as buffer. The buffer has to be big enough to hold the given number of messages of the given size: `sizeofMsg` * `maxnofMsg` bytes

Return value

Void.

Examples

Mailbox used as keyboard buffer:

```
OS_MAILBOX MBKey;
char MBKeyBuffer[6];

void InitKeyMan(void) {
  /* create mailbox functioning as type ahead buffer */
  OS_CREATEMB(&MBKey, 1, sizeof(MBKeyBuffer), &MBKeyBuffer);
}
```

Mailbox used to transfer complex commands from one task to an other:

```
/*
* example for mailbox used to transfer commands to a task
* that controls 2 motors
*/

typedef struct {
  char Cmd;
  int Speed[2];
  int Position[2];
} MOTORCMD ;

OS_MAILBOX MBMotor;

#define MOTORCMD_SIZE 4
char     BufferMotor[sizeof(MOTORCMD)*MOTORCMD_SIZE];

void MOTOR_Init(void) {
  /* create mailbox that holds commands messages */
  OS_CREATEMB(&MBMotor, sizeof(MOTORCMD), MOTORCMD_SIZE, &BufferMotor);
}
```

9.6. Single byte mailbox functions

In a lot (if not the most) situations, mailboxes are used to just hold and transfer single byte messages. This is for example the case for a mailbox that takes the character received or sent via serial interface or normally for a mailbox used as keyboard buffer. In some of these case time is very critical, especially if a lot of data is transferred in short periods of time. In order to minimize the overhead caused by the mailbox management of ***embOS***, there are all of the functions described above available for single byte mailboxes. The general functions `OS_PutMail`, `OS_PutMailCond`, `OS_GetMail`, `OS_GetMailCond` can transfer messages of sizes between 1 and 127 bytes each. Their single byte equivalents `OS_PutMail1`, `OS_PutMailCond1`, `OS_GetMail1`, `OS_GetMailCond1` function exactly the same way with the exception that they execute a lot faster since the management is easier. It is recommended you use the single byte versions if you transfer a lot of single byte data via mailboxes.

`OS_PutMail1`, `OS_PutMailCond1`, `OS_GetMail1`, `OS_GetMailCond1` function exactly the same way as their more universal equivalents and are therefore not described in detail. The only difference is that they can only be used for single byte mailboxes.

9.7. OS_PutMail / OS_PutMail1: Store message

Description

Stores a new message of the predefined size in the mailbox.

Prototype

```
void OS_PutMail  (OS_MAILBOX * pMB, void* pMail);
void OS_PutMail1 (OS_MAILBOX * pMB, const char* pMail);
```

Parameter	Meaning
pMB	Pointer to the mailbox
pMail	Pointer to the message to store

Return value

Void.

Add. information

If the mailbox is full, the task is suspended.
Since this routine might require a suspension, it must not be called from an interrupt routine. Use →`OS_PutMailCond` →`OS_PutMailCond1` instead.

Example

Single byte mailbox as keyboard buffer:

```
OS_MAILBOX MBKey;
char MBKeyBuffer[6];

void KEYMAN_StoreKey(char k) {
  OS_PutMail1(&MBKey, &k);  /* store key, wait if no space in buffer */
}

void KEYMAN_Init(void) {
  /* create mailbox functioning as type ahead buffer */
  OS_CREATEMB(&MBKey, 1, sizeof(MBKeyBuffer), &MBKeyBuffer);
}
```

9.8. OS_PutMailCond / OS_PutMailCond1: Store Message if possible

Description

Stores a new message of the predefined size in the mailbox, if the mailbox is able to accept one more message. This routine will never suspend the calling task.

Prototype

```
char OS_PutMailCond  (OS_MAILBOX * pMB, void* pMail);
char OS_PutMailCond1 (OS_MAILBOX * pMB, const char* pMail);
```

Parameter	Meaning
pMB	Pointer to the mailbox
pMail	Pointer to the message to store

Return value

Returns 0 if message could be stored (success) , otherwise 1.

Add. information

If the mailbox is full, the message is not stored.
This routine can be called from an interrupt routine.

Example

```
OS_MAILBOX MBKey;
char MBKeyBuffer[6];


char KEYMAN_StoreCond(char k) {
  return OS_PutMailCond1(&MBKey, &k); /* store key if space in buffer */
}
```

This example can be used with the sample program shown earlier to create a mailbox as keyboard buffer.

9.9. OS_GetMail / OS_GetMail1

Description

Retrieves a new mail of the predefined size from a mailbox and will suspend the calling task until a message is available.

Prototype

```
void OS_GetMail (OS_MAILBOX * pMB, void* pDest);
void OS_GetMail1(OS_MAILBOX * pMB, char* pDest);
```

Parameter	Meaning
`pMB`	Pointer to the mailbox
`pDest`	Pointer to the memory area that the message should be stored at. You have to make sure that this pointer points to a valid memory area and that there is sufficient space for an entire message. The message size (in bytes) has been defined upon creation of the mailbox

Return value

Void.

Add. information

If the mailbox is empty, the task is suspended until the mailbox receives a new message.
Since this routine might require a suspension, it may not be called from an interrupt routine. Use →`OS_GetMailCond` / →`OS_GetMailCond1` instead if you have to retrieve data from a mailbox from within an ISR.

Example

```
OS_MAILBOX MBKey;
char MBKeyBuffer[6];

char WaitKey(void) {
  char c;
  OS_GetMail1(&MBKey, &c);
  return c;
}
```

9.10. OS_GetMailCond / OS_GetMailCond1

Description

Retrieves a new mail of the predefined size from a mailbox, if a message is available. This function never suspends the calling task.

Prototype

```
char OS_GetMailCond (OS_MAILBOX * pMB, void* pDest);
char OS_GetMailCond1(OS_MAILBOX * pMB, char* pDest);
```

Parameter	Meaning
pm	Pointer to the mailbox
pDest	Pointer to the memory area that the message should be stored at. You have to make sure that this pointer points to a valid memory area and that there is sufficient space for an entire message. The message size (in bytes) has been defined upon creation of the mailbox

Add. information

If the mailbox is empty, no message is retrieved, but the program execution continues.
Can be called from an interrupt routine.

Return value

0 on success: message retrieved
1 no message could be retrieved (mailbox is empty !), destination remains unchanged

Example

```
OS_MAILBOX MBKey;
char MBKeyBuffer[6];

/*
* If a key has been pressed, it is taken out of the mailbox and returned to
* caller.
* Otherwise, 0 is returned.
*/
char GetKey(void) {
  char c =0;
  OS_GetMailCond1(&MBKey, &c)
  return c;
}
```

9.11. OS_ClearMB: Empty a Mailbox

Description

Clears all messages in the specified mailbox.

Prototype

```
void OS_ClearMB(OS_MAILBOX * pMB);
```

Parameter	Meaning
pMB	Pointer to the mailbox

Return value

Void.

Add. information

None.

Example

```
OS_MAILBOX MBKey;
char MBKeyBuffer[6];

/*
* Clear keyboard type ahead buffer
*/
void ClearKeyBuffer(void) {
  OS_ClearMB(&MBKey);
}
```

9.12. OS_GetMessageCnt

Description

Return no. of messages.

Prototype

```
char OS_GetMessageCnt(OS_MAILBOX * pMB);
```

Parameter	Meaning
pMB	Pointer to the mailbox

Return value

Returns the number of messages currently in the mailbox.

Add. information

None.

Example

```
char GetKey(void) {
  if (OS_GetMessageCnt(&MBKey)) return WaitKey();
  return 0;
}
```

9.13. OS_DeleteMB

Description

Deletes the specified mailbox.

Prototype

```
void OS_DeleteMB(OS_MAILBOX * pMB);
```

Parameter	Meaning
pMB	Pointer to the mailbox

Return value

Void.

Add. information

In order to keep the system fully dynamic, it is essential that mailboxes can be created dynamically. This also means there has to be a way to delete the mailbox when it is no longer needed. The memory that has been used by the mailbox for the control structure and the buffer can then be reused or reallocated.
It is the programmers responsibility to:

1. make sure that the program does not use the mailbox any more
2. make sure that the mailbox that shall be deleted does actually exist, i.e. has been created first before deleting the mailbox.

Example

```
OS_MAILBOX MBSerIn;
char MBSerInBuffer[6];

void Cleanup(void) {
  OS_DeleteMB(MBSerIn);
  return 0;
}
```

10. Queues

10.1. Why Queues ?

In the preceding chapter inter task communication using mailboxes was described. Mailboxes can handle small messages with fixed data size only.
Queues enable inter task communication with large messages or messages of various size.

10.2. Basics

A queue consists of a data buffer and a control structure that is managed by the real time operating system. The queue behaves like a normal buffer: you can put something (called a message) in and retrieve it later. Queues work as FIFO: first in, first out. So a message that is put in first will be retrieved first.
There are two major differences to mailboxes:

1. Queues accept messages of various size. When putting a message into a queue, the message size has to be passed as additional parameter.
2. Retrieving a message from the queue does not copy the message but gives a pointer to the message and the size of the message. This enhances performance, because the data is copied only once, when the message is written into the queue.
3. The retrieving function has to delete every message after processing it.

10.3. Number of and size of queues, type of messages

The number of queues is limited by the amount of available memory only.
The size of a queue is limited by the amount of available memory only.
Any data structure can be written into a queue. The message size is not fixed.

10.4. OS_Q_Create: Creating a message queue

Description

Creates and initializes a message queue.

Prototype

```
void OS_Q_Create(OS_Q* pQ,
                 void*pData,
                 OS_UINT Size)
```

Parameter	Meaning
`pQ`	Pointer to a data structure of type OS_Q reserved for the management of the message queue
`pData`	Pointer to a memory area used as data buffer for the queue.
`Size`	Size of the data buffer in bytes

Return value

Void.

Examples

Queue used to transfer data to memory:

```
define MEMORY_QSIZE 10000;
static OS_Q  _MemoryQ;
static char  _acMemQBuffer[MEMORY_QSIZE];

void MEMORY_Init(void) {
  OS_Q_Create(&_MemoryQ, &_acMemQBuffer, sizeof(_acMemQBuffer));
}
```

10.5. OS_Q_Put: Store message

Description

Stores a new message of given size in a queue.

Prototype

`int OS_Q_Put(OS_Q* pQ, const void* pSrc, OS_UINT Size)`

Parameter	Meaning
pQ	Pointer to the queue
pSrc	Pointer to the message to store
Size	Size of the actual message to store

Return value

Returns 0 if message could be stored (success) , otherwise 1.

Add. information

If the queue is full, the task is not suspended, the function returns a value unequal to zero.
Since this routine never suspends a task, it may also be called from an interrupt routine.

Example

```
char MEMORY_Write(char* pData, int Len) {
  return OS_Q_Put(&_MemoryQ, pData, Len));
}
```

10.6. OS_Q_GetPtr: Retrieve message

Description

Retrieves a message from the queue, if one message is available.
This routine will suspend the calling task, as long as no message is available in the queue.

Prototype

`int OS_Q_GetPtr(OS_Q* pQ, void**ppData)`

Parameter	Meaning
`pQ`	Pointer to the queue
`ppData`	Address of pointer to the message to be retrieved from queue.

Return value

Returns the message size of the retrieved message.
Sets the pointer to the actual message that should be retrieved.

Add. information

If the queue is empty, the calling task is suspended.
Therefore this routine must not be called from within an interrupt routine.
The retrieved message is not removed from the queue. This has to be done by a call of `OS_Q_Purge()after` the message wass processed.

Example

```
static void MemoryTask(void) {
  char MemoryEvent;
  int Len;
  char* pData;
  while (1) {
    Len = OS_Q_GetPtr(&_MemoryQ, &pData);             /* Get message     */
    Memory_WritePacket(*(U32*)pData, pData+4, Len);  /* Process message */
    OS_Q_Purge(&_MemoryQ);                           /* Delete message  */
  }
}
```

10.7. OS_Q_GetPtrCond: Retrieve message if available

Description

Retrieves a message from the queue, if one message is available and delivers the size of the message as return value.
If no message is available, the functions returns with a size of zero, indicating, that there was no message in the queue. This routine will never suspend the calling task.

Prototype

`int OS_Q_GetPtrCond(OS_Q* pQ, void**ppData)`

Parameter	Meaning
`pQ`	Pointer to the queue
`ppData`	Address of pointer to the message to be retrieved from queue.

Return value

0: No message available in queue.
>0: Size of message that was retrieved from queue.

Add. information

If the queue is empty, the calling task is not suspended, the function returns with zero. The value of ppData is undefined.
If one message could be retrieved, this message is not removed from the queue. This has to be done by a call of `OS_Q_Purge()`after the message was processed.

Example

```
static void MemoryTask(void) {
  char MemoryEvent;
  int Len;
  char* pData;
  while (1) {
    Len = OS_Q_GetPtrCond(&_MemoryQ, &pData);          /* Check message   */
    if (Len > 0) {
      Memory_WritePacket(*(U32*)pData, pData+4, Len);  /* Process message */
      OS_Q_Purge(&_MemoryQ);                           /* Delete message  */
    } else {
      DoSomethingElse();
    }
  }
}
```

10.8. OS_Q_Purge: Delete message in queue

Description

Deletes the last message in queue.

Prototype

```
void OS_Q_Purge(OS_Q* pQ)
```

Parameter	Meaning
pQ	Pointer to the queue

Return value

Void.

Add. information

This routine should be called by the task that retrieved the last message from queue, after the message is processed.

Example

```
static void MemoryTask(void) {
  char MemoryEvent;
  int Len;
  char* pData;
  while (1) {
    Len = OS_Q_GetPtr(&_MemoryQ, &pData);             /* Get message     */
    Memory_WritePacket(*(U32*)pData, pData+4, Len);  /* Process message */
    OS_Q_Purge(&_MemoryQ);                            /* Delete message  */
  }
}
```

10.9. OS_Q_GetMessageCnt: Get number of messages in queue

Description

Deletes the last message in queue.

Prototype

```
void OS_Q_Purge(OS_Q* pQ)
```

Parameter	Meaning
pQ	Pointer to the queue

Return value

Void.

Add. information

This routine should be called by the task that retrieved the last message from queue, after the message is processed.

Example

```
static void MemoryTask(void) {
  char MemoryEvent;
  int Len;
  char* pData;
  while (1) {
    Len = OS_Q_GetPtr(&_MemoryQ, &pData);            /* Get message     */
    Memory_WritePacket(*(U32*)pData, pData+4, Len);  /* Process message */
    OS_Q_Purge(&_MemoryQ);                           /* Delete message  */
  }
}
```

11. Events

Events are another means of communication between tasks. In contrast to semaphores and mailboxes, events are messages to a single, specified recipient. In other words: An event is send to a specified task.

The purpose of an event is to enable a task to wait for a particular event (or for one of several events) to occur. This task can be kept inactive until the event is signaled by an other task, a S/W timer or an interrupt handler. The event can be anything that the software is made aware of in any way. Examples are the change of an input signal, the expiration of a timer, a key press, the reception of a character or a complete command.

Every task has an 1 byte (8 bits) mask, which means that 8 different events can be signaled to and distinguished by every task.

By calling `OS_WaitEvent,` a task waits for one of the events specified as bitmask.

As soon as one of the events actually occurs, it has to be signaled to this task by calling `OS_SignalEvent`.

The waiting task will then be put in the ready state immediately and activated according to the rules of the scheduler as soon as it becomes the task with the highest priority of all the tasks in the READY state.

11.1. OS_WaitEvent

Description

Waits for the specified event and clears the event memory after the event occurs.

Prototype

```
char OS_WaitEvent(char EventMask);
```

Parameter	Meaning
EventMask	The events that the task will be waiting for.

Return value

Returns all events that have actually occurred.

Add. information

Lets the task wait for the occurrence of one of the specified events and then clears the event memory. If none of the specified events is signaled, the task is suspended. The first of the specified events will wake the task. These events have to be signaled by an other task, a S/W timer or an interrupt handler.
Every 1 bit in the event mask enables the according event.

Example

```
OS_WaitEvent(3);           /* Wait for event 1 or 2 to be signaled */
```

Further example: → `OS_SignalEvent`

11.2. OS_WaitSingleEvent

Description

Waits for the specified event and clears only those event after the event occurs.

Prototype

```
char OS_WaitSingleEvent(char EventMask);
```

Parameter	Meaning
EventMask	The events that the task will be waiting for.

Return value

Returns all masked events that have actually occurred.

Add. information

Lets the task wait for the occurrence of one of the specified events and then clears the masked events only. If none of the specified events is signaled, the task is suspended. The first of the specified events will wake the task. These events have to be signaled by an other task, a S/W timer or an interrupt handler.
Every 1 bit in the event mask enables the according event.
All unmasked events remain unchanged.

Example

```
OS_WaitSingleEvent(3);       /* Wait for event 1 or 2 to be signaled */
```

11.3. OS_WaitEventTimed

Description

Waits for the specified events for a given time.

Prototype

```
char OS_WaitEventTimed(char EventMask, int TimeOut);
```

Parameter	Meaning
EventMask	The events that the task will be waiting for.
TimeOut	Maximum time in timer ticks, until the events have to be signaled.

Return value

Returns the events that have actually occurred within the specified time.
Returns 0, if no events were signaled in time

Add. information

Lets the task wait for the occurrence of one of the specified events and then clears the event memory. If none of the specified events is available, the task is suspended for the given time. The first of the specified events will wake the task, if the event has been signaled by an other task, a S/W timer or an interrupt handler within the specified TimeOut time.
If no event was signaled, the Task is activated after the specified TimeOut time, all actual events are returned and then cleared.
Every 1 bit in the event mask enables the according event.

Example

```
OS_WaitEventTimed(3, 10); /* Wait for event 1/2 to be signaled within 10 ms */
```

11.4. OS_WaitSingleEventTimed

Description

Waits for the specified events for a given time.

Prototype

```
char OS_WaitSingleEventTimed(char EventMask, int TimeOut);
```

Parameter	Meaning
EventMask	The events that the task will be waiting for.
TimeOut	Maximum time in timer ticks, until the events have to be signaled.

Return value

Returns the masked events that have occurred within the specified time.
Returns 0, if none of the masked events were signaled in time

Add. information

Lets the task wait for the occurrence of one of the specified events and then clears the masked events.
Unmasked events remain unchanged.
If none of the specified events is available, the task is suspended for the given time. The first of the specified events will wake the task, if the event has been signaled by an other task, a S/W timer or an interrupt handler within the specified TimeOut time.
If no event was signaled, the task is activated after the specified TimeOut time and the function returns zero.
Every 1 bit in the event mask enables the according event.

Example

```
OS_WaitSingleEventTimed(3, 10); /* Wait for event 1/2 to be signaled within 10
ms */
```

11.5. OS_SignalEvent

Description

Signals the event(s) specified to the task specified.

Prototype

```
void OS_SignalEvent(char Event, OS_TASK* pTask);
```

Parameter	Meaning
`Event`	The event(s) to signal 1 means event 1 2 means event 2 4 means event 3 ... 128 means event 8 multiple events can be signaled as the sum of the single events, e.g. 6 will signal event 2 & 3
`pTask`	the task that the events are sent to

Return value

Void.

Add. information

If the specified task is waiting for one of these events, it will be put in the ready state and activated according to the rules of the scheduler.
Usually it is sufficient to just signal 1 to the task since it can find out itself which event has occurred.

Example

Task is waiting for serial reception or keyboard

The task that handles the serial input and the keyboard, waits for a character to be received either via keyboard (EVENT_KEYPRESSED) or serial interface (EVENT_SERIN).

```
/*
* just a small demo for events
*/

#define EVENT_KEYPRESSED (1)
#define EVENT_SERIN (2)

char Stack0[96], Stack1[64];  /* stack space  */
OS_TASK TCB0, TCB1;           /* Data area for tasks (task control blocks) */

void Task0(void) {
  while(1)
    OS_WaitEvent(EVENT_KEYPRESSED | EVENT_SERIN)
      /* check & handle key press         */
      /* check & handle serial reception  */
  }
}

void TimerKey(void) {
 /* more code to find out if key has been pressed  */
  OS_SignalEvent(EVENT_SERIN, &TCB0); /* notify Task that key was pressed */
}

void InitTask(void) {
  OS_CREATETASK(&TCB0, 0, Task0, 100, Stack0);        /* Create Task0 */
}
```

If the task would wait for a key to be pressed only, `OS_GetMail` could simply be called. The task would then be deactivated until a key is pressed. If the task has to handle multiple mailboxes as in this case, events are a good option.

11.6. OS_GetEventsOccured

Description

Get List of events

Prototype

```
char OS_GetEventsOccured(OS_TASK* pTask);
```

Parameter	Meaning
`pTask`	The task who's event mask is to be returned `NULL` means current task

Return value

Returns the bit mask of the events that have actually occurred.

Add. information

This is one way for a task to find out which events have been signaled. The task is not suspended, if no events are available. By calling this function, the actual events remain signaled, the event memory is not cleared.

11.7. OS_ClearEvents: Clear List of Events

Description

Returns the actual state of events and then clears the events of the specified task.

Prototype

```
char OS_ClearEvents(OS_TASK* pTask);
```

Parameter	Meaning
pTask	The task who's event mask is to be cleared `NULL` means current task

Return value

Returns the bit mask of the events that were actually signaled before clearing.

12. Stacks

12.1.Some basics

The stack is the memory-area used to store the return-address of function calls, parameters, and local variables and for temporary storage. Interrupt-routines also use the stack to save the return address and flag register, except in case the CPU does have a separate stack for interrupt functions. Please check out the *CPU & Compiler Specifics* manual of ***embOS*** documentation for details on your processor’s stack. A "normal" single-task program needs exactly one stack. In a multitasking system, every task has to have its own stack.
The stack has to have a minimum size, which is determined by the sum of the stack-usage of the routines in the worst-case nesting. If the stack is to small, a section of the memory that is not reserved for the stack will be overwritten, a serious program-failure is most likely to occur.
embOS monitors the stack size and if available also interrupt stack size in the debug version and calls the failure-routine `OS_Error` if it detects a stack-overflow. However, ***embOS*** cannot reliably detect a stack overflow.

A stack that has been defined bigger than necessary does not hurt; it is only a waist of memory.
The debug and stack check builds of ***embOS*** fill the Stack with control characters when it is created and check these control-characters every time the task is deactivated in order to detect a stack-overflow.
In case a stack overflow is detected, `OS_Error` will be called.

12.2.System stack

Before ***embOS*** takes over control (before call to `OS_Start()`), a program does use the so-called system stack. This is the same stack, as a non-embOS program for this CPU would use. After transferring control to ***embOS*** scheduler by calling `OS_Start()`, system stack is used only when no task is executed for the following:

- ***embOS*** Scheduler
- ***embOS*** Software timers (and the callback)

For details regarding required size of your system stack, please refer the *CPU & Compiler Specifics* manual of ***embOS*** documentation.

12.3.Task stack

Each ***embOS*** task does have a separate stack. Location and size of this stack is defined when creating a task. Minimum size of a task stack depends pretty much on the CPU and compiler. For details, please see *CPU & Compiler Specifics* manual of ***embOS*** documentation.

12.4.Interrupt stack

For reduction of stack size in a multi-tasking environment, some processors use a specific stack area for interrupt service routines (hardware interrupt stack). If there is no interrupt stack, you will have to add stack requirements of your interrupt service routines to each task stack.
Even if the CPU does not support an interrupt stack by hardware, ***embOS*** may support a separate stack for interrupts by calling function `OS_EnterIntStack()` at beginning of an interrupt service routine and `OS_LeaveIntStack()` at its very end. In case the CPU does already support hardware interrupt stack or a separate interrupt stack is not supported at all, these function calls are implemented as empty macros.
We recommend using `OS_EnterIntStack()` and `OS_LeaveIntStack()` even if there is currently no additional benefit for your specific CPU, because code using them might reduce stack size on another CPU or a new version of ***embOS*** with support for an interrupt stack for your CPU.
For details about interrupt stack, please check out the *CPU & Compiler Specifics* manual of ***embOS*** documentation.

12.5. OS_GetStackSpace

Description

Returns the unused portion of the stack.

Prototype

```
int OS_GetStackSpace(OS_TCB* pTask);
```

Parameter	Meaning
pTask	The task who's stack space is to be checked `NULL` means current task

Return value

Returns the unused portion of the stack in bytes.

Add. information

In most cases, the stack size required by a task can not be easily calculated, since it takes quite some time to calculate the worst case nesting and the calculation itself is difficult.
There is an other approach:
The required stack size can be figured out using the function `OS_GetStackSpace`. `OS_GetStackSpace` returns the number of unused bytes on the stack. If there is a lot of space left, you can reduce the size of this stack and vice versa.

This function is available in the debug and stack check builds of *embOS* only, since only these initialize the stack space used for the tasks.

Example

```
void CheckSpace(void) {
  printf("Unused Stack[0]  %d", OS_GetStackSpace(&TCB[0]);
  OS_Delay(1000);
  printf("Unused Stack[1]  %d", OS_GetStackSpace(&TCB[1]);
  OS_Delay(1000);
}
```

Attention

This routine does not reliably detect the amount of stack space left. (This is because it can only detect modified bytes on the stack. Unfortunately, space used for register storage or local variables is not always modified. However, in most cases this routine will detect the correct amount of stack bytes.)
In case of doubt, be generous with your stack-space or use other means to verify that the allocated stack space is sufficient.

13. Interrupts

In this chapter, you will find a very basic description about using interrupt service routines in cooperation with *embOS*. Details for your CPU and compiler can be found in the manual “CPU & Compiler Specifics” of *embOS* documentation.
Interrupts are interruptions of a program caused by hardware. Normal interrupts are maskable and can occur at any time unless they are disabled with the CPU's disable-interrupt-instruction.
There are several good reasons for using interrupt-routines. They can respond very fast to external events like the status change on an input, the expiration of a hardware timer, reception or completion of transmission of a character via serial interface or other events.

13.1. Rules for interrupt handlers

General rules

There are some general rules for interrupt handlers. These rules apply to both "single task programming" as well as to multi task programming using ***embOS.***

- Interrupt handlers preserve all registers
  Interrupt handlers have to restore the environment of a task completely. This environment normally consists of the registers only, so the interrupt routine has to make sure that all registers that are modified during interrupt execution have to be saved at the start and restored at the end of the interrupt routine.
- Interrupt handler have to be finished quickly.
  Calculation intensive parts of the program should be kept out of the interrupt handler. The interrupt handler should only be used to store a received value or to trigger an operation in the regular program (a task). It should not wait in any form or perform a polling operation.

Additional rules

A preemptive multitasking system like ***embOS*** needs to know if the program it is interrupting is part of the current task or an interrupt handler. This is so because ***embOS*** can not perform a task switch during the execution of an interrupt handler. However, it can perform the task switch at the end of the interrupt routine.
If it would interrupt the interrupt routine; the interrupt routine would be continued as soon as the interrupted task becomes the current task again. This is not a problem for interrupt handlers that do not allow further interruptions, (which do not enable interrupts) and that do not call any ***embOS*** function.
This leads us to the following rule:

- Interrupt functions that re-enable interrupts or use any ***embOS*** functions need to use `OS_EnterInterrupt()` as first and
  `OS_LeaveInterrupt()` or
  `OS_LeaveInterruptNoSwitch()` as last line.

The task switch then occurs in the routine `OS_LeaveInterrupt()`. The end of the interrupt service routine is executed at a later point, when the interrupted task is made ready again. If you debug an interrupt routine, do not be confused. This has proven to be the most efficient way of initiating a task switch from within an interrupt service routine.
If fast task-activation is not required, `OS_LeaveInterruptNoSwitch()` can be used instead.

13.2. Calling ***embOS*** routines from within an ISR

OS_EnterInterrupt(), OS_LeaveInterrupt(), OS_LeaveInterruptNoSwitch().

The use of `OS_EnterInterrupt()` informs ***embOS*** that interrupt code is executing and has the following effects:

- disables task-switches
- keeps interrupts in internal routines disabled

If `OS_EnterInterrupt()` is used, it should be the first function to be called in the interrupt handler.

If `OS_EnterInterrupt()` is used, `OS_LeaveInterrupt()` or `OS_LeaveInterruptNoSwitch()` should be the last function to be called in the interrupt handler.
`OS_LeaveInterrupt()` informs ***embOS*** that the end of the interrupt routine is reached. If the interrupt has caused a task switch, it is executed now -unless the program which was interrupted was in a critical region.

`OS_LeaveInterruptNoSwitch()` informs ***embOS*** that the end of the interrupt routine is reached, but does not execute the task switch from within the ISR, but at the next possible occasion. This will be the next call of an ***embOS*** function or the Scheduler Interrupt if the program is not in a critical region.

Examples

Interrupt routine using `OS_EnterInterrupt()` / `OS_LeaveInterrupt()`:

```
__interrupt void ISR_Timer(void) {
  OS_EnterInterrupt();
  OS_SignalEvent(1,&Task);   /* any functionality could be here */
  OS_LeaveInterrupt();
}
```

13.3. Enabling / Disabling interrupts from "C"

During the execution of a task, maskable interrupts are normally enabled. In certain sections of the program however, it can be necessary to disable interrupts for short periods of time to make a section of the program an atomic operation that can not be interrupted. An example would be the access to a global volatile variable of type long on an 8/16 bit CPU:

Bad example

```
volatile long lvar;

void routine (void) {
  lvar ++;
}
```

In order to make sure that the value does not change between the two or more accesses that are needed, the interrupts have to be temporarily disabled.
The problem with disabling and re-enabling interrupts is the following: Functions that disable/enable the interrupt can not be nested.
Your C-compiler offers 2 intrinsic functions for enabling and disabling interrupts. These functions can still be used, but it is recommended you use the functions that ***embOS*** offers (To be precise, they only look like functions, but are macros in reality).
If you do not use this recommended ***embOS*** functions, you may run into a problem if routines which require a portion of the code to run with disabled interrupts are nested or call an OS-routine. We recommend to disable the interrupt only for short periods of time, if possible. Also you should not call routines when interrupts are disabled, because this could lead to long interrupt latency times. If you do this, you may also safely use the compiler provided intrinsics to disable interrupts.

OS_IncDI()

Short for: Increment and disable interrupts
Increments the Interrupt disable counter (OS_DICnt) and disables interrupts.
Defined in RTOS.h:

OS_DecRI()

Short for: Decrement and restore interrupts
Decrements the counter and enables interrupts if the counter reaches 0.

The functions mentioned above are in reality macros, so they use very little space only and execute very fast. It is important that they are used as a pair: `OS_IncDI()` first, then `OS_DecRI()`.

Example

```
volatile long lvar;

void routine (void) {
  OS_IncDI();
  lvar ++;
  OS_DecRI();
}
```

`OS_IncDI()` increments the interrupt disable counter which is used for the entire OS and is therefore consistent with the rest of the program: Any routine can be called, and the interrupts will not be switched on before the matching `OS_DecRI()` has been executed. These 2 functions are actually macros defined in `RTOS.H`. They are very efficient and use no more than a few bytes. However, if you need to disable the interrupts for a short moment only where no routine is called as in the example above, you could also use the pair `OS_DI()` and `OS_RestoreI()`. These are a tiny little bit more efficient because the interrupt disable counter `OS_DICnt` is not modified twice, but only checked once. They do have the disadvantage that they do not work with routines because the status of `OS_DICnt` is not actually changed and should be used with great care. In case of doubt, use `OS_IncDI()` and `OS_DecRI()`.

OS_DI()

Short for **Disable Interrupts**
Disables interrupts. Does not change the interrupt disable counter.

OS_EI()

Short for **Enable Interrupts**
Please refrain from using it directly unless you are sure that the interrupt enable count has the value zero. (Because it does not take the interrupt-disable counter into account)

OS_RestoreI()

Short for **Restore Interrupts**
Restores the status of the interrupt flag, based on the interrupt disable counter.

Example

```
volatile long lvar;

void routine (void) {
  OS_DI();
  lvar ++;
  OS_RestoreI();
}
```

Definitions of interrupt control macros (in RTOS.h)

```
#define OS_IncDI()       { OS_ASSERT_DICnt(); OS_DI(); OS_DICnt++; }
#define OS_DecRI()       { OS_ASSERT_DICnt(); if (--OS_DICnt==0) OS_EI(); }
#define OS_RestoreI()    { OS_ASSERT_DICnt(); if (OS_DICnt==0)   OS_EI(); }
```

13.4. Nesting interrupt routines

For applications requiring short interrupt latency, you may re-enable interrupts inside an interrupt handler. Therefore use `OS_EnterNestableInterrupt()` and `OS_LeaveNestableInterrupt()` within your interrupt handler.
Per default, interrupts are disabled in an interrupt handler (ISR) because the CPU disables interrupts with the execution of the interrupt handler. Re-enabling interrupts in an interrupt handler allows the execution of further interrupts with equal or higher priority than that of the current interrupt. (nesting interrupts)
Nested interrupts can lead to problems that are difficult to track; therefore it is not really recommended to enable the execution of interrupts form within an interrupt handler. As it is important, that ***embOS*** keeps track of the status of the interrupt enable / disable flag, disabling of the interrupt has to be done using the functions that ***embOS*** offers for this purpose. To enable the interrupt in an interrupt handler, use `OS_EnterNestableInterrupt();` you need to use `OS_LeaveNestableInterrupt()` to disable the interrupts right before ending the interrupt routine again in order to restore the default condition. The call of `OS_EnterNestableInterrupt()` prevents further task switches. Re-enabling interrupts will make it possible that an ***embOS***-Scheduler interrupt shortly interrupts this ISR. In this case, ***embOS*** needs to know that an other ISR is still active and it may not perform a task switch.

OS_EnterNestableInterrupt()

Re-enables interrupts and increments the ***embOS*** internal critical region counter, thus disabling further task switches. This function should be the first call inside an interrupt handler, when nested interrupts are required. The function is implemented as a macro and offers the same functionality, as the former `OS_EnterInterrupt()` and `OS_DecRI()`, but is more efficient, which means, it results in smaller and faster code.

OS_LeaveNestableInterrupt()

This function disables further interrupts, then decrements the ***embOS*** internal critical region count, thus re-enabling task switches, if the critical region count reached zero again.
This function is the counterpart of `OS_EnterNestableInterrupt()` and has to be the last function call inside an interrupt handler, when nested interrupts where enabled before by calling `OS_EnterNestableInterrupt()`. The function OS_LeaveNestableInterrupt() is implemented as a macro and offers the same functionality, as the former `OS_IncDI()`in combination with `OS_LeaveInterrupt()`, but is more efficient, which means, it results in smaller and faster code.

```
 __interrupt void ISR_Timer(void) {
  OS_EnterNestableInterrupt(); /* Enable interrupts, but disable task switch*/
  /*
  * any code legal for interrupt-routines can be placed here
  */
  IntHandler();
  OS_LeaveNestableInterrupt();  /* Disable interrupts, allow task switch    */
}
```

13.5. Non maskable interrupts (NMIs)

embOS performs atomic operations by disabling interrupts. Since NMIs can not be masked, they can interrupt these atomic operations. Therefore NMIs should be used with great care and may under no circumstances call any ***embOS*** - routines.

14. Critical Regions

Critical regions are program sections during which the scheduler is switched off, meaning that no task switch and no execution of a software-timer is allowed except for a situation in which the active task has to wait. Effectively preemptions are switched off.
A typical example for a critical region would be the execution of a program section that handles a time critical hardware access, e.g. writing multiple bytes into a EEPROM, where the bytes have to be written in a certain amount of time or a section that writes data into global variables used by a different task and therefore needs to make sure the data is consistent.
A "critical region" can be defined anywhere during the execution of a task. S/W timers and interrupts are executed as critical regions anyhow, so it does not hurt but it does not do any good either to declare a critical region there.
If a task switch becomes due during the execution of a critical region, it will be performed right after the critical region is left.
Critical regions can be nested; the scheduler will be switched on again after the outermost loop is left. Interrupts are still legal in a critical region. However, software-timer will not be executed during a critical region but right after it is left.

14.1. OS_EnterRegion

Description

Indicates to the OS the beginning of a critical region.

Prototype

`void OS_EnterRegion(void);`

Return value

Void.

Add. information

`OE_EnterRegion` is not actually a function but a macro. However, it behaves very much like a function with the difference that is a lot more efficient.
Usage of the macro indicates to ***embOS*** the beginning of a critical region. A critical region counter (`OS_RegionCnt`), which is 0 by default, is incremented, so that the routine can be nested. The counter will be decremented by a call to the routine `OS_LeaveRegion`. If this counter reaches 0 again, the critical region ends.
Interrupts are not disabled using `OS_EnterRegion`; disabling the interrupts will on the other side disable preemptive task switches.

Example

```
void SubRoutine(void) {
  OS_EnterRegion();
  /* this code will not be interrupted by the OS  */
  OS_LeaveRegion();
}
```

14.2. OS_LeaveRegion

Description

Indicates to the OS the end of a critical region.

Prototype:

```
void OS_LeaveRegion(void);
```

Return value

Void.

Add. information

`OS_LeaveRegion` is not actually a function but a macro. However, it behaves very much like a function with the difference that is a lot more efficient.
Usage of the macro indicates to ***embOS*** the end of a critical region. A critical region counter (`OS_RegionCnt`), which is 0 by default, is decremented.
If this counter reaches 0 again, the critical region ends.

Example

Refer to section for `OS_EnterRegion`.

15. System variables

The system variables are described here for a deeper understanding of how the OS works and to make debugging easier.
Please, do not change the value of any system variables.
These variables are accessible and not declared constant, but they should only be altered by functions of ***embOS***. However, some of these variables can be very useful, especially the time variables.

15.1. Time Variables

15.1.1. OS_Time

Description

This is the time variable which contains the current system time in ticks (usually equivalent to ms)

Prototype

```
extern volatile OS_U32 OS_Time;
```

Add. information

The time variable has a resolution of one time unit, which is normally 1/1000 sec and normally the time between two successive calls to the ***embOS*** interrupt handler. Instead of accessing this variable directly, you should do so by using `OS_GetTime()` or `OS_GetTime32()`.

15.1.2. OS_TimeDex

Basically for internal use only. Contains the time at which the next task switch or timer activation is due. If `((int)(OS_Time - OS_TimeDex) >=0)`, the task-list and timer list will be checked for a task or timer to activate. After activation of this timer, `OS_TimeDex` will be assigned the time stamp of the next task or timer to be activated.

15.2. OS internal variables and data-structures

embOS internal variables are not explained here as this is in no way required to use ***embOS***. Your application should not rely on any of the internal variables, as only the documented API functions are guaranteed to remain unchanged in future versions of ***embOS***.

Important

Do not alter any system variables

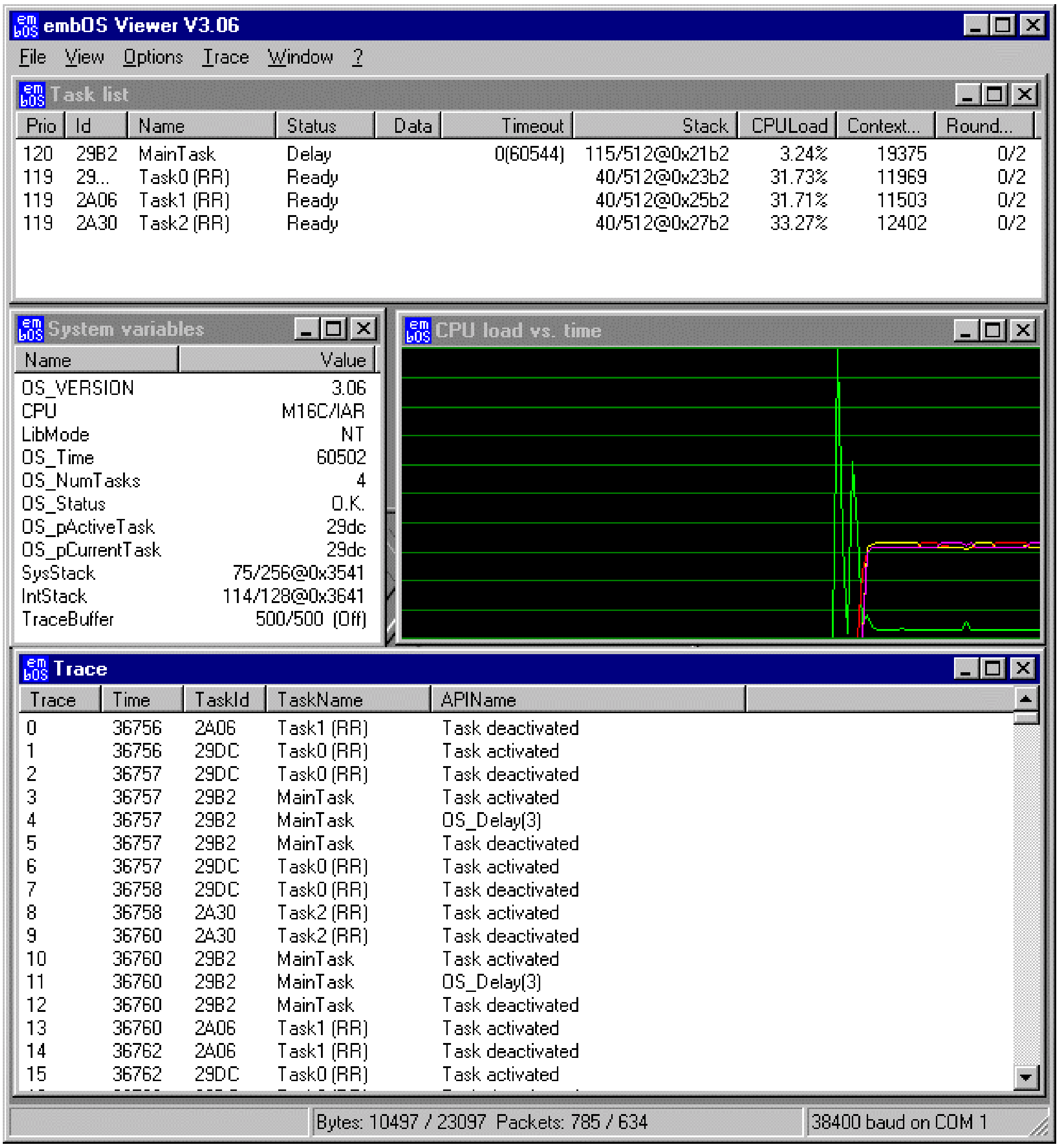
16. STOP / HALT / IDLE Mode

Most CPUs support power saving STOP, HALT or IDLE modes. Usage of these modes is one possibility to save power consumption during idle times. As long as the timer interrupt will wake up the system every embOS tick, or other interrupts will activate tasks, these modes may be used to save power consumption. If required, you may modify the `OS_Idle()` routine, which is part of the hardware dependant module `RtosInit.c` to switch the CPU to power saving mode during idle times. Please check out the *CPU & Compiler Specifics* manual of ***embOS*** documentation for details on your processor.

17. embOSView: Profiling and analyzing

17.1. Overview

embOSView displays the state of a running application using ***embOS.*** A serial interface (UART) is normally used to communicate with the target.
The hardware dependent routines and defines to communicate with embOS-View are located in RTOSInit.c. This file has to be configured properly. For details on how to configure this file, please refer the *CPU & Compiler Specifics* manual of ***embOS*** documentation.
The embOSView utility is shipped as embosView.exe with ***embOS*** and runs under Windows 9x / NT / 2000. The latest version is available on our website www.segger.com



embOSView is a very helpful tool for analysis of the running target application.

17.2. Task list window

embOSView shows the state of every created task of the target application in the Task list window. The information shown depends on the library used in your application

Item	Explanation	Builds
Prio	Actual priority of task	All
Id	Task Id, which is the address of task control block	All
Name	Name given during creation	All
Status	Actual state of task (Executing, delay, waiting etc)	All
Data	Meaning depends on status	All
Timeout	Time of next activation	All
Stack	Used stack size, max. stack size, stack location	S, SP, D, DP, DT
CPULoad	Percentage CPU load caused by task	SP, DP, DT
Context-Switches	Number of activations since reset	SP, DP, DT

The task list window is very helpful in analysis of stack usage and CPU load for every running task.

17.3. System variables

embOSView shows the actual state of major system variables in the system variables window. The information shown also depends on the library used in your application:

Item	Explanation	Builds
OS_VERSION	Actual version of ***embOS***	All
CPU	Target CPU and compiler	All
LibMode	Library mode used for target application	All
OS_Time	Actual system time in timer ticks	All
OS_NumTasks	Actual number of defined tasks	All
OS_Status	shows actual error code (or O.K.)	All
OS_pActiveTask	Active task, that should actually run	SP, D, DP, DT
OS_pCurrentTask	Actual running task	SP, D, DP, DT
SysStack	Used size, max. size and location of system stack	SP, DP, DT
IntStack	Used size, max. size and location of interrupt stack	SP, DP, DT
TraceBuffer	Actual count, maximum size and actual state of trace buffer	all trace builds

17.4. Sharing the SIO for Terminal I/O

The SIO used by embOSView may also be used by the application at the same time for both input and output. This can be very helpful. Terminal input is often used as keyboard input, where terminal output may be used to output debug messages. Input and output is done via the terminal window, which can be shown by menu ‘View | Terminal’
To ensure communication via the terminal window in parallel with the viewer functions, the application has to use the two functions `OS_SendString()` for

sending and `OS_SetRxCallback()` to hook a reception routine, that receives one byte.

OS_SendString

Description

Sends a string over SIO to the terminal window.

Prototype

```
void OS_SendString(const char* s);
```

Parameter	Meaning
`s`	Points to a zero terminated string that should be sent to the terminal

Add. information

This function uses `OS_COM_Send1`() which is defined in RTOSInit.c.

OS_SetRxCallback

Description

Sets a callback hook to a routine for receiving one character.

Prototype

```
typedef void OS_RX_CALLBACK(OS_U8 Data)
OS_RX_CALLBACK* OS_SetRxCallback(OS_RX_CALLBACK* cb)
```

Parameter	Meaning
`cb`	Pointer to the application routine that should be called, when one character is received over serial interface

Return value

OS_RX_CALLBACK* as described above. This is the pointer to the callback function that was hooked before the call.

Add. information

The user function is called from ***embOS***. The received character is passed as parameter. See example below.

Example

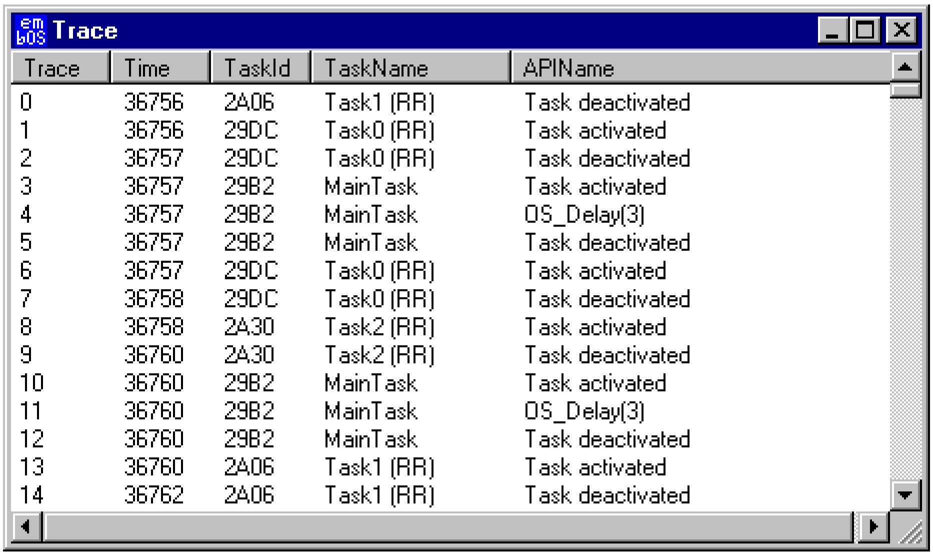
```
void GUI_X_OnRx(OS_U8 Data); /* Callback ... called from Rx-interrupt */

void GUI_X_Init(void) {
  OS_SetRxCallback( &GUI_X_OnRx);
}
```

17.5. Using the API-trace

With ***embOS*** version 3.06 or higher, a trace feature of API call was introduced. This requires the use of the trace build libraries in the target application.
The trace build libraries implement a buffer for 100 trace entries. Tracing of API calls can be started and stopped from embOSView via menu ‘Trace’, or it can

also be started and stopped from within the application by use of the new functions `OS_TraceEnable()` and `OS_TraceDiasable()`.
Individual filters may be defined, to determine which API calls should be traced for different tasks or from within interrupt or timer routines.
Once trace was started, the API calls are recorded in the trace buffer, which is periodically read by embOSView. The result is shown in the Trace window:

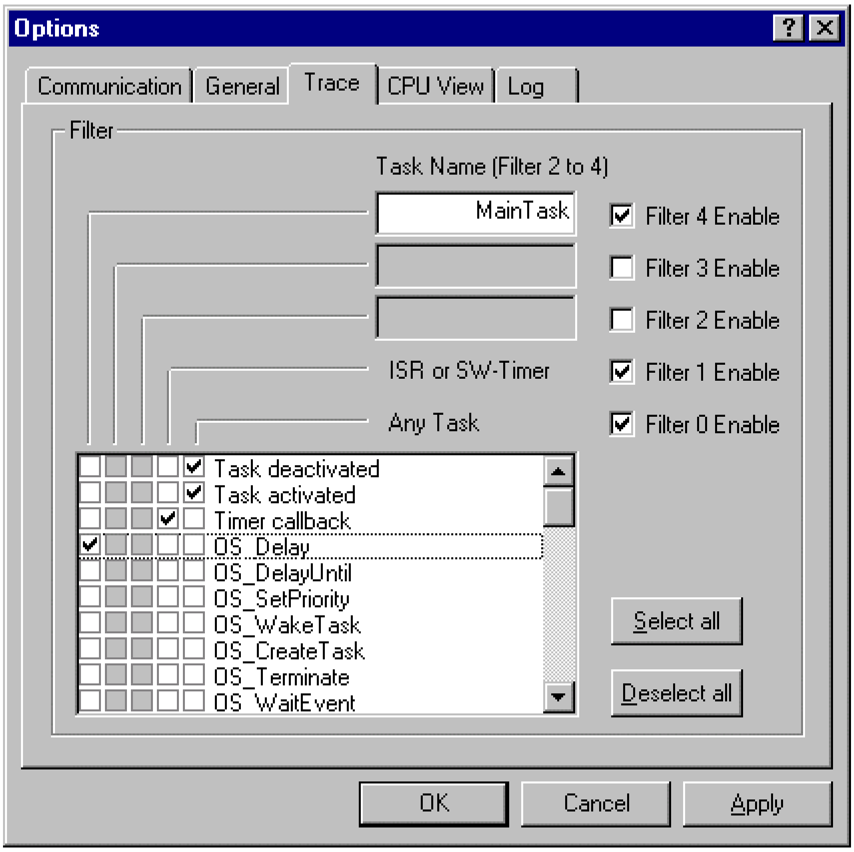

Trace

Trace	Time	TaskId	TaskName	APIName
0	36756	2A06	Task1 (RR)	Task deactivated
1	36756	29DC	Task0 (RR)	Task activated
2	36757	29DC	Task0 (RR)	Task deactivated
3	36757	29B2	MainTask	Task activated
4	36757	29B2	MainTask	OS_Delay(3)
5	36757	29B2	MainTask	Task deactivated
6	36757	29DC	Task0 (RR)	Task activated
7	36758	29DC	Task0 (RR)	Task deactivated
8	36758	2A30	Task2 (RR)	Task activated
9	36760	2A30	Task2 (RR)	Task deactivated
10	36760	29B2	MainTask	Task activated
11	36760	29B2	MainTask	OS_Delay(3)
12	36760	29B2	MainTask	Task deactivated
13	36760	2A06	Task1 (RR)	Task activated
14	36762	2A06	Task1 (RR)	Task deactivated

Every entry in the trace list is recorded with the actual system time. In case of calls or events from tasks, the task ID and task name (limited to 15 characters) is also recorded. Parameters of API calls are recorded if possible and are shown as part of the APIName column. In the example above, this is shown for OS_Delay(3).
Once the trace buffer is full, trace is automatically stopped. The trace list and buffer can be cleared from embOSView.

17.5.1. Setting up trace from embOSView

Three different kinds of trace filter are defined for tracing. These filters can be set up from embOSView via menu ‘Options | Setup | Trace’:
Filter 0 is non task specific and records all specified events regardless of the task. As the Idle loop is no task, calls from within the Idle loop are not traced.
Filter 1 is specific for interrupt service routines, s/w timer and all calls that occur outside a running task. These calls may come from the Idle loop or during startup, when no task is running.
Filter 2 to 4 allow trace of API calls from named tasks.



To enable or disable a filter, simply check or uncheck the corresponding checkbox ‘Filter 0 Enable’ to ‘Filter 4 Enable’.
For any of those five filters, individual API functions can be enabled or disabled by checking or unchecking the corresponding checkboxes in the list.
To speed up the process, there are two buttons available:
‘Select All’ enables trace of all API functions for the actual enabled (checked) filters.
‘Deselect All’ disables trace of all API functions for the actual enabled (checked) filters.
Filter 2 to 4 allow trace of task specific API calls. Therefore a task name can be specified for each of those filters.
In the example above, Filter 4 is configured to trace calls of OS_Delay from the task called ‘MainTask’.
After the settings are saved (via Apply or OK button), the new settings are sent to the target application.

17.6. Trace filter setup functions

Tracing of API or user function calls can be started or stopped from embOS-View. Per default trace is initially disabled in an application program. It may be very helpful to control the recording of trace events directly from the application. This can be done by the following functions:

OS_TraceEnable

Description

Enables trace of actual filtered API calls.

Prototype

```
void OS_TraceEnable(void);
```

Add. information

The trace filter conditions should have been set up before a call of this function. This functionality is available in trace builds only. In none trace builds this API call is removed by the preprocessor.

OS_TraceDisable

Description

Disables trace of API and user function calls.

Prototype

```
void OS_TraceDisable(void);
```

Add. information

This functionality is available in trace builds only. In none trace builds this API call is removed by the preprocessor.

OS_TraceEnableAll

Description

Sets up the first trace filter (Filter 0: ‘Any task’), enables trace of all API calls and then enables trace function.

Prototype

```
void OS_TraceEnableAll(void);
```

Add. information

The trace filter conditions of all the other trace filters are not affected. This functionality is available in trace builds only. In none trace builds this API call is removed by the preprocessor.

OS_TraceDisableAll

Description

Sets up the first trace filter (Filter 0: ‘Any task’), disables trace of all API calls and also disables trace.

Prototype

```
void OS_TraceDisableAll(void);
```

Add. information

The trace filter conditions of all the other trace filters are not affected, but tracing is stopped. This functionality is available in trace builds only. In none trace builds this API call is removed by the preprocessor.

OS_TraceEnableId

Description

Sets the specified id value in the first trace filter (Filter 0: ‘Any task’), thus enabling trace of the specified function, but does not start trace.

Prototype

```
void OS_TraceEnableId(OS_U8 Id);
```

Parameter	Meaning
`Id`	Id value of API call that should be enabled for trace 0 <= Id <= 127 Values from 0 to 99 are reserved for ***embOS***

Add. information

To enable trace of a specific ***embOS*** API function, you have to use the correct Id value. These values are defined as symbolic constants in RTOS.h
This function may also be used to enable trace of your own functions. This functionality is available in trace builds only. In none trace builds this API call is removed by the preprocessor.

OS_TraceDisableId

Description

Resets the specified id value in the first trace filter (Filter 0: ‘Any task’), thus disabling trace of the specified function, but does not stop trace.

Prototype

```
void OS_TraceDisableId(OS_U8 Id);
```

Parameter	Meaning
`Id`	Id value of API call that should be enabled for trace 0 <= Id <= 127 Values from 0 to 99 are reserved for ***embOS***

Add. information

To disable trace of a specific ***embOS*** API function, you have to use the correct Id value. These values are defined as symbolic constants in RTOS.h
This function may also be used to disable trace of your own functions. This functionality is available in trace builds only. In none trace builds this API call is removed by the preprocessor.

OS_TraceEnableFilterId

Description

Sets the specified id value in the specified trace filter, thus enabling trace of the specified function, but does not start trace.

Prototype

```
void OS_TraceEnableFilterId(OS_U8 FilterIndex, OS_U8 id)
```

Parameter	Meaning
`FilterIndex`	Index of the Filter, that should be affected. 0 <= FilterIndex <= 4 0 affects Filter 0 (‘Any Task’) and so on
`id`	Id value of API call that should be enabled for trace 0 <= Id <= 127 Values from 0 to 99 are reserved for ***embOS***

Add. information

To enable trace of a specific ***embOS*** API function, you have to use the correct Id value. These values are defined as symbolic constants in RTOS.h
This function may also be used to enable trace of your own functions. This functionality is available in trace builds only. In none trace builds this API call is removed by the preprocessor.

OS_TraceDisableFilterId

Description

Resets the specified id value in the specified trace filter, thus disabling trace of the specified function, but does not stop trace.

Prototype

```
void OS_TraceDisableFilterId(OS_U8 FilterIndex, OS_U8 id)
```

Parameter	Meaning
FilterIndex	Index of the Filter, that should be affected. 0 <= FilterIndex <= 4 0 affects Filter 0 (‘Any Task’) and so on
id	Id value of API call that should be enabled for trace 0 <= Id <= 127 Values from 0 to 99 are reserved for ***embOS***

Add. information

To disable trace of a specific ***embOS*** API function, you have to use the correct Id value. These values are defined as symbolic constants in RTOS.h
This function may also be used to disable trace of your own functions. This functionality is available in trace builds only. In none trace builds this API call is removed by the preprocessor.

17.7. Trace record functions

The following functions are used to write (record) data into the trace buffer. As long as only ***embOS*** API calls should be recorded, these functions are used internally by the trace build libraries.
If for some reason, you want to trace own functions with own parameters, you may call one of those functions.
All those functions have the following points in common:

- To record data, trace must be enabled.
- An Id value in the range from 100 to 127 has to be used as id parameter. Id values from 0 to 99 are internally reserved for ***embOS***
- The specified events (id’s) have to be enabled in any of the trace filters.
- Active system time and current task are automatically recorded together with the specified event.

OS_TraceVoid

Description

Writes an entry which is only identified by its id into the trace buffer.

Prototype

```
void OS_TraceVoid(OS_U8 id);
```

Parameter	Meaning
`id`	Id value that should be written into trace buffer 100 <= Id <= 127 Values from 0 to 99 are reserved for ***embOS***

Add. information

This functionality is available in trace builds only. In none trace builds this API call is removed by the preprocessor.

OS_TracePtr

Description

Writes an entry with id and a pointer as parameter into the trace buffer.

Prototype

```
void OS_TracePtr(OS_U8 id, void* p);
```

Parameter	Meaning
`id`	Id value that should be written into trace buffer 100 <= Id <= 127 Values from 0 to 99 are reserved for ***embOS***
`p`	any void pointer that should be recorded as parameter

Add. information

The pointer passed as parameter, will be displayed in the trace list window of embOSView. This functionality is available in trace builds only. In none trace builds this API call is removed by the preprocessor.

OS_TraceData

Description

Writes an entry with id and an integer as parameter into the trace buffer.

Prototype

```
void OS_TraceData (OS_U8 id, int v);
```

Parameter	Meaning
`id`	Id value that should be written into trace buffer 100 <= Id <= 127 Values from 0 to 99 are reserved for ***embOS***
`v`	any integer value that should be recorded as parameter

Add. information

The value passed as parameter, will be displayed in the trace list window of embOSView. This functionality is available in trace builds only. In none trace builds this API call is removed by the preprocessor.

OS_TraceDataPtr

Description

Writes an entry with id, an integer and a pointer as parameter into the trace buffer.

Prototype

```
void OS_TraceDataPtr(OS_U8 id, int v, void*p);
```

Parameter	Meaning
`id`	Id value that should be written into trace buffer 100 <= Id <= 127 Values from 0 to 99 are reserved for ***embOS***
`v`	any integer value that should be recorded as parameter
`p`	any void pointer that should be recorded as parameter

Add. information

The values passed as parameter, will be displayed in the trace list window of embOSView. This functionality is available in trace builds only. In none trace builds this API call is removed by the preprocessor.

OS_TraceU32Ptr

Description

Writes an entry with id, a 32 bit unsigned integer and a pointer as parameter into the trace buffer.

Prototype

```
void OS_TraceU32Ptr(OS_U8 id, OS_U32 p0, void*p1);
```

Parameter	Meaning
`Id`	Id value that should be written into trace buffer 100 <= Id <= 127 Values from 0 to 99 are reserved for ***embOS***
`p0`	any unsigned 32 bit value that should be recorded as parameter
`p1`	any void pointer that should be recorded as parameter

Add. information

The values passed as parameter, will be displayed in the trace list window of embOSView. This function may be used to record two pointer. This functionality is available in trace builds only. In none trace builds this API call is removed by the preprocessor.

17.8. Application controlled trace example

As described above, the user application can enable and setup the trace conditions without the need of a connection or command from embOSview. Also the trace record functions can be called from any user function to write data into the trace buffer. Therefore id numbers from 100 to 127 may be used.
This can be very helpful to trace API and user functions just after starting the application at a moment, when the communication to embOSView is not available or setup from embOSView is not complete.

```
#include “RTOS.h”

#ifndef OS_TRACE_FROM_START
  #define OS_TRACE_FROM_START 1
```

```
#endif

/*  Application specific trace id numbers   */
#define APP_TRACE_ID_SETSTATE  100

char MainState;

/*  Sample of application routine with trace   */
void SetState(char* pState, char Value) {
  #if OS_TRACE
    OS_TraceDataPtr(APP_TRACE_ID_SETSTATE, Value, pState);
  #endif
  * pState = Value;
}

/*  Sample main routine, that enables and setup API and function call trace
    from start  */
void main(void) {
  OS_InitKern();
  OS_InitHW();
  #if (OS_TRACE && OS_TRACE_FROM_START)
    /*  OS_TRACE is defined in trace builds of the library          */
    OS_TraceDisableAll();        /*  Disable all API trace calls    */
    OS_TraceEnableId(APP_TRACE_ID_SETSTATE);        /*  User trace  */
    OS_TraceEnableFilterId(APP_TRACE_ID_SETSTATE);  /*  User trace  */
    OS_TraceEnable();
  #endif
  /*  Application specific initilisation  */
  SetState(&MainState, 1);
  OS_CREATETASK(&TCBMain, "MainTask", MainTask, PRIO_MAIN, MainStack);
  OS_Start();   /* Start multitasking  -> MainTask() */
}
```

Note:
The example above shows, how a trace filter can be set up by application. As described earlier, OS_TraceEnableID() sets the trace filter 0, that affects calls from any running task. The first call of SetState() in the example above would not be traced, because there is no task running at that moment. Therefore the additional filter setup routine OS_TraceEnableFilterId() is called with filter 1, which results in trace of calls from outside running tasks.

Per default, embOSView lists all user function traces in the trace list window as ‘Routine’, followed by the specified ID and two parameter as Hex value.
The example above would result in
Routine100(0xabcd, 0x01)
Where 0xabcd is the pointer address and 0x01 is the parameter recorded from OS_TraceDataPtr().

17.9. embOS.ini: User defined functions

In order to be able to use the built-in trace (available in trace builds of embOS) for functions of the application program, embOSView can be customized. This customization is done in the Setup file 'embOS.ini'.
This setup file is parsed at startup of embOSView. It is optional; you will not see an error message if it can not be found.

The following shows a sample embOS.ini file:

```
# File: embOS.ini
#
# embOSView Setup file
#
# embOSView loads this file at startup. It has to reside in the same
# directory as the execuatble itself.
#
# Note: The file is not required in order to run embOSView. You will not get
# an error message if it is not found. However, you will get an error message
# if the contents of the file are invalid.


#
# Define add. API functions.
# Syntax: API( <Index>, <Routinename> [parameters])
# Index: Integer, between 100 and 127
# Routinename: Identifier for the routine. Should be no more than 32
characters
# parameters: Optional paramters. A max. of 2 parameters can be specified.
#              Valid parameters are:
#                int
#                ptr
#             Every parameter has to be proceeded by a colon.
#
API( 100, "Routine100")
API( 101, "Routine101", int)
API( 102, "Routine102", int, ptr)
```

17.9.1. Defining User functions for trace

To enable trace setup for user functions, embOSView needs to know an id number, the function name and the type of two optional parameters that can be traced.
The format is explained in the sample file above.

18. Debugging

18.1. Run-time errors

Some error-conditions can be detected during runtime. These are:

- Usage of uninitialized data structures
- Invalid pointers
- Resource unused that has not been used by this task before
- `OS_LeaveRegion` called more often than `OS_EnterRegion`
- stack-overflow (This feature is not available for some processors)

Which run-time errors can be detected depends on how much checking is performed. Unfortunately, additional checking costs memory and speed (It is not really significant, but there is a difference).

If ***embOS*** detects a run-time error, it calls the routine

```
void OS_Error(int ErrCode);
```

This routine is shipped in source as part of the module `RTOSINIT.C.` The routine simply disables further tasks switches and then after re-enabling interrupts loops forever as follows:

```
/*
   Run-time error reaction
*/

void OS_Error(int ErrCode) {
  OS_EnterRegion();     /* Avoid further task switches */
  OS_DICnt =0;          /* Allow interrupts so we can communicate */
  OS_EI();
  OS_Status = ErrCode;
  while (OS_Status);
}
```

In case you are using embOSView, you can see value and meaning of OS_Status in the system variable window.

When using an emulator you should set a breakpoint at the beginning of this routine or simply stop the program after a failure. The error code is passed to the function as parameter.
You can modify the routine to accommodate your own hardware; this could mean that your target-hardware sets an error-indicating LED or shows a little message on the display.

Important

When modifying the `OS_Error()` routine, the first statement needs to be the disabling of scheduler via `OS_EnterRegion()`; the last statement needs to be the infinite loop.

If you look at the `OS_Error()` routine, you will see that it is more complicated than necessary. 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 using your in circuit-emulator, 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.

18.2. List of error codes

Value	Symbolic name	Explanation
120	`OS_ERR_STACK`	stack overflow or invalid stack
128	`OS_ERR_INV_TASK`	task control block invalid or not initialized or overwritten
129	`OS_ERR_INV_TIMER`	timer control block invalid or not initialized or overwritten
130	`OS_ERR_INV_MAILBOX`	mailbox control block invalid or not initialized or overwritten
132	`OS_ERR_INV_CSEMA`	control block for counting semaphore invalid or not initialized or overwritten
133	`OS_ERR_INV_RSEMA`	control block for resource semaphore invalid or not initialized or overwritten
135	`OS_ERR_MAILBOX_NOT1`	One of the following 1 byte mailbox functions has been used on a multi byte mailbox: `OS_PutMail1()`, `OS_PutMailCond1()`, `OS_GetMail1()`, `OS_GetMailCond1()`
140	`OS_ERR_MAILBOX_NOT_IN_LIST`	The mailbox is not in the list of mailboxes as expected. Possible Reasons: a) one mailbox data structure overwritten
142	`OS_ERR_TASKLIST_CORRUPT`	The OS internal tasklist is destroyed
150	`OS_ERR_UNUSE_BEFORE_USE`	`OS_Unuse()` has been called before `OS_Use()`
151	`OS_ERR_LEAVEREGION_BEFORE_ENTERREGION`	`OS_LeaveRegion()` has been called before `OS_EnterRegion()`
152	`OS_ERR_LEAVEINT`	Error in `OS_LeaveInterrupt()`

153	`OS_ERR_DICNT`	The interrupt disable counter (`OS_DICnt`) is out of range (0-15). The counter is affected by the following API calls:
		`OS_IncDI()`
		`OS_DecRI()`
		`OS_EnterInterrupt()`
		`OS_LeaveInterrupt()`

154	`OS_ERR_INTERRUPT_DISABLED`	`OS_Delay()` or `OS_DelayUntil()` called from inside a critical region with interrupts disabled
160	`OS_ERR_ILLEGAL_IN_ISR`	Illegal function call in interrupt service routine: A routine that may not be called from within an ISR has been called from within an ISR.
161	`OS_ERR_ILLEGAL_IN_TIMER`	Illegal function call in interrupt service routine: A routine that may not be called from within a software timer has been called from within a Timer.
170	`OS_ERR_2USE_TASK`	Task control block has been initialized by calling a create function twice.
171	`OS_ERR_2USE_TIMER`	Timer control block has been initialized

Value	Symbolic name	Explanation
		by calling a create function twice.
172	`OS_ERR_2USE_MAILBOX`	Mailbox control block has been initialized by calling a create function twice.
173	`OS_ERR_2USE_BSEMA`	Binary semaphore has been initialized by calling a create function twice.
174	`OS_ERR_2USE_CSEMA`	Counting semaphore has been initialized by calling a create function twice.
175	`OS_ERR_2USE_RSEMA`	Resource semaphore has been initialized by calling a create function twice.

The latest version of defined error table is part of the comment just before the `OS_Error()` function declaration in the source file `RtosInit.c`

19. Supported development tools

embOS has been developed with and for a specific C-Compiler version for the selected target processor. Please check the file RELEASE.HTML for details. It works with the specified C-Compiler only, since other C-Compiler's may use different calling conventions (incompatible object file formats) and therefore might be incompatible. However, if you prefer to use a different C-Compiler, please contact us, we will do our best to satisfy your needs in the shortest possible time.

19.1.Reentrance

All routines, that can be used from different tasks at the same time have to be fully reentrant. A routine is in use, from the moment when it is being called until it returns or the task that has called it is terminated.
All routines supplied with your real-time operating system are fully reentrant. If for some reason you have to have routines that are non - reentrant in your program that can be used from more than one task, it is recommended to use a resource-semaphore to avoid this kind of problem.

C-Routines and reentrance

Normally, the "C"-Compiler generates code that is fully reentrant. However, the compiler has options that force it to generate non-reentrant code (in order to optimize the performance of the compiler). It is recommended not to use these options; but it is possible under certain circumstances.

Assembly routines and reentrance

As long as assembly-functions access local variables and parameters only, they are fully reentrant. Everything else has to be thought about carefully.

20. Limitations

Max. no. of tasks	limited by avail. RAM only
Max. no. of priorities	limited by avail. RAM only
Max. no. of semaphore	limited by avail. RAM only
Max. no. of mailboxes	limited by avail. RAM only
Max. no. of queues	limited by avail. RAM only
Max. size. of queue	limited by avail. RAM only
Max. no. of timer	limited by avail. RAM only
Event flags :	8 bit / task

If you miss additional functions, we appreciate your feedback and will do our best to implement these functions if they fit into the concept.
Do not hesitate to contact us. If you need to make changes to ***embOS***, the full source-code is available.

21. Source code of kernel and library

embOS is available in two versions:

1. Object version: Object code + h/w init source
2. Full source version: Full sources

Since this is the document that describes the object version, the internal data structures are not explained in detail. The object version offers the full functionality of ***embOS*** including all supported memory models of the compiler, the debug libraries as described and the source code for idle task and hardware init. However, the object version does not allow source level debugging of the library routines and the kernel.

The full source version gives you the ultimate options: ***embOS*** can be recompiled for different data sizes; different compile options give you full control of the generated code, making it possible to optimize the system for versatility or minimum memory requirements. You can debug the entire system and even modify it for new memory models or other CPUs.

21.1. Building ***embOS*** libraries

The ***embOS*** libraries can only be built, if you have purchased a source code version of ***embOS***.
In the root path of ***embOS***, you will find a DOS batch file PREP.BAT, which needs to be modified to match the installation directory of your C compiler. Once this work is done, you can call the batch file M.BAT to build all ***embOS*** libraries for your CPU.
The build process should run without any error or warning message. If the build process reports any problem please check the following:

- Are you using the same compiler version as mentioned in the file RELEASE.HTML ?
- Can you compile a simple test file after running PREP.BAT and does it really use the compiler version you have specified ?
- Is there anything mentioned about possible compiler warnings in the RELEASE.HTML ?

If you still have a problem, please let us know.

22. Additional modules

22.1. Keyboard-Manager: KEYMAN.C

Keyboard-driver module supplied in "C". It serves both as example and as a module that can actually be used in your application. The module can be used in most applications with only little changes to the hardware-specific part. It needs to be initialized on startup and creates a task that checks the keyboard 50 times per second.

Changes req. for your hardware

void ReadKeys(void);

How to implement into your program

<u>Example</u>

```
void main(void) {
  OS_InitKern();        /* initialize OS (should be first !)          */
  OS_InitHW();          /* initialize Hardware for OS (see RtosInit.c)*/
  /* You need to create at least one task here ! */
  OS_CREATETASK(&TCB0, "HP Task", Task0, 100, Stack0);  /*Create Task0*/
  OS_CREATETASK(&TCB1, "LP Task", Task1,  50, Stack1);  /*Create Task1*/
  InitKeyMan();         /*  Initialize keyboard manager               */
  OS_Start();
}
```

22.2. Additional libraries and modules

For all ***embOS*** compatible real time operating systems, there are additional libraries and modules available. However, these modules can also be used without ***embOS*** or with a different operating system.

Since these libraries are written in ANSI-"C", they can be used for any target CPU that an ANSI-"C" Compiler exists for. In general, these modules are highly optimized for both low memory consumption (especially in RAM) and high speed.
The modules can be scaled for optimum performance at minimum memory consumption using compile-time switches. Unused portions of the modules are not even compiled, your program stays lean and fast.

emWin

The complete solution for graphical LCDs
fully scaleable graphical user interface featuring:
different fonts, (from 4*6 to 16*32)
line drawing, bitmap drawing,
advanced drawing : (e.g. Circles)
display routines for strings, decimal, hexadecimal, binary values, multiple windows
ultra-fast, yet still very compact (typical Between 8 and 20 kB ROM)
Everything you need for graphic displays !
Any LCD * Any LCD-controller * Any CPU
(monochrome and color version available, Bitmapconverter, Fontconverter, PC-Simulation and Viewer … Check out our website !)

emLoad

Boot-loader software

23. FAQ (frequently asked questions)

Q : Can I implement different priority scheduling algorithms ?
A : Yes, the system is fully dynamic, which means that task-priorities can be changed while the system is running (Using `OS_SetPriority`). This feature can be used to change priorities in a way that basically every desired algorithm can be implemented. One way would be to have a task control task with a priority higher than that of all other tasks that dynamically changes priorities. Normally, the priority controlled round-robin algorithm is perfect for real-time applications.

Q : Can I use a different interrupt source for ***embOS*** ?
A : Yes, any periodical signal can be used, i.e. any internal timer, but it could also be an external signal.

Q : What interrupt priorities can I use for the interrupts my program uses ?
A : Any.

24. Glossary

Some technical terms used in this manual are explained below.

Active Task	Only one task can execute at any given time. The Task currently executing is called the active task
CPU	Central Processing Unit. The "brain" of a microcontroller
ISR	Interrupt service routine. The routine that is called automatically by the processor when an interrupt is acknowledged. ISR have to preserve the entire context of a task, i.e. all registers.
NMI	non maskable interrupt Interrupts that can not be masked (disabled) by software. Example Watchdog timer interrupt.
Processor	Short for microprocessor. The CPU core of a controller
Priority	Every task in an RTOS has a priority. Tasks with higher priority are preferred by the scheduler.
Resource	anything in the computer system of limited availability : e.g. memory, timers, computation time
RTOS	Real time operating system
Scheduler	The program section of an RTOS that selects the active task
Task	program running on a processor. A multi-tasking system allows multiple tasks to execute independently from one another.
TICK	The OS timer interrupt. Usually equals 1 ms.
Timeslice	The time (number of ticks) which a task will be executed until a round robin task change may occur

25. Index