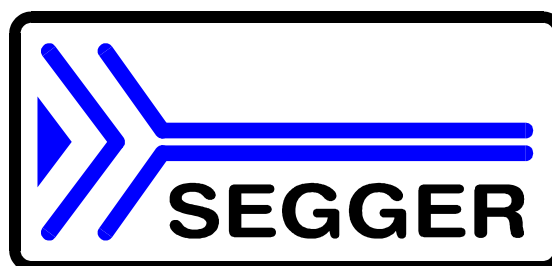# embOS

Real Time Operating System

Software Version 3.30

CPU independent

User's & reference manual

Document revision 5

A product of SEGGER Microcontroller Systeme GmbH

Disclaimer

The information in this document is subject to change without notice. While the information herein is assumed to be accurate, SEGGER MICROCONTROLLER SYSTEME GmbH (the manufacturer) assumes no responsibility for any errors or omissions.

The author makes and you receive no warranties or conditions, express, implied, statutory or in any communications with you. The manufacturer specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the manufacturer. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license. If you have received this product as a trial version for evaluation, you are entitled to evaluate it, but you may under no circumstances use it in a product. If you want to do so, you must obtain a fully licensed version from the manufacturer.

© 1996 - 2005 Segger Microcontroller Systeme GmbH

Trademarks

Names mentioned in this manual may be trademarks of their respective companies. Brand and product names are trademarks or registered trademarks of their respective holders.

Registration

Please register the software via email. This way we can make sure you will receive updates or notifications of updates as soon as they become available. For registration please provide the following information:

- Company name and address
- Your name
- Your job title
- Your email address and telephone number
- Name and version of the product

Please send this information to: register@segger.com.

Contact address

SEGGER Microcontroller Systeme GmbH
Heinrich-Hertz-Strasse 5
D-40721 Hilden
Germany
Tel.:   +49-2103-2878-0
Fax: +49-2103-2878-28
Email :   support@segger.com
Internet: http://www.segger.com/

Software and manual versions

This manual describes the software version 3.26. If any error occurs, please inform us and we will try to assist you as soon as possible.

For further information on topics or routines not yet specified, please contact us.

Print date: 22.05.2006

Software	Manual	Date	By	Explanation
3.30	5	060519	AW	Chapter 4.8: OS_Suspend() modified. Passing a NULL pointr is allowed since version 3.30b of embOS. Chapter 16.1.1: Data type of variable OS_Time corrected
3.28	4	051109	AW	Chapter 7: OS_SignalCSemaMax() function added. Chapter14: Explanation of interrupt latencies and high / low priorities added
3.28	3	050926	AW	Chapter 6: OS_DeleteRSema() function added.
3.28	2	050707	AW	Chapter 4: OS_GetSuspendCnt() function added.
3.28	1	050425	AW	Version number changed to 3.28 to fit current embOS version. Chapter 18.1.2; Type of return value of OS_GetTime32() corrected.
3.26		050209	AW	Chapter 4: OS_Terminate() modified due to new features of version 3.26. Chapter 24: Source code version: additional compile time switches and build process of libraries explained more in detail.
3.24		041115	AW	Chapter 6: Some prototype declarations showed type OS_SEMA instead of OS_RSEMA. Corrected.
3.22	1	040816	AW	Chapter 8: New Mailbox functions added OS_PutMailFront() OS_PutMailFront1() OS_PutMailFrontCond() OS_PutMailFrontCond1()
3.20	5	040621	RS AW	Software timers: Maximum timeout values and OS_TIMER_MAX_TIME described. Chapter 14: Description of rules for interrupt handlers revised. OS_LeaveNestableInterruptNoSwitch() added which was not described before.
3.20	4	040329	AW	OS_CreateCSema() prototype declaration corrected. Return type is void. OS_Q_GetMessageCnt() prototype declaration corrected. OS_Q_Clear() function description added. OS_MEMF_FreeBlock() prototype declaration corrected.
3.20	3	040204	AW	OS_CREATEMB() Range for parameter MaxnofMsg corrected. Upper limit is 65535, but was declared 65536 in previous manuals.
3.20	2	031128	AW	Code samples modified: Task stacks defined as array of int, because most CPUs require alignment of stack on integer aligned addresses.
3.20	1	031016	AW	Chapter 4: Type of task priority parameter corrected to unsigned char. Chapter 4: OS_DelayUntil(): Sample program modified. Chapter 4: OS_Suspend() added. Chapter 4: OS_Resume() added. Chapter 5: OS_GetTimerValue(): Range of return value corrected. Chapter 6: Sample program for usage of resource semaphores modified. Chapter 6: OS_GetResourceOwner(): Type of return value corrected. Chapter 8: OS_CREATEMB(): Types and valid range of parameter corrected. Chapter 8: OS_WaitMail() added Chapter 10: OS_WaitEventTimed(): Range of timeout value specified.
3.12	1	021015	AW	Chapter 8: OS_GetMailTimed() added Chapter 11 (Heap type memory management) inserted Chapter 12 (Fixed block size memory pools) inserted
3.10	3	020926 020924 020910	KG KG KG	Index and glossary revised. Section 16.3 (Example) added to Chapter 16 (Time-related routines). Revised for language/grammar. Version control table added. Screenshots added: superloop, cooperative/preemptive multitasking, nested interrupts, low-res and hi-res measurement. Section 1.3 (Typographic conventions) changed to table. Section 3.2 added (Single-task system). Section 3.8 merged with section 3.9 (How the OS gains control). Chapter 4 (Configuration for your target system) moved to after Chapter 15 (System variables). Chapter 16 (Time-related routines) added.

Contents

1. About this document

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

1.1. Assumptions

This document 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" programming language
- The target processor
- DOS command line

If you feel that your knowledge of "C" is not sufficient, we recommend The C Programming Language by Kernighan and Richie (ISBN 0-13-1103628), which describes the standard in C-programming and, in newer editions, also covers the ANSI "C" standard.

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- and compiler-independent introduction to *embOS* and to be a reference for all *embOS* API functions.
For a quick and easy startup with *embOS*, please refer to Chapter 2 in the *CPU & Compiler Specifics* manual of *embOS* documentation, which includes a step-by-step introduction to using *embOS*.

1.3. Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
Keyword	Text that you enter at the command-prompt or that appears on the display (i.e. system functions, file- or pathnames).
Parameter	Parameters in API functions.
Sample	Sample code in program examples.
Emphasis	Very important sections.
Term	Important terms.

2. Introduction to *embOS*

2.1. What is *embOS*?

embOS is a priority-controlled multitasking system, designed to be used as an 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, as well as high speed and versatility.

2.2. Features

Throughout the development process of *embOS*, the limited resources of microcontrollers have always been kept in mind. The internal structure of the real time operating system (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 well worth the time and effort 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. The minimum memory consumption is little more than 1 kb of 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 such as semaphores, mailboxes and events.

Some features of *embOS* include:

- Preemptive scheduling:
  Guarantees that of all tasks in READY state the one with the highest priority executes, except for situations where priority inversion applies.
- Round-robin scheduling for tasks with identical priorities.
- Preemptions can be disabled for entire tasks or for sections of a program.
- Up to 255 priorities:
  Every task can have an individual priority $\Rightarrow$ the response of tasks can be precisely defined according to the requirements of the application.
- Unlimited number of tasks
  (limited only by the amount of available memory).
- Unlimited number of semaphores
  (limited only by the amount of available memory).
- 2 types of semaphores: resource and counting.
- Unlimited number of mailboxes
  (limited only by the amount of available memory).
- Size and number of messages can be freely defined when initializing mailboxes.
- Unlimited number of software timers
  (limited only by the amount of available memory).
- 8-bit events for every task.
- Time resolution can be freely selected (default is 1ms).
- Easily accessible time variable.
- Power management:
- Unused calculation time can automatically be spent in halt mode $\Rightarrow$ power-consumption is minimized.
- Full interrupt support:
  Interrupts can call any function except those that require waiting for data, as well as 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 $\Rightarrow$ short interrupt latency time.
- Nested interrupts are permitted.
- *embOS* has its own interrupt stack (usage optional).
- Frame application for an 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 (an RTOS), only one task can be executed by the CPU at a time. This is called a single-task system. A real time operating system allows the execution of multiple tasks on a single CPU. All tasks execute as if they completely "owned" the entire CPU. The tasks are scheduled, meaning that the RTOS can activate and deactivate every task.

3.2. Single-task system (superloop)

A *superloop* application is basically a program that runs in an endless loop, calling OS functions to execute the appropriate operations (task level). No real time kernel is used, so *interrupt service routines* (ISRs) must be used for real time parts of the software or critical operations (interrupt level). This type of system is typically used in small, uncomplex systems or if real time behavior is not critical.
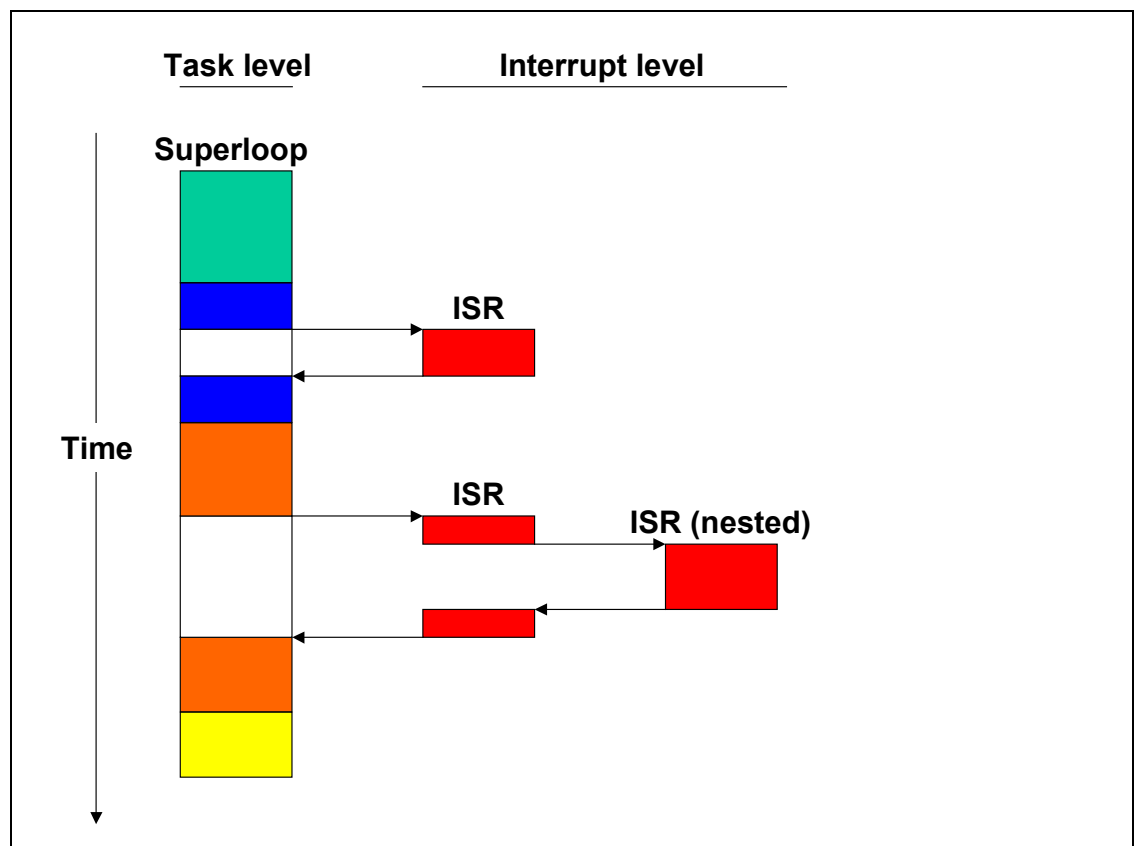
Of course, there are fewer preemption and synchronization problems with a superloop application. Also, since no real time kernel is used, only one stack exists in ROM, meaning that ROM size is smaller and less RAM is used up for stacks. However, superloops can become difficult to maintain if the program becomes too large. Since one software component cannot be interrupted by another component (only by ISRs), the reaction time of one component depends on the execution time of all other components in the system. Real time behavior is therefore poor.

3.3. Multitasking systems

There are different scheduling systems in which the calculation power of the CPU can be distributed among tasks.

3.3.1. Cooperative multitasking

Cooperative multitasking 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", which means that other tasks have no chance of being executed by the CPU while the first task is being carried out. This is illustrated in the diagram below. Even if an ISR makes a higher priority task ready to run, the interrupted task will be returned to and finished before the task switch is made.
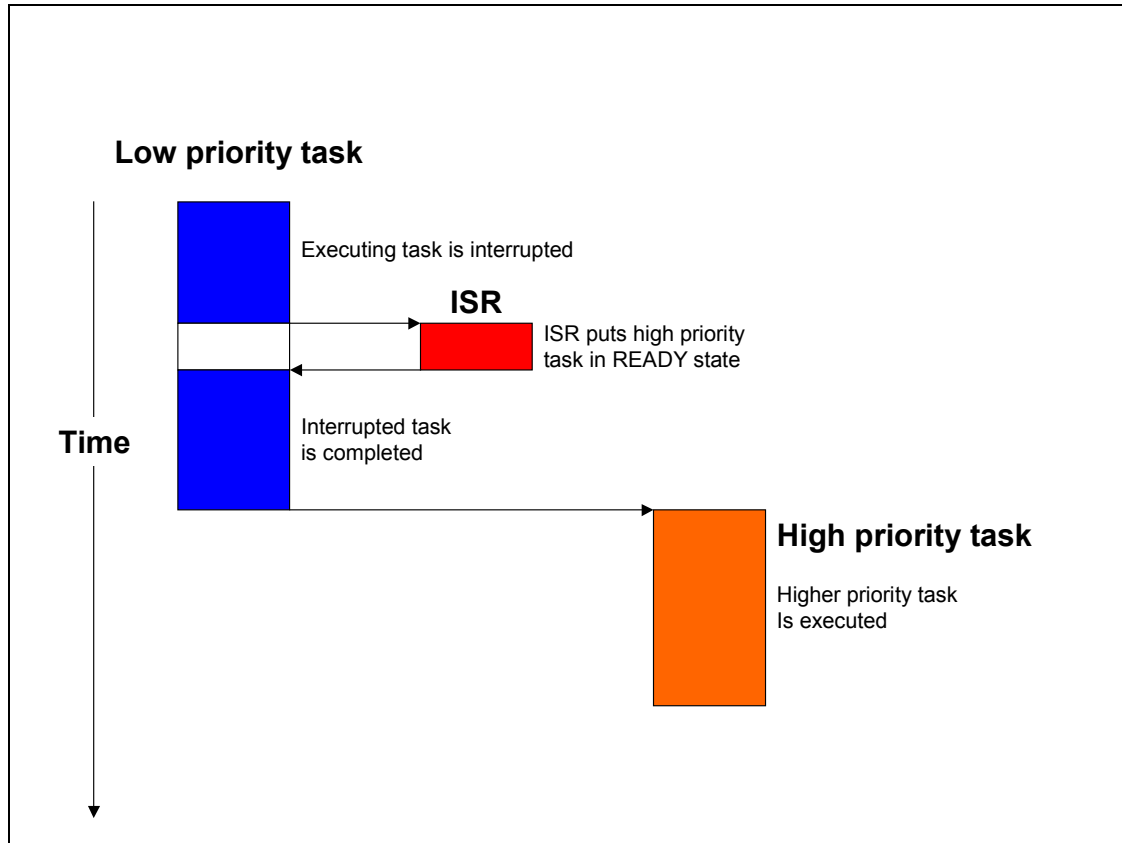
3.3.2. Preemptive multitasking

Real time systems like **embOS** operate 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. The highest-priority task in the READY state is therefore always executed, whether it is an interrupted task or not. If an ISR makes a higher priority task ready, a task switch will occur and the task will be executed before the interrupted task is returned to.
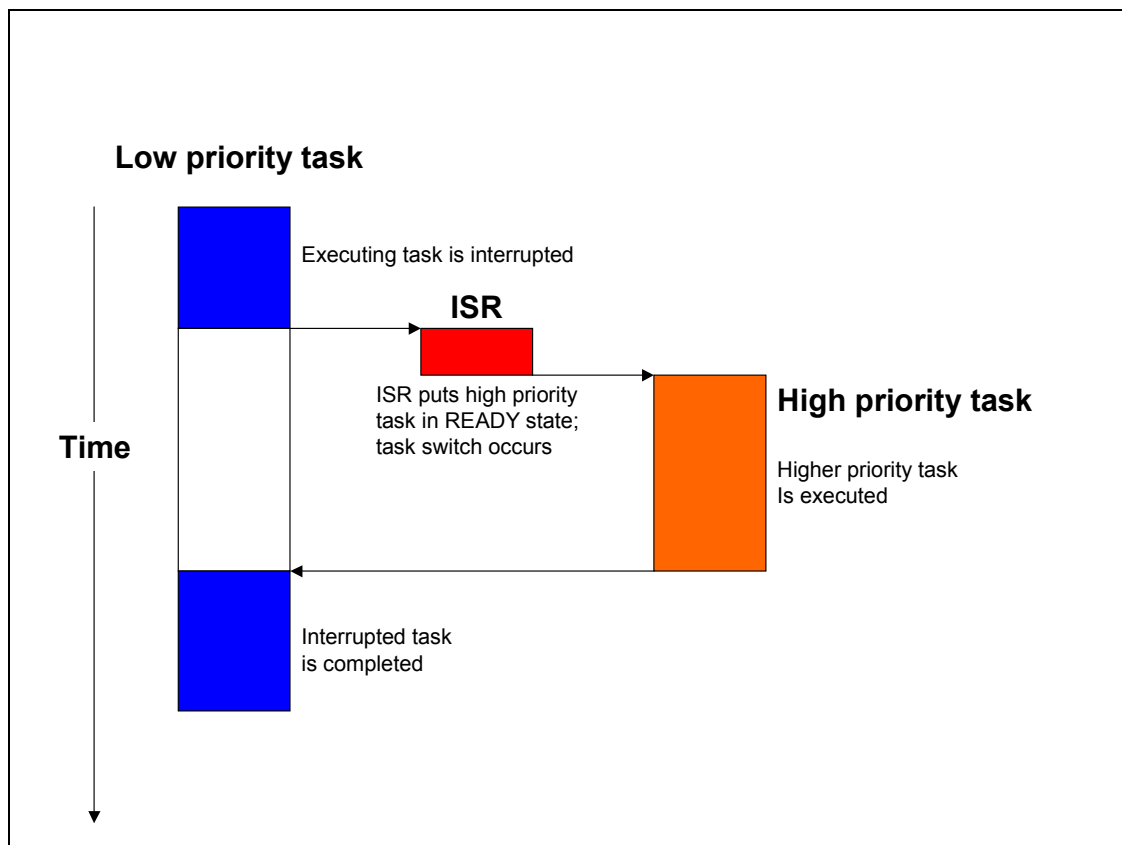
3.4. Scheduling

There are different algorithms that determine which task to execute, called *schedulers*. 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 sema-phore, waiting for event, etc.). The scheduler selects one of the tasks in the READY state and activates it (executes the program of this task). The task which is currently executing is referred to as the *active task*. The main differ-ence between schedulers is in how they distribute the computation time be-tween the tasks in READY state.

3.4.1. Round-robin scheduling algorithm

With round-robin scheduling, the scheduler has a list of tasks and, when deac-tivating the active task, activates the next task that is in the READY state. Round-robin can be used with either preemptive or cooperative multitasking. It works well if you do not need to guarantee response time, if the response time is not an issue of importance, or if all tasks have the same priority. Round-robin scheduling can be illustrated as follows:
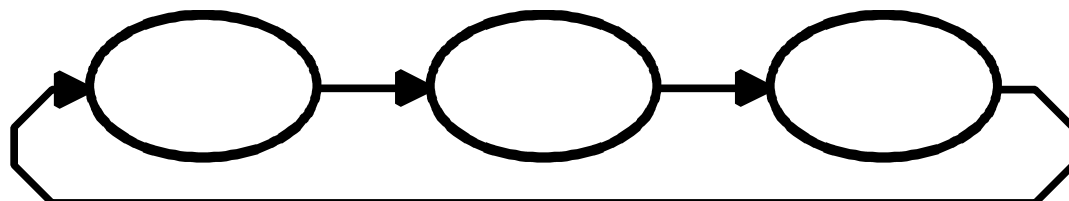
All tasks are on the same level; the possession of the CPU changes periodically after a predefined execution time. This time is called *timeslice*, and may be de-fined individually for every task.

3.4.2. Priority-controlled scheduling algorithm

In real-world applications, different tasks require different response times. For example, in an application that controls a motor, a keyboard and a display, the motor usually requires faster reaction time than the 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 cannot guarantee a specific reaction time, an improved algorithm should be used.

In priority-controlled scheduling, every task is assigned a *priority*. The order of execution depends on this priority. The rule is very simple:
The scheduler activates the task that has the highest priority of all tasks in the 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, known as *critical regions*.

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 whether one task is more important than another. Tasks with identical priority cannot block each other for longer than their timeslices. But round-robin scheduling also costs time if two or more tasks of identical priority are ready and no task of higher priority is ready, because it will constantly switch between the identical-priority tasks. It is more efficient to assign a different priority to each task, which will avoid unnecessary task switches.

3.4.3. Priority inversion

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

But what happens if the highest-priority task is blocked because it is waiting for a resource owned by a lower-priority task? According to the above rule, it would wait until the low-priority-task becomes 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 task that is blocking the highest-priority task gets assigned the highest priority until it releases the resource, unblocking the task which originally had highest priority. This is known as *priority inversion*.

3.5. Communication between tasks

In a *multitasking* (multithreaded) program, multiple tasks work completely separately. But since they all work in the same application, it will sometimes be necessary for them to communicate information to one another.

3.5.1. Global variables

The easiest way to do this is by using global variables. In certain situations, it can make sense for tasks to communicate via global variables, but most of the time this method 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 and power, and the reaction time depends on how often you poll.

3.5.2. Communication mechanisms

When multiple tasks work with one another, they often have to:

- exchange data,
- synchronize with another task, or
- make sure that a resource is used by no more than one task at a time.

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

Mailboxes and queues

A *mailbox* is basically a data buffer managed by the RTOS and is used to send a message to a task. It works without conflicts even if multiple tasks and interrupts try to access it simultaneously. *embOS* also automatically activates any task that is waiting for a message in a mailbox the moment it receives new data and, if necessary, automatically switches to this task.

A *queue* works in a similar manner, but is used to send one or more messages to a task. Queues can handle larger messages than mailboxes, and those messages can be different sizes.
For more information, see Chapter 8: "Mailboxes" and Chapter 9: "Queues".

Semaphores

Two types of *semaphores* are used to synchronize tasks and to manage resources. The most common are *resource semaphores*, although *counting semaphores* are also used. For details and samples, please refer to Chapter 6: "Resource Semaphores" and Chapter 7: "Counting Semaphores". Samples can also be found on our website at www.segger.com.

Events

A task can wait for a particular *event* without using any calculation time. The idea is as simple as it is convincing; there is no sense in polling if we can simply activate a task the moment the event that it is waiting for occurs. This saves a great deal of calculation power and ensures that the task can respond to the event without delay. Typical applications for events are those where a task waits for data, a pressed key, a received command or character, or the pulse of an external real time clock.
For further details, refer to the Chapter 10: "Events".

3.6. How task-switching works

A real time multitasking system lets multiple tasks run like multiple single-task programs, quasi-simultaneously, 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 *stack* has the same function as in a single-task system: storage of return addresses of function calls, parameters and local variables, and temporary storage of intermediate calculation results and register values. Each task can have a different stack size. More information can be found in Chapter 11: "Stacks".

The task control block (TCB) is a data structure assigned to a task when it is created. It contains status information of the task, including the stack pointer, task priority, current task status (ready, waiting, reason for suspension, etc.) and other management data. This information allows an interrupted task to continue execution exactly where it left off. TCBs are only accessed by the RTOS.

3.7. Switching stacks

The following diagram demonstrates the process of switching from one stack to another.

The scheduler deactivates the task to be suspended (Task 0) by saving the processor registers on its stack. It then activates the higher-priority task (Task n) by loading the stack pointer (SP) and the processor registers from the values stored on Task n's stack.
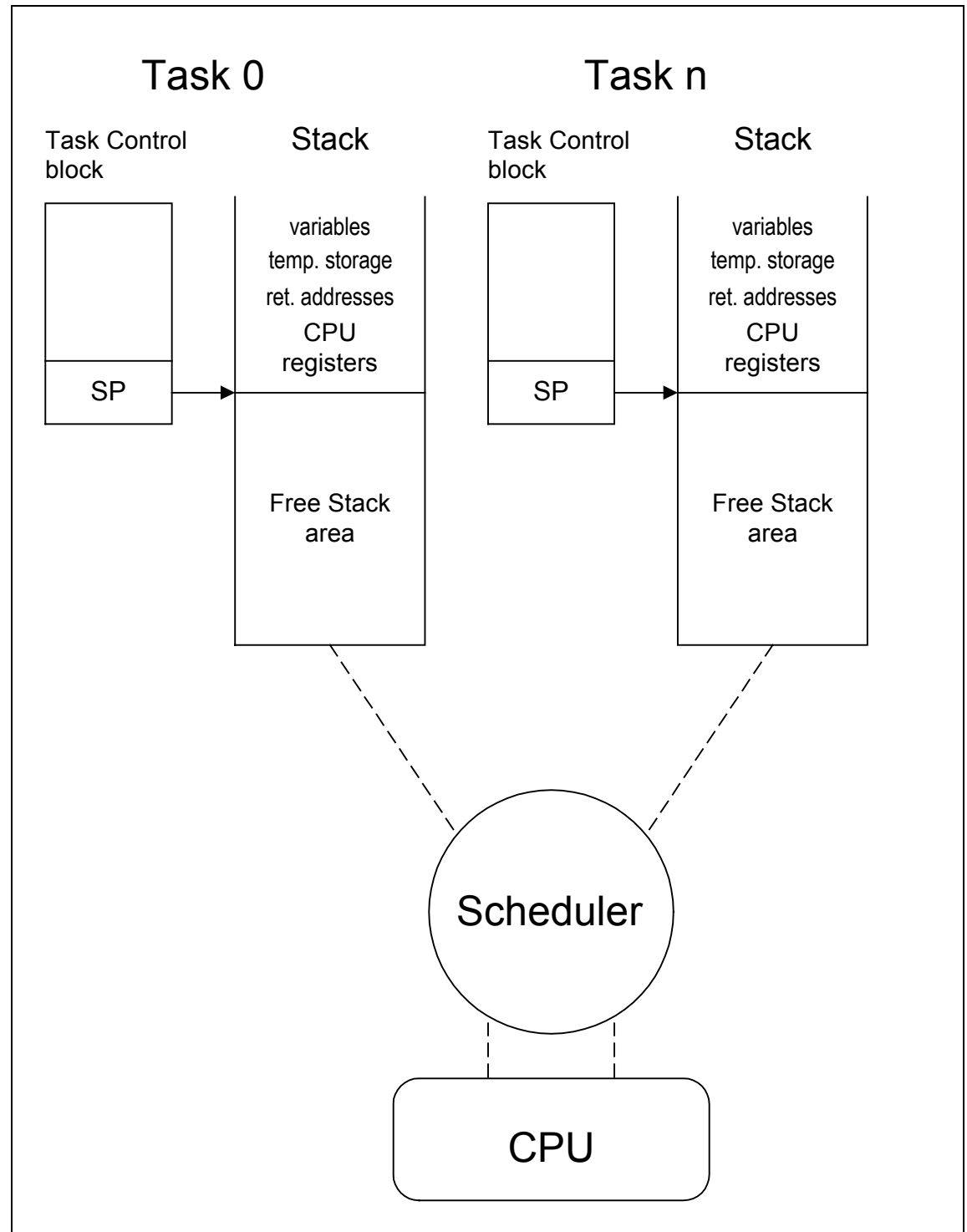
3.8. Change of task status

A task may be in one of several states at any given time. When a task is created, it is automatically put into the READY state (TS_READY).

A task in the READY state is activated as soon as there is no other READY task with higher priority. Only one task may be active at a time. If the task is deactivated or a task with higher priority becomes READY, the active task is simply placed back into the READY state.

The active task may be delayed for or until a specified time; in this case it is put into the DELAY state (TS_DELAY) and the next highest priority task in the READY state is activated.

The active task may also have to wait for an event (or semaphore, mailbox, or queue). If the event has not yet occurred, the task is put into the waiting state and the next highest priority task in the READY state is activated.

A non-existent task is one that is not yet available to *embOS*; it has either not been created yet or it has been terminated.

The following illustration shows all possible task states and transitions between them.
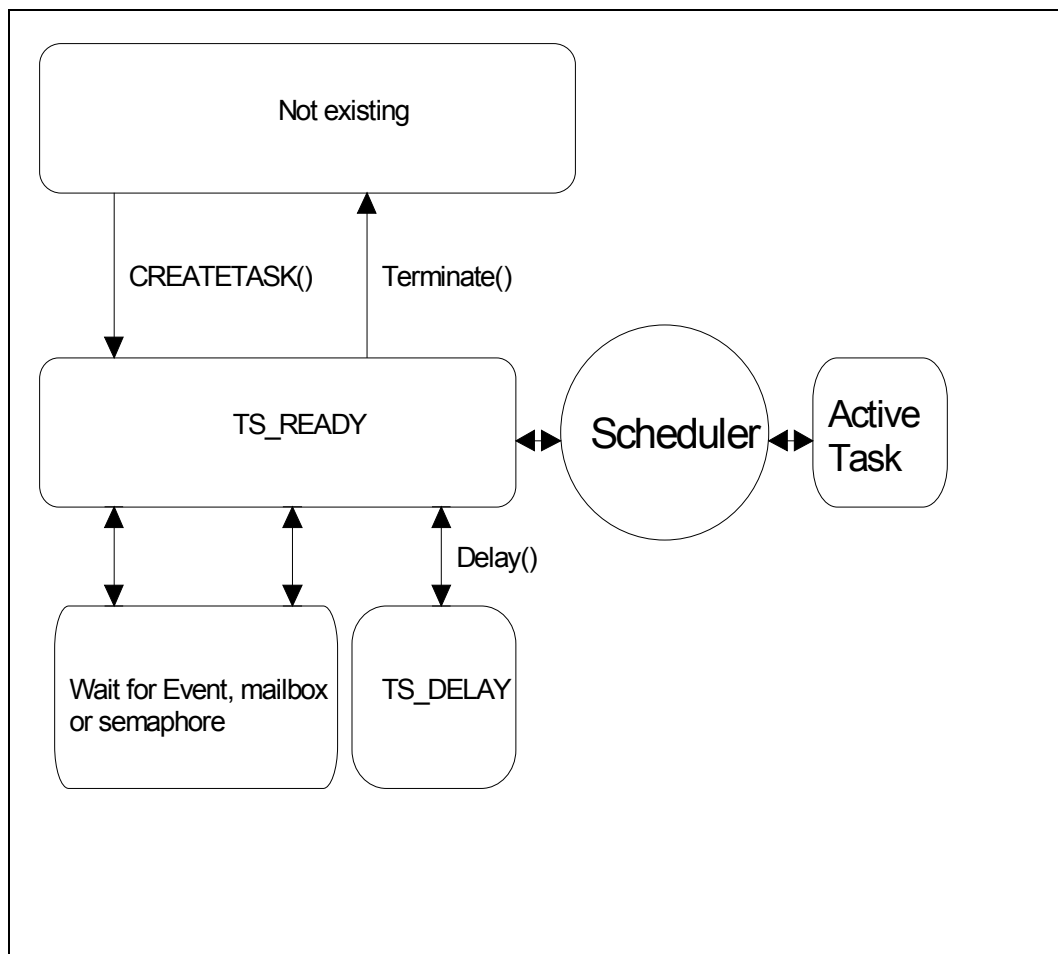
3.9. How the OS gains control

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

The startup code does the following:

- Loads the stack pointers(s), with the default values, which is for most CPUs the end of the defined stack segment(s)
- Initialize all data segments to their respective values
- Calls the `main()` routine

In a single-task-program, the `main()` routine is part of the user program which takes control immediately after the "C" startup. Normally, *embOS* works with the standard "C" startup module without any modification. If there are any changes required, they 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 then on, the scheduler controls which task is executed.

`main()` will not be interrupted by any of the created tasks, because those tasks are executed only after the call to `OS_Start()`. It is therefore usually recommended to create all or most of your tasks here, as well as your control structures such as mailboxes and semaphores. A good practice is to write software in the form of modules which are (up to a point) reusable. These modules usually have an initialization routine, which creates the required task(s) and/or control structures. 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 that has been created in `main()`.

Please note that `OS_Start()` is called only once during the startup process and does not return.

The flowchart below illustrates the starting procedure:

```
                    ┌──────────┐
                   (  Reset of  )
                   (    CPU     )
                    └────┬─────┘
                         ▼
                   ┌──────────┐
                   │ Load SP  │
                   └────┬─────┘
                        ▼
                   ┌──────────┐
                   │   Init   │
                   │  memory  │
                   └────┬─────┘
                        ▼
                   ┌──────────┐      Init
                   │  main()  │      Hardware
                   │          │      Create Tasks,
                   └────┬─────┘      Semaphor
                        ▼
                   ╭──────────╮
                   │  embOS   │
                   │ Scheduler│
                   ╰──────────╯
          ╭──────╮          ╭──────╮
          │ Task │          │ Task │
          ╰──────╯          ╰──────╯
               ╭──────╮
               │ Task │
               ╰──────╯
```
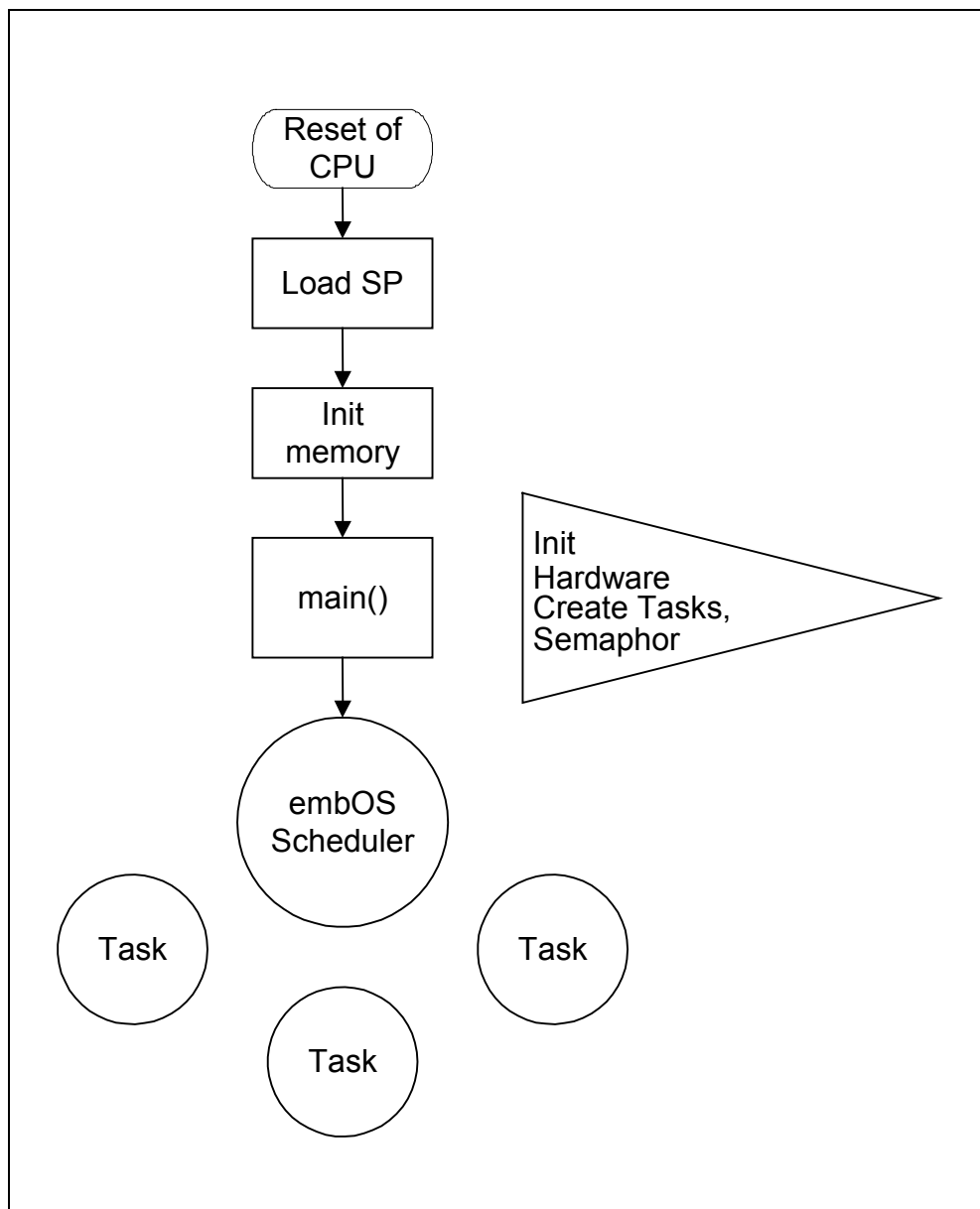
3.10. Different builds of *embOS*

embOS comes in different builds, or 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 (though bigger and slower) version for development which will catch most of the common application programming errors. Of course, you may also use the release version of *embOS* during development, but it will not catch these 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 such as bigger task control blocks, additional ROM (app. 200 bytes) and additional 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 check, 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: Debug including trace, profiling	OS_LIBMODE_DT	Maximum run time checking, plus tracing API calls and profiling

4. Task routines

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

- The task routine cannot take parameters.
- The task routine must never be called directly from your application.
- The task routine must not return.
- The task routine should be implemented as an endless loop, or it must terminate itself (see examples below).
- The task routine needs to be started from the scheduler, after the task is created and `OS_Start()` is called.

Example of task routine as an endless loop

```
/*  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 task routine that terminates itself

```
/*  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 do so 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.

4.1. OS_CREATETASK(): Create a task

Description

Creates a task.

Prototype

```
void   OS_CREATETASK(OS_TASK*       pTask,
                     char*          pName,
                     void*          pRoutine,
                     unsigned 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. Must be within the following 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.

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 the READY state. If there is another task with the same priority, the new task will be placed right before it.

This macro is normally used to create a task instead of the function call OS_CreateTask(), because it has fewer parameters and is therefore easier to use.

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.

The absolute value of Priority is of no importance, only the value in comparison to the priorities of other tasks.

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. Most CPUs cannot use the entire memory area as stack. Most CPUs require alignment of stack in multiples of bytes. This is automatically done, when task stack is defined as array of int.

Example

```
OS_STACKPTR int 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 */
}
```

4.2. OS_CreateTask(): Create a task

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 (or 0) if not used.
Priority	Priority of the task. Must be within the following range: 1 <= 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.
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 ticks that the task will run until it suspends; thus enabling another task with the same priority. This parameter has no effect on some ports of *embOS* for efficiency reasons.

Return value

Void.

Add. information

This function works the same way as OS_CREATETASK(), except that all parameters of the task can be specified.
The task can be dynamically created because the stack size is not calculated automatically as it is with the macro.

Important

The stack that you define has to reside in an area that the CPU can actually use as stack. Most CPUs cannot use the entire memory area as stack. Most CPUs require alignment of stack in multiples of bytes. This is automatically done, when task stack is defined as array of int.

Example

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

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);
}
```

4.3. OS_Delay(): Suspend for fixed time

Description

Suspends the calling task for a specified period of time.

Prototype

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

Parameter	Meaning
ms	Time interval to delay. Must be within 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

The calling task will be put into the TS_DELAY state for the period of time specified.

The task will stay in the delayed 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 another task or by 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");
}
```

4.4. OS_DelayUntil(): Suspend until

Description

Suspends the calling task until a specified time.

Prototype

```
void OS_DelayUntil(int t);
```

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

Return value

Void.

Add. information

The calling task will be put into the TS_DELAY state until the time specified. `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 = OS_GetTime();
  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.

4.5. OS_SetPriority(): Change priority of a task

Description

Assigns a specified priority to a specified task.

Prototype

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

Parameter	Meaning
pt	Pointer to a data structure of type OS_TASK.
Priority	Priority of the task. Must be within the following range: 1 <= Priority <= 255 Higher values indicate higher priorities.

Return value

Void.

Add. information

Can be called at any time 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.

4.6. 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.

Return value

Priority of the specified task as unsigned char (range 1 to 255).

Add. information

If pt is the NULL pointer, the function returns the priority of the currently running task.

If pt does not specify a valid task, the debug version of *embOS* calls OS_Error().

The release version of *embOS* cannot check the validity of pt and may therefore return invalid values if pt does not specify a valid task.

4.7. OS_SetTimeSlice(): Change timeslice of a task

Description

Assigns a specified timeslice value to a 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. Must be within the following range: 1 <= TimeSlice <= 255.

Return value

Previous timeslice value of the task as unsigned char.

Add. information

Can be called at any time from any task or software timer. Setting the timeslice value only affects the tasks running in round-robin mode. This means another 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.

4.8. OS_Suspend(): Suspend a task

Description

Suspends the specified task.

Prototype

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

Parameter	Meaning
pTask	Pointer to a data structure of type OS_TASK which is used as task control block (and reference) for the task that should be suspended. Since version 3.30b of embOS, passing a NULL pointer is allowed and will suspend the current task.

Return value

Void.

Add. information

If pTask is the NULL pointer, the current task suspends.

If the function succeeds, execution of the specified task is suspended and the task's suspend count is incremented.

The specified task will be suspended immediately. It can only be restarted by a call of OS_Resume().

Every task has a suspend count with a maximum value of OS_MAX_SUSPEND_CNT. If the suspend count is greater than zero, the task is suspended.

Calling OS_Suspend() more often than OS_MAX_SUSPEND_CNT times without calling OS_Resume(), the task's internal suspend count is not incremented and OS_Error() is called with error OS_ERR_SUSPEND_TOO_OFTEN in debug builds.

4.9. OS_Resume(): Restarts a suspended task

Description

Decrements the suspend count of specified task and resumes the task, if the suspend count reaches zero.

Prototype

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

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

Return value

Void.

Add. information

The specified task's suspend count is decremented. If the resulting value is 0, the execution of the specified task is resumed.

If the task is not blocked by other task blocking mechanisms, the task will be set back in ready state and continues operation according to the rules of the scheduler.

In debug versions of *embOS*, the OS_Resume() function checks the suspend count of the specified task. If the suspend count is 0 when OS_Resume() is called, the specified task is not currently suspended and OS_Error() is called with error OS_ERR_RESUME_BEFORE_SUSPEND.

4.10. OS_GetSuspendCnt(): Retrieve suspension count of a task

Description

This function may be used to examine whether a task is suspended by previous calls of `OS_Suspend()`.
The function returns the suspension count and thus suspension state of the specified task.

Prototype

`unsigned char OS_GetSuspendCnt(OS_TASK* pTask);`

Parameter	Meaning
PTask	Pointer to a data structure of type OS_TASK.

Return value

Suspension count of the specified task as unsigned character value.
0:          Task is not suspended.
>0:   Task is suspended by at least one call of `OS_Suspend()`.

Add. information

If `pTask` does not specify a valid task, the debug version of **embOS** calls `OS_Error()`.
The release version of **embOS** cannot check the validity of `pTask` and may therefore return invalid values if `pTask` does not specify a valid task.
When tasks are created and terminated dynamically, `OS_IsTask()` may be called prior calling `OS_GetSuspendCnt()` to examine whether the task is valid.
The returned value can be used to resume a suspended task by calling `OS_Resume()` as often as indicated by the returned value.

Example

```
/**  demo-function to illustrate the use of OS_GetSuspendCnt() */

void ResumeTask(OS_TASK* pTask) {
  unsigned char SuspendCnt;
  SuspendCnt = OS_GetSuspendCnt(pTask);
  while(SuspendCnt > 0) {
    OS_Resume(pTask);                   /* May cause a task switch */
    SuspendCnt--;
  }
}
```

4.11. OS_Terminate(): Terminate a task

Description

Ends (terminates) a task.

Prototype

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

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

Return value

Void.

Add. information

If pTask is the NULL pointer, the current task terminates.
The specified task will terminate immediately. The memory used for stack and task control block can be reassigned.

Since version 3.26 of *embOS*, all resources which are held by the terminated task are released. Any task may be terminated regardless of its state. This functionality is default for any 16bit or 32bit CPU and may be changed by re-compiling *embOS* sources.
On 8bit CPUs terminating tasks that hold any resources is prohibited. To enable safe termination, the *embOS* sources have to be recompiled with the compile time switch OS_SUPPORT_CLEANUP_ON_TERMINATE activated.

Important:

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

4.12. OS_WakeTask(): Resume a time suspended task

Description

Ends 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 is used as task control block (and reference) for this task.

Return value

Void.

Add. information

Puts the specified task (already suspended for a certain amount of time with `OS_Delay()` or `OS_DelayUntil()` 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), this command is ignored.

4.13. OS_IsTask(): Check whether a task is valid

Description

Determines 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 is used as task control block (and reference) for this task.

Return value

Character value:
0: TCB is not used by any task
1: TCB is used by a task

Add. information

This function checks to see if the specified 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 determine whether the task control block and stack for the task may be reused for another task in applications that create and terminate tasks dynamically.

4.14. OS_GetTaskID(): Retrieve ID of current task

Description

Returns the ID of the currently running task.

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 determine which task is executing. This may be helpful if the reaction of any function depends on the currently running task.

4.15. OS_GetpCurrentTask(): Retrieve TCB of current task

Description

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

Prototype

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

Return value

OS_TASK*: A pointer to the task control block structure.

Add. information

This function may be used to determine which task is executing. This may be helpful if the reaction of any function depends on the currently running task.

5. Software timers

A *software timer* is an object that calls a user-specified routine after a specified delay. A basically unlimited number of software timers can be defined with the macro `OS_CREATETIMER()`.

Timers can be stopped, started and retriggered much like hardware timers. When defining a timer, you specify any routine that is to be called after the expiration of the delay. 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()`.

Maximum timeout / period

The timeout value is stored as integer, thus a 16-bit value on 8/16-bit CPUs, a 32-bit value on 32 bit CPUs. The comparisons are done as signed comparisons, (since expired time-outs are permitted). This means that only 15-bits can be used on 8/16 bit CPUs, 31 bits on 32-bit CPUs. An other factor to take into account is the maximum time spent in critical regions. Since during critical regions timer may expire, but the timer routine can not be called (timers are "put on hold"), the maximum time that the system spends at once in a critical region needs to be deducted. In most systems, this is no more than a single tick. However, to be safe, we have assumed that your system spends no more than up to 255 ticks in a row in a critical region and defined a macro which defines the maximum timeout value. It is normally 0x7F00 for 8/16-bit systems or 7FFFFF00 for 32-bit Systems and defined in RTOS.h as OS_TIMER_MAX_TIME. If your system does spend more than 255 ticks without break in a critical section (effectively disabling the scheduler during this time ... not recommended), you have to make sure your application uses shorter timeouts.

5.1. OS_CREATETIMER(): Create a software timer

Description

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 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 as follows:

```
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);
}
```

5.2. OS_CreateTimer(): Create a software timer

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 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 as follows:

```
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);
}
```

5.3. OS_StartTimer(): Start a timer

Description

Starts a 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.

Important

This function has no effect on running timers.
It also has no effect on timers that are not running, but are expired. Use `OS_RetriggerTimer()` to restart those timers.

5.4. OS_StopTimer(): Stop a timer

Description

Stops a 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.

5.5. OS_RetriggerTimer(): Restart a timer

Description

Restarts a 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 or with the function OS_SetTimerPeriod().

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);
}
```

5.6. OS_SetTimerPeriod(): Set restart value

Description

Sets a new timer reload value for a 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): Minimum 1 Maximum 32767.

Return value

Void.

Add. information

OS_SetTimerPeriod() sets the initial time value of the specified timer. Period is the reload value of the timer to be used 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);
}
```

5.7. OS_DeleteTimer(): Delete a timer

Description

Stops and deletes a 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.

5.8. OS_GetTimerPeriod(): Retrieve restart value

Description

Returns the current reload value of a 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 permitted range of timer values.

Add. information

The period returned is the reload value of the timer set as initial value when the timer is retriggered by OS_RetriggerTimer().

5.9. OS_GetTimerValue(): Retrieve remaining time

Description

Returns the remaining timer value of a 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 0 and 32767, which is the permitted range of timer values.

Add. information

The timer value is the remaining time until the timer expires and calls its call-back function.

5.10. OS_GetTimerStatus(): Retrieve timer status

Description

Returns the current timer status of a 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.

5.11. OS_GetpCurrentTimer(): Retrieve current timer

Description

Returns a pointer to the data structure of the timer that just expired.

Prototype

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

Return value

OS_TIMER*: A pointer 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;
```

6. Resource semaphores

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 lot more.

The basic procedure is as follows:

Any task that uses a 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. If a second task now tries to use the same resource while it is in use by the first task, this second 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:

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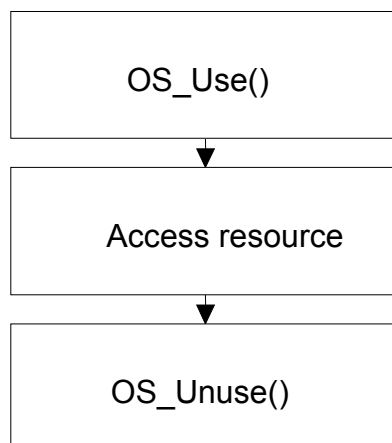
A resource semaphore contains a counter that keeps track of how many times the resource has been claimed by calling `OS_Request()` or `OS_Use()` by a particular task. It is released when that counter reaches 0, which means the `OS_Unuse()` routine has to be called exactly the same number of times as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

On the other hand, a task cannot release a resource that it does not own by calling `OS_Unuse()`. In the debug version of *embOS*, 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()`.

Example for use of resource semaphore

Here, two tasks access an LC display completely independently from each other. The LCD is a resource that needs to be protected with a resource semaphore. One task may not interrupt another task which is writing to the LCD, because otherwise the following might occur:

- Task A positions the cursor.
- Task B interrupts Task A and repositions the cursor.
- Task A writes to the wrong place in the LCD's memory.

To avoid this type of situation, every the LCD must be accessed by a task, it is first claimed by a call to `OS_Use()` (and is automatically waited for if the resource is blocked). After the LCD has been written to, it is released by a call to `OS_Unuse()`.

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


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

In most applications, the routines that access a resource should automatically call OS_Use() and OS_Unuse() so that when using the resource you do not have to worry about it and can use it just as you would in a single-task system. The following is an example of how to implement a resource 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);
}
```

6.1. OS_CREATERSEMA(): Create resource semaphore

Description

Macro that 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.

6.2. OS_Use(): Use a resource

Description

Claims a 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

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

Add. information

The following situations are possible:

- 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 being used by another task.
  The execution of this task is suspended 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.

Important:

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

The following diagram illustrates the function of the `OS_Use()` routine
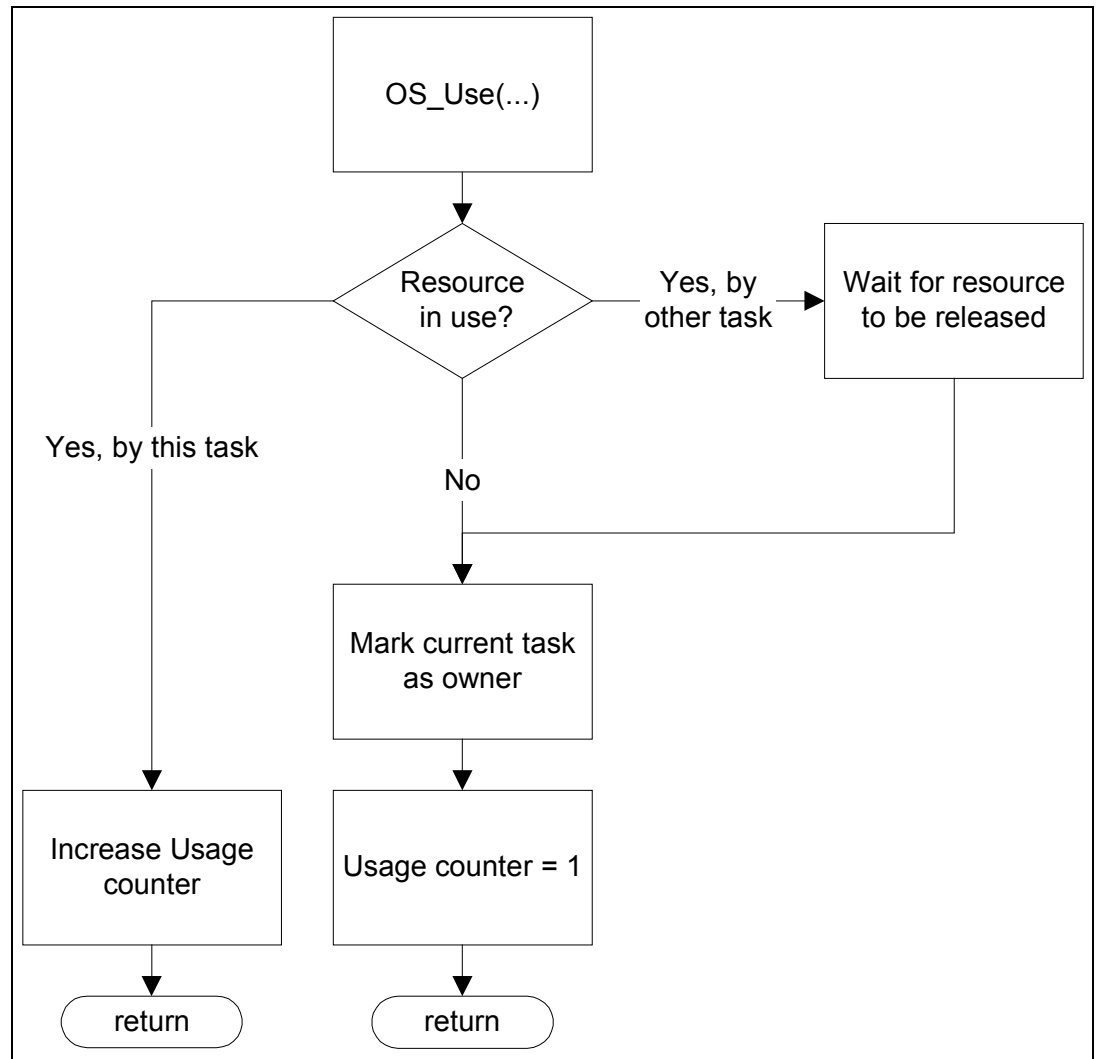
6.3. OS_Unuse(): Release a resource

Description

Releases a semaphore currently in use by a 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 sema-phore 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.

Important:

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

6.4. OS_Request(): Request a resource

Description

Requests a 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 was available, in use now by calling task
0: Resource was not available.

Add. Information

The following diagram illustrates how OS_Request() works:
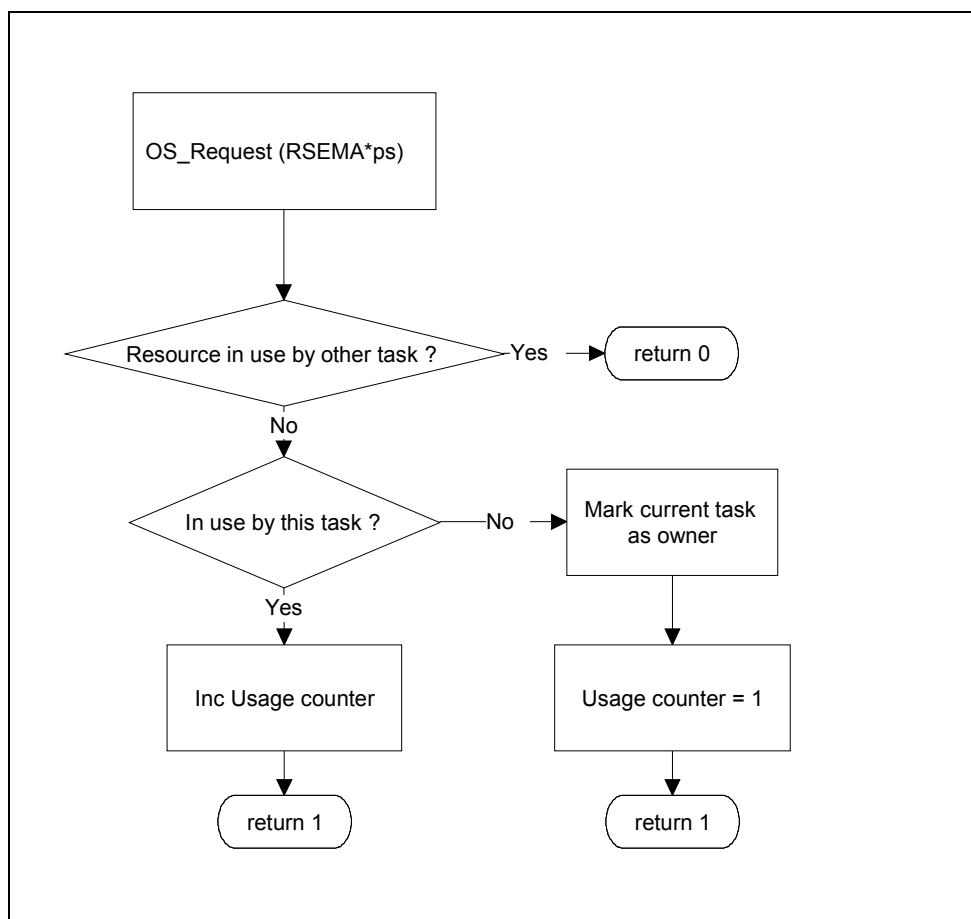
Example

```
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     */
```

6.5. OS_GetSemaValue(): Retrieve usage counter value

Description

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

Prototype

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

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

Return value

The counter of the semaphore.
A value of 0 means the resource is available.

Add. information

None.

6.6. OS_GetResourceOwner(): Retrieve blocking task

Description

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

Prototype

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

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

Return value

Pointer to the task that is blocking the resource.
A value of 0 means the resource is available.

Add. information

None.

6.7. OS_DeleteRSema(): Delete a resource semaphore

Description

Deletes a specified resource semaphore. The memory of that semaphore may be reused for other purposes or may be used to create an other resources semaphore using the same memory.

Prototype

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

Parameter	Meaning
pRSema	Pointer to a data structure of type OS_RSEMA.

Return value

Void.

Add. information

Before deleting a resource semaphore, make sure that no task is claiming the resources semaphore.

The debug version of *embOS* will call OS_Error(), if a resources semaphore is deleted when it is already used.

In systems with dynamic creation of resource semaphores, it is required to delete a resource semaphore, before re-creating it. Otherwise the semaphore handling will not work correctly.

7. 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 sometimes. 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, or any interrupt in any way.

Example for use of counting semaphore

```
OS_STACKPTR int 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_CREATECSEMA(&SEMALCD);                     /* Create Semaphore  */
  OS_CREATETASK(&TCB0, NullTask0, 100, Stack0);  /* Create Task0      */
  OS_CREATETASK(&TCB1, NullTask1,  50, Stack1);  /* Create Task1      */
}
```

7.1. OS_CREATECSEMA(): Create counting semaphore

Description

Macro that 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 OS_CSEMA needs 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, use the function `OS_CreateCSema()`.

7.2. OS_CreateCSema(): Create counting semaphore

Description

      Creates a counting semaphore with a specified initial count value.

Prototype

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

Parameter	Meaning
pCSema	Pointer to a data structure of type OS_CSEMA.
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 OS_CSEMA needs 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.

7.3. OS_SignalCSema(): Increment counter

Description

Increments the counter of a semaphore

Prototype

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

Parameter	Meaning
pCSema	Pointer to a data structure of type OS_CSEMA.

Return value

Void.

Add. information

`OS_SignalCSema()` signals an event to a semaphore by incrementing its counter. 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.

7.4. OS_SignalCSemaMax(): Increment counter upto a maximum value

Description

Increments the counter of a semaphore up to a specified maximum value.

Prototype

```
void OS_SignalCSemaMax(OS_CSEMA * pCSema,
                       unsigned char MaxValue );
```

Parameter	Meaning
pCSema	Pointer to a data structure of type OS_CSEMA.
MaxValue	1 <= MaxValue <= 255 Limit of semaphore count value.

Return value

Void.

Add. information

OS_SignalCSemaMax() signals an event to a semaphore by incrementing its counter, as long as current value of semaphore counter is below the specified maximum value. If one or more tasks are waiting for an event to be signaled to this semaphore, the tasks are put into ready state and the task that has the highest priority will become the active task.

Calling OS_SignalCSemaMax() with a MaxValue of 1 handles a counting semaphore as a binary semaphore.

7.5. OS_WaitCSema(): Decrement counter

Description

Decrements the counter of a semaphore.

Prototype

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

Parameter	Meaning
pCSema	Pointer to a data structure of type OS_CSEMA.

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 another 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.

Important:

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

7.6. OS_WaitCSemaTimed(): Decrement counter with timeout

Description

Decrements a semaphore counter if the semaphore is available within a speci-fied time.

Prototype

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

Parameter	Meaning
pCSema	Pointer to a data structure of type OS_CSEMA.
TimeOut	Maximum time until semaphore should be available

Return value

Integer value:
0: Failed, semaphore not available within timeout time
1: OK, semaphore was available and counter decremented.

Add. information

If the counter of the semaphore is not 0, the counter is decremented and pro-gram execution continues.
If the counter is 0, `WaitCSemaTimed()` waits until the semaphore is signaled by another task, a timer or an interrupt handler via a call to `OS_SignalCSema()`. The counter is then decremented and program execu-tion continues.
If the semaphore was not signaled within the specified time, the program exe-cution continues but returns a value of 0.
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.

Important:

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

7.7. OS_GetCSemaValue(): Retrieve counter value

Description

Returns the counter value of a specified semaphore.

Prototype

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

Parameter	Meaning
pCSema	Pointer to a data structure of type OS_CSEMA.

Return value

The counter value of the semaphore.

Add. information

None.

7.8. OS_DeleteCSema(): Delete a counting semaphore

Description

Deletes a specified semaphore. The memory of that semaphore may be reused for other purposes.

Prototype

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

Parameter	Meaning
pCSema	Pointer to a data structure of type OS_CSEMA.

Return value

Void.

Add. information

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

8. Mailboxes

8.1. Why mailboxes?

In the preceding chapters, task synchronization by the use of semaphores was described. Unfortunately, semaphores cannot transfer data from one task to another. If we needed to transfer data between tasks via a buffer for example, we could use a resource semaphore every time we accessed the buffer. But doing so would make the program less efficient. Another major disadvantage would be that we could not access the 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 would have to disable interrupts every time and in every place that we accessed these variables. This is possible, but it is a path full of pitfalls. It is also not easy for a task to wait for a character to be placed in a buffer without polling the global variable that contains the number of characters in the buffer. Again, there is a way out – 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.

8.2. 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 very simply just means "item of data". It will become clearer in the following typical applications explained in the following section.

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

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

These limitations have been placed on mailboxes in order to guarantee efficient coding and also to ensure efficient management. The limitations are normally not a problem.

For handling messages larger than 127 bytes, you may use queues. For more information, please refer to Chapter 9: "Queues".

8.3. 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 a key is detected as having been pressed, that key is put into a mailbox that is used as a keyboard buffer. The message is then retrieved by the task that handles keyboard input. The message in this case is typically a single byte that holds the key code; the message size is therefore 1 byte.

> The advantage of a keyboard buffer is that management 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 particular mailbox. The number of keys that can be stored in the type-ahead buffer depends only on the size of the mailbox buffer, 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 with mailboxes. Both your task programs and your interrupt handlers store or retrieve data to/from the same mailboxes. As with a keyboard buffer, the message size is 1 character.

> For interrupt-driven sending, the task places the character(s) in the mailbox using `OS_PutMail()` or `OS_PutMailCond()`; the interrupt handler that is activated when a new character can be sent retrieves this character with `OS_GetMailCond()`.

> For interrupt-driven receiving, the interrupt handler that is activated when a new character is received puts it in the mailbox using `OS_PutMailCond()`; the task receives it using `OS_GetMail()` or `OS_GetMailCond()`.

A buffer for commands sent to a task

> Assume you have one task controlling a motor as you might have in applications that control a machine. A simple way to give commands to this task for controlling the motor would be to define a structure for commands. The message size would then be the size of this structure.

8.4. OS_CREATEMB(): Create a mailbox

Description

Macro that creates a new mailbox.

Prototype

```
void OS_CREATEMB(OS_MAILBOX*   pMB,
                 unsigned char sizeofMsg,
                 unsigned int  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. (1 <= sizeofMsg <= 127)
MaxnofMsg	Maximum no. of messages. (1 <= MaxnofMsg <= 65535)
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 specified 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 another:

```
/*
* 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);
}
```

8.5. Single-byte mailbox functions

In many (if not the most) situations, mailboxes are used simply to hold and transfer single-byte messages. This is the case, for example, with a mailbox that takes the character received or sent via serial interface, or normally with a mailbox used as keyboard buffer. In some of these cases, 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*, variations on some mailbox functions are available for single-byte mailboxes. The general functions `OS_PutMail()`, `OS_PutMailCond()`, `OS_GetMail()`, and `OS_GetMailCond()` can transfer messages of sizes between 1 and 127 bytes each. Their single-byte equivalents `OS_PutMail1()`, `OS_PutMailCond1()`, `OS_GetMail1()`, and `OS_GetMailCond1()` function the same way with the exception that they execute much faster since management is simpler. It is recommended to use the single-byte versions if you transfer a lot of single byte-data via mailboxes.

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

8.6. OS_PutMail() / OS_PutMail1(): Store a message

Description

Stores a new message of a predefined size in a 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 calling 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 if you have to store data in a mailbox from within an ISR.

Important:

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

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);
}
```

8.7. OS_PutMailCond() / OS_PutMailCond1(): Store a message if possible

Description

Stores a new message of a predefined size in a mailbox, if the mailbox is able to accept one more message.

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

0:       Success; message stored.
1:       Message could not be stored (mailbox is full).

Add. information

If the mailbox is full, the message is not stored.
This function never suspends the calling task. It may therefore 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 handle a mailbox as keyboard buffer.

8.8. OS_PutMailFront() / OS_PutMailFront1(): Store a message in front into a mailbox

Description

Stores a new message of a predefined size into a mailbox in front of all other messages. This new message will be retrieved first.

Prototype

```
void OS_PutMailFront  (OS_MAILBOX * pMB, void* pMail);
void OS_PutMailFront1 (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 calling task is suspended. Since this routine might require a suspension, it must not be called from an interrupt routine. Use `OS_PutMailFrontCond()`/`OS_PutMailFrontCond1()` instead if you have to store data in a mailbox from within an ISR.

This function is useful to store "emergency" messages into a mailbox which have to be handled quick.

It may also be used in general instead of `OS_PutMail()` to change the FIFO structure of a mailbox into a LIFO structure.

Important:

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

Example

Single-byte mailbox as keyboard buffer:

```
OS_MAILBOX MBCmd;
char MBCmdBuffer[6];

void KEYMAN_StoreCommand(char k) {
  OS_PutMailFront1(&MBCmd, &k); /* store command, wait if no space in buffer*/
}

void KEYMAN_Init(void) {
  /* create mailbox functioning as command buffer */
  OS_CREATEMB(&MBCmd, 1, sizeof(MBCmdBuffer), &MBCmdBuffer);
}
```

8.9. OS_PutMailFrontCond() / OS_PutMailFrontCond1(): Store a message in front into a mailbox if possible

Description

Stores a new message of a predefined size into a mailbox in front of all other messages, if the mailbox is able to accept one more message. The new message will be retrieved first.

Prototype

```
char OS_PutMailFrontCond  (OS_MAILBOX * pMB, void* pMail);
char OS_PutMailFrontCond1 (OS_MAILBOX * pMB,
                           const char* pMail);
```

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

Return value

0:        Success; message stored.
1:        Message could not be stored (mailbox is full).

Add. information

If the mailbox is full, the message is not stored.
This function never suspends the calling task. It may therefore be called from an interrupt routine.
This function is useful to store "emergency" messages into a mailbox which have to be handled quick.
It may also be used in general instead of `OS_PutMail()` to change the FIFO structure of a mailbox into a LIFO structure.

8.10. OS_GetMail() / OS_GetMail1(): Retrieve a message

Description

Retrieves a new message of a predefined size from a mailbox.

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. Make sure that it 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.

Important:

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

Example

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

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

8.11. OS_GetMailCond() / OS_GetMailCond1(): Retrieve a message if possible

Description

Retrieves a new message of a predefined size from a mailbox, if a message is available.

Prototype

```
char OS_GetMailCond (OS_MAILBOX * pMB, void* pDest);
char OS_GetMailCond1(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. Make sure that it 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

Retrun value

0:      Success; message retrieved.
1:      Message could not be retrieved (mailbox is empty); destination remains unchanged.

Add. information

If the mailbox is empty, no message is retrieved, but the program execution continues.
This function never suspends the calling task. It may therefore also be called from an interrupt routine.

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;
}
```

8.12. OS_GetMailTimed(): Retrieve a message within a given time

Description

Retrieves a new message of a predefined size from a mailbox, if a message is available within a given time.

Prototype

```
char OS_GetMailTimed(OS_MAILBOX * pMB,
                     void* pDest,
                     int Timeout);
```

Parameter	Meaning
pMB	Pointer to the mailbox.
pDest	Pointer to the memory area that the message should be stored at. Make sure that it 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
Timeout	Maximum time in timer ticks until the requested mail has to be available.

Retrun value

0:       Success; message retrieved.
1:       Message could not be retrieved (mailbox is empty); destination remains unchanged.

Add. information

If the mailbox is empty, no message is retrieved, the task is suspended for the given timeout.
The task continues execution, according to the rules of the scheduler, as soon as a mail is available within the given timeout, or after the timeout value has expired.

Important:

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

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_GetMailTimed(&MBKey, &c, 10) /* Wait for 10 timer ticks */
  return c;
}
```

8.13. OS_WaitMail(): Wait until a mail is available

Description

Waits until a mail is available, but does not retrieve the message from the mail-box.

Prototype

```
void OS_WaitMail(OS_MAILBOX* pMB)
```

Parameter	Meaning
pMB	Pointer to the mailbox.

Return value

Void.

Add. information

If the mailbox is empty, the task is suspended until a mail is available, otherwise the task continues.
The task continues execution, according to the rules of the scheduler, as soon as a mail is available, but the mail is not retrieved from the mailbox.

Important:

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

8.14. OS_ClearMB(): Empty a mailbox

Description

Clears all messages in a 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);
}
```

8.15. OS_GetMessageCnt(): Get number of messages in mailbox

Description

Returns number of messages currently in a specified mailbox.

Prototype

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

Parameter	Meaning
pMB	Pointer to the mailbox.

Return value

The number of messages in the mailbox.

Add. information

None.

Example

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

8.16. OS_DeleteMB(): Delete a mailbox

Description

Deletes a 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 a 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 programmer's responsibility to:
- make sure that the program no longer uses the mailbox to be deleted
- make sure that the mailbox to be deleted actually exists (i.e. has been created first).

Example

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

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

9. Queues

9.1. Why queues?

In the preceding chapter, intertask communication using mailboxes was described. Mailboxes can handle small messages with fixed data size only.
Queues enable intertask communication with larger messages or with messages of various sizes.

9.2. 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 three major differences between queues and mailboxes:

1. Queues accept messages of various size. When putting a message into a queue, the message size is passed as a parameter.
2. Retrieving a message from the queue does not copy the message, but returns a pointer to the message and its size. 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.

Both the number and size of queues is limited only by the amount of available memory.

Any data structure can be written into a queue. The message size is not fixed.

9.3. OS_Q_Create(): Create 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));
}
```

9.4. 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 message to store

Return value

0:      Success; message stored.
1:      Message could not be stored (queue is full).

Add. information

If the queue is full, the function returns a value unequal to 0.
This routine never suspends the calling task. It may therefore also be called from an interrupt routine.

Example

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

9.5. OS_Q_GetPtr(): Retrieve message

Description

Retrieves a message from a 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

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

Add. information

If the queue is empty, the calling task is suspended until the queue receives a new message.
Since this routine might require a suspension, it must not be called from an interrupt routine. Use OS_GetPtrCond() instead.
The retrieved 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_GetPtr(&_MemoryQ, &pData);           /* Get message     */
    Memory_WritePacket(*(U32*)pData, pData+4, Len);  /* Process message */
    OS_Q_Purge(&_MemoryQ);                           /* Delete message  */
  }
}
```

9.6. OS_Q_GetPtrCond(): Retrieve message if possible

Description

Retrieves a message from a queue, if one message is available.

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 function returns 0. The value of ppData is undefined. This function never suspends the calling task. It may therefore also be called from an interrupt routine.
If a message could be retrieved, it 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();
    }
  }
}
```

9.7. OS_Q_Purge(): Delete one message in queue

Description

Deletes the last retrieved message in a 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 the 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  */
  }
}
```

9.8. OS_Q_Clear(): Delete all messages in queue

Description

Deletes all message in a queue.

Prototype

```
void OS_Q_Clear(OS_Q* pQ);
```

Parameter	Meaning
pQ	Pointer to the queue.

Return value

Void.

Add. information

None.

9.9. OS_Q_GetMessageCnt(): Get number of messages in queue

Description

Returns the number of messages currently in a queue.

Prototype

```
int OS_Q_GetMessageCnt(OS_Q* pQ);
```

Parameter	Meaning
pQ	Pointer to the queue.

Return value

The number of messages in the queue.

Add. information

None.

10. 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 sent 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 another 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 include 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 a 1-byte (8-bit) 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 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. It will be 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.

10.1. OS_WaitEvent(): Wait for event, then clear all events

Description

Waits for one of the events specified in the bitmask and clears the event memory after an event occurs.

Prototype

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

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

Return value

All events that have actually occurred.

Add. information

If none of the specified events are signaled, the task is suspended. The first of the specified events will wake the task. These events are signaled by another task, a S/W timer or an interrupt handler.

Any bit in the 8-bit event mask may enable the according event.

Example

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

For a further example, see OS_SignalEvent().

10.2. OS_WaitSingleEvent(): Wait for event, then clear masked events only

Description

Waits for one of the events specified as bitmask and clears only that event after it occurs.

Prototype

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

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

Return value

All masked events that have actually occurred.

Add. information

If none of the specified events are signaled, the task is suspended. The first of the specified events will wake the task. These events are signaled by another task, a S/W timer or an interrupt handler.
Any bit in the 8-bit event mask may enable the according event.
All unmasked events remain unchanged.

Example

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

10.3. OS_WaitEventTimed():Wait for event with timeout

Description

Waits for the specified events for a given time, and clears the event memory after an event occurs.

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. (1 <= TimeOut <= 32767)

Return value

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

Add. information

If none of the specified events are available, the task is suspended for the given time. The first of the specified events will wake the task if the event is signaled by another task, a S/W timer or an interrupt handler within the specified Time-Out time.

If no event is signaled, the task is activated after the specified timeout and all actual events are returned and then cleared.

Any bit in the 8-bit event mask may enable the according event.

Example

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

10.4. OS_WaitSingleEventTimed(): Wait for event, then clear masked events, with timeout

Description

Waits for the specified events for a given time; after an event occurs, only that event is cleared.

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. (1 <= TimeOut <= 32767)

Return value

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

Add. information

If none of the specified events are available, the task is suspended for the given time. The first of the specified events will wake the task if the event is signaled by another task, a S/W timer or an interrupt handler within the specified TimeOut time.
If no event is signaled, the task is activated after the specified timeout and the function returns zero.
Any bit in the 8-bit event mask may enable the according event.
All unmasked events remain unchanged.

Example

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

10.5. OS_SignalEvent(): Signal a task that an event has occured

Description

Signals event(s) to a specified task.

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 events 2 & 3).
pTask	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.

Example

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

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

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

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

void Task0(void) {
  OS_U8 MyEvent;
  while(1)
    MyEvent = OS_WaitEvent(EVENT_KEYPRESSED | EVENT_SERIN)
    if (MyEvent & EVENT_KEYPRESSED) {
      /* handle key press         */
    }
    if (MyEvent & EVENT_SERIN) {
      /* 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 were only waiting for a key to be pressed, 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.

10.6. OS_GetEventsOccured(): Get a list of events

Description

Returns a list of events that have occurred for a specified task.

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

The event mask of the events that have actually occurred.

Add. information

By calling this function, the actual events remain signaled. The event memory is not cleared.

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.

10.7. OS_ClearEvents(): Clear list of events

Description

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

Prototype

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

Parameter	Meaning
pTask	The task who's events are to be returned and cleared, NULL means current task.

Return value

The events that were actually signaled before clearing.

11. Heap type memory management

ANSI "C" offers some basic dynamic memory management functions. These are malloc, free, and realloc.

Unfortunately, these routines are not thread-safe; they can only be used from one task or by multiple tasks if they are called sequentially. Therefore, *embOS* offers task-safe variants of these routines. These variants have the same names as their ANSI counterparts, but are prefixed OS_; they are called OS_malloc(), OS_free(), OS_realloc(). The thread-safe variants that *embOS* offers use the standard ANSI routines, but make sure that the calls are serialized using a resource semaphore.

If heap memory management is not supported by the standard C-libraries for a specific CPU, *embOS* heap memory management is not implemented.

Heap type memory management is part of the *embOS* libraries. It does not use any resources if it is not referenced by the application (i.e. if the application does not use any memory management API function).

Note that another aspect of these routines may still be a problem: the memory used for the functions (known as heap) may fragment. This can lead to a situation where the total amount of memory is sufficient, but there is not enough memory available in a single block to satisfy an allocation request.

11.1. API reference

API routine	Short explanation
OS_malloc	Allocates a block of memory on the heap.
OS_free	Frees a block of memory previously allocated.
OS_realloc	Changes allocation size.

12. Fixed block size memory pools

Fixed block memory pools contain a specific number of fixed-size blocks of memory. The location in memory of the pool, the size of each block and the number of blocks are set at run time by the application via a call to the create function. The advantage of fixed memory pools is that a block of memory can be allocated from within any task in a very short, determined period of time.

12.1. API reference

All API functions for fixed block size memory pools are prefixed `OS_MEMF_`.

API routine	Short explanation
Create / Delete	
OS_MEMF_Create	Create fixed block memory pool.
OS_MEMF_Delete	Delete fixed block memory pool.
Allocation	
OS_MEMF_Alloc	Allocate memory block from a given memory pool. Wait indefinitely if no block is available.
OS_MEMF_AllocTimed	Allocate memory block from a given memory pool. Wait no longer than given timeout if no block is available.
OS_MEMF_Request	Allocate block from a given memory pool, if available. Non-blocking.
Release	
OS_MEMF_Release	Release memory block from a given memory pool.
OS_MEMF_FreeBlock	Release memory block from any pool.
Info	
OS_MEMF_GetNumFreeBlocks	Returns the number of available blocks in a pool.
OS_MEMF_IsInPool	Returns !=0 if block is in memory pool.
OS_MEMF_GetMaxUsed	Returns the maximum number of blocks in a pool which have been used at a time.
OS_MEMF_GetNumBlocks	Returns the number of blocks in a pool.
OS_MEMF_GetBlockSize	Returns the size of one block of given pool.

12.2. OS_MEMF_Create(): Create a fixed size memory pool

Description

Creates and initializes a fixed block size memory pool.

Prototype

```
void OS_MEMF_Create(OS_MEMF* pMEMF,
                     void* pPool,
                     OS_U16 NumBlocks,
                     OS_U16 BlockSize);
```

Parameter	Meaning
pMEMF	Pointer to the control data structure of memory pool.
pPool	Pointer to memory to be used for the memory pool. Required size is: NumBlocks * (BlockSize + OS_MEMF_SIZEOF_BLOCKCONTROL).
NumBlocks	Number of blocks in the memory pool.
BlockSize	Size of one block in bytes.

Return value

Void.

Add. information

OS_MEMF_SIZEOF_BLOCKCONTROL gives the number of bytes used for control and debug purposes. It is guaranteed to be 0 in release or stack check builds.
Before using any memory pool, it has to be created.
The debug version of libraries keeps track of created and deleted memory pools. The release and stack check versions do not.

12.3. OS_MEMF_Delete(): Delete a fixed size memory pool

Description

Deletes a fixed block size memory pool. After deletion, the memory pool and memory blocks inside this pool may no longer be used.

Prototype

```
void OS_MEMF_Delete(OS_MEMF* pMEMF);
```

Parameter	Meaning
pMEMF	Pointer to the control data structure of memory pool.

Return value

Void.

Add. information

This routine is provided for completeness. It is not used in the majority of applications because there is no need to dynamically create/delete memory pools.
Most applications prefer to have a static memory pool design; memory pools are created at startup (before calling OS_Start()) and will never be deleted.
The debug version of libraries mark the memory pool as deleted.

12.4. OS_MEMF_Alloc(): Retrieve one block from memory pool

Description

Requests allocation of a memory block.
Waits until a block of memory is available.

Prototype

```
void* OS_MEMF_Alloc(OS_MEMF* pMEMF, int Purpose);
```

Parameter	Meaning
pMEMF	Pointer to the control data structure of memory pool.
Purpose	This is a parameter which is used for debugging only. Its value has no effect on program execution, but may be remembered in debug builds to allow run time analysis of memory allocation problems.

Return value

Pointer to the allocated block.

Add. Information

If there is no free memory block in the pool, the calling task is suspended until a memory block becomes available.
The retrieved pointer must be delivered to `OS_MEMF_Release()` as parameter to free the memory block. The pointer must not be modified.

12.5. OS_MEMF_AllocTimed(): Retrieve block with timeout

Description

Requests allocation of a memory block.
Waits until a block of memory is available or the timeout has expired.

Prototype

```
void* OS_MEMF_AllocTimed(OS_MEMF* pMEMF,
                         int Timeout,
                         int Purpose);
```

Parameter	Meaning
pMEMF	Pointer to the control data structure of memory pool.
Timeout	Timeout, given in ticks. 0 or negative values are permitted.
Purpose	This is a parameter which is used for debugging only. Its value has no effect on program execution, but may be remembered in debug builds to allow run time analysis of memory allocation problems.

Return value

!=NULL   pointer to the allocated block
NULL   if no block has been allocated.

Add. Information

If there is no free memory block in the pool, the calling task is suspended until a memory block becomes available or the timeout has expired.
The retrieved pointer must be delivered to `OS_MEMF_Release()` as parameter to free the memory block. The pointer must not be modified.

12.6. OS_MEMF_Request(): Retrieve memory block if available

Description

Requests allocation of a memory block.
Continues execution in any case.

Prototype

```
void* OS_MEMF_Request(OS_MEMF* pMEMF, int Purpose);
```

Parameter	Meaning
pMEMF	Pointer to the control data structure of memory pool.
Purpose	This is a parameter which is used for debugging only. Its value has no effect on program execution, but may be remembered in debug builds to allow run time analysis of memory allocation problems.

Return value

!=NULL   pointer to the allocated block
NULL   if no block has been allocated.

Add. Information

The calling task is never suspended by calling `OS_MEMF_Request()`.
The retrieved pointer must be delivered to `OS_MEMF_Release()` as parameter to free the memory block. The pointer must not be modified.

12.7. OS_MEMF_Release(): Free a memory block in pool

Description

Releases a memory block that was previously allocated.

Prototype

```
void OS_MEMF_Release(OS_MEMF* pMEMF, void* pMemBlock);
```

Parameter	Meaning
pMEMF	Pointer to the control data structure of memory pool.
pMemBlock	Pointer to the memory block to free.

Return value

Void.

Add. Information

The pMemblock pointer has to be the one that was delivered form any retrival function described above. The pointer must not be modified between allocation and release.
The memory block becomes available for other tasks waiting for a memory block from the pool.

If any task is waiting for a fixed memory block, it is activated according to the rules of the scheduler.

12.8. OS_MEMF_FreeBlock(): Free a memory block

Description

Releases a memory block that was previously allocated. The memory pool does not need to be denoted.

Prototype

```
void OS_MEMF_FreeBlock(void* pMemBlock);
```

Parameter	Meaning
pMemBlock	Pointer to the memory block to free.

Return value

Void.

Add. Information

The pMemblock pointer has to be the one that was delivered form any retrieval function described above. The pointer must not be modified between allocation and release.

This function may be used instead of OS_MEMF_Release(). It has the advantage that only one parameter is needed. *embOS* itself will find the associated memory pool.

The memory block becomes available for other tasks waiting for a memory block from the pool.

If any task is waiting for a fixed memory block, it is activated according to the rules of the scheduler.

12.9. OS_MEMF_GetNumBlocks(): Returns number of blocks in pool

Description

Info routine to examine the total number of all memory blocks in the pool.

Prototype

```
int OS_MEMF_GetNumFreeBlocks(OS_MEMF* pMEMF);
```

Parameter	Meaning
pMEMF	Pointer to the control data structure of memory pool.

Return value

returns the number of blocks in the specified memory pool. This is the value that was given as parameter during creation of the memory pool.

12.10. OS_MEMF_GetBlockSize(): Returns size of one memory block

Description

Info routine to examine the size of one memory block in the pool.

Prototype

```
int OS_MEMF_GetBlockSize(OS_MEMF* pMEMF);
```

Parameter	Meaning
pMEMF	Pointer to the control data structure of memory pool.

Return value

returns the size in bytes of one memory block in the specified memory pool. This is the value that was given as parameter during creation of the memory pool.

12.11. OS_MEMF_GetNumFreeBlocks(): Returns number of free blocks in pool

Description

Info routine to examine the number of free memory blocks in the pool.

Prototype

```
int OS_MEMF_GetNumFreeBlocks(OS_MEMF* pMEMF);
```

Parameter	Meaning
pMEMF	Pointer to the control data structure of memory pool.

Return value

The number of free blocks actually available in the specified memory pool.

12.12. OS_MEMF_GetMaxUsed(): Returns max. number of used blocks in pool

Description

Info routine to examine the amount of memory blocks in the pool that were used concurrently since creation of the pool.

Prototype

```
int OS_MEMF_GetMaxUsed(OS_MEMF* pMEMF);
```

Parameter	Meaning
pMEMF	Pointer to the control data structure of memory pool.

Return value

returns the maximum number of blocks in the specified memory pool that were used concurrently since creation of the pool.

12.13. OS_MEMF_IsInPool(): Check if block belongs to pool

Description

Info routine to examine whether a memory block reference pointer belongs to the specified memory pool.

Prototype

```
char OS_MEMF_IsInPool(OS_MEMF* pMEMF, void* pMemBlock);
```

Parameter	Meaning
pMEMF	Pointer to the control data structure of memory pool.
pMemBlock	Pointer to a memory block which should be checked

Return value

0: Pointer does not belong to memory pool.
1: Pointer belongs to the pool.

13. Stacks

The stack is the memory area used to store the return address of function calls, parameters, and local variables, as well as for temporary storage. Interrupt routines also use the stack to save the return address and flag register, except in cases where the CPU has a separate stack for interrupt functions. Take a look at 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 needs 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 too small, a section of the memory that is not reserved for the stack will be overwritten, and 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), calling 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 larger 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 characters every time the task is deactivated in order to detect a stack overflow. If an overflow is detected, OS_Error() will be called.

13.1. 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 that a non- *embOS* program for this CPU would use. After transferring control to the *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 to the *CPU & Compiler Specifics* manual of *embOS* documentation.

13.2. Task stack

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

13.3. Interrupt stack

To reduce stack size in a multitasking environment, some processors use a specific stack area for interrupt service routines (called a 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 a hardware interrupt stack, *embOS* may support a separate stack for interrupts by calling the function `OS_EnterIntStack()` at beginning of an interrupt service routine and `OS_LeaveIntStack()` at its very end. In case the CPU already supports hardware interrupt stacks or if 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 that uses 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 stacks, please see the *CPU & Compiler Specifics* manual of *embOS* documentation.

13.4. OS_GetStackSpace()

Description

Returns the unused portion of a task 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

The unused portion of the task stack in bytes.

Add. information

In most cases, the stack size required by a task cannot be easily calculated, since it takes quite some time to calculate the worst-case nesting and the calculation itself is difficult.

However, the required stack size can be figured out using the function `OS_GetStackSpace()`, which 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 only available in the debug and stack check builds of *embOS* , since only these builds initialize the stack space used for the tasks.

Important

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

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);
}
```

14. Interrupts

In this chapter, you will find a very basic description of using interrupt service routines (ISRs) in cooperation with *embOS*. Specific details for your CPU and compiler may be found in the *CPU & Compiler Specifics* manual of *embOS* documentation.

Interrupts are interruptions of a program caused by hardware. When an interrupt occurs, the CPU normally saves some of its registers and executes a subroutine called an interrupt service routine, or ISR. After the ISR is completed, the program returns to the highest-priority task in the READY state. Normal interrupts are maskable; they can occur at any time unless they are disabled with the CPU's "disable interrupt" instruction. ISRs are also nestable – they can be recognized and executed within other ISRs.

There are several good reasons for using interrupt routines. They can respond very quickly to external events such as 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. Interrupts effectively allow events to be processed as they occur.

14.1. Interrupt latency

Interrupt latency is the time between interrupt request and execution of the first instruction of the interrupt service routine.
Every computer system has an interrupt latency. The latency depends on various factors and differs even on the same computer system. The value that one is typically interested in is the worst case interrupt latency.
The interrupt latency is a sum of a lot of different smaller delays explained below.

14.1.1. Causes of interrupt latencies

- The first delay is typically in the hardware: The interrupt request signal needs to be synchronized to the CPU clock. Depending on the synchronization logic, typically up to 3 CPU cycles can be lost before the interrupt request has reached the CPU core.
- The CPU will typically complete the current instruction. This instruction can take a lot of cycles; on most systems, divide, push-multiple or memory-copy instructions are the instructions which require most clock cycles. On top of the cycles required by the CPU, there are in most cases additional cycles required for memory access. In an ARM7 system, the instruction
  `STMDB  SP!,{R0-R11,LR};` Push parameters and perm. Registers
  is typically the worst case instruction.
  It stores 13 32-bit registers on the stack. The CPU requires 15 clock cycles.
- The memory system may require additional cycles for wait states.
- After completion of the current instruction, the CPU performs a mode switch or pushes registers (typically PC and flag registers) on the stack. In general, modern CPUs (such as ARM) perform a mode switch, which requires less CPU cycles than saving registers.
- Pipeline fill
  Most modern CPUs are pipelined. Execution of an instruction happens in various stages of the pipeline. An instruction is executed when it has reached its final stage of the pipeline. Since the mode switch has flushed the pipeline, a few extra cycles are required to refill the pipeline.

14.1.2. Additional causes for interrupt latencies

There can be additional causes for interrupt latencies.
These depend on the type of system used, but we list a few of them.

- Latencies caused by cache line fill
  If the memory system has one or multiple caches, these may not contain the required data. In this case, not only the required data is loaded from memory, but in a lot of cases a complete line fill needs to be performed, reading multiple words from memory
- Latencies caused by cache write back.
  A cache miss may cause a line to be replaced. If this line is marked as dirty, it needs to be written back to main memory, causing an additional delay.
- Latencies caused by MMU translation table walks.
  Translation table walks can take a considerable amount of time, especially as they involve potentially slow main memory accesses. In real-time interrupt handlers, translation table walks caused by the TLB not containing translations for the handler and/or the data it accesses can increase interrupt latency significantly.
- Application program.
  Of course the application program can cause add. latencies by disabling interrupts. This can make sense in some situations, but of course causes add. latencies.
- Interrupt routines
  On most systems, one interrupt disables further interrupts. Even if the interrupts are re-enabled in the ISR, this takes a few instructions, causing add. latency.
- RTOS (Real Time Operating system)
  An RTOS also needs to temporarily disable the interrupts which can call API-functions of the RTOS.
  Some RTOSes disable all interrupts, effectively worsening interrupt latencies for all interrupts,
  some (like *embOS*) disable only low-priority interrupts and do thereby not affect the latency of high priority interrupts.

14.2. Zero interrupt latency

Zero interrupt latency in the strict sense is not possible as explained above. What we mean when we say "Zero interrupt latency" is that the latency of high priority interrupts is not affected by the RTOS; a system using *embOS* will have the same worst-case interrupt latency for high priority interrupts as a system running without

14.3. High / low priority interrupts

Most CPUs support interrupts with different priorities. Different priorities have two effects:

- If different interrupts occur simultaneously, the interrupt with higher priority takes precedence and it's ISR is executed first.
- Interrupts can never be interrupted by other interrupts of the same or lower level of priority

How many different levels of interrupts there are depends on the CPU and the interrupt controller. Details are explained in the CPU/MCU/SOC manuals and the *embOS* port specific documentation.

embOS distinguishes two different levels of interrupts: High / Low priority interrupts.

The *embOS* port specific documentation explains where "the line is drawn", which interrupts are considered high and which interrupts are considered low priority.

In general, the differences are:
Low priority interrupts
- May call embOS API functions
- Latencies caused by *embOS*

High priority interrupts
- May not call embOS API functions
- No Latencies caused by *embOS* (Zero latency)

Example of different interrupt priority levels

M16C CPUs support 8 interrupt priority levels. With *embOS*, the 3 highest priority levels are treated as "High priority interrupts".
ARM CPUs support normal interrupts (IRQ) and fast interrupt (FIQ). Using *embOS*, the FIQ is treated as "High priority interrupt".

14.4. Rules for interrupt handlers

14.4.1. General rules

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

- Interrupt handlers preserve all registers.
  Interrupt handlers must restore the environment of a task completely. This environment normally consists of the registers only, so the ISR has to make sure that all registers modified during interrupt execution are saved at the beginning and restored at the end of the interrupt routine.

- Interrupt handlers have to be finished quickly.
  Calculations of intensive parts of the program should be kept out of interrupt handlers. An interrupt handler should only be used to store a received value or to trigger an operation in the regular program (task). It should not wait in any form or perform a polling operation.

14.4.2. Additional rules for preemptive multitasking

A preemptive multitasking system like *embOS* needs to know if the program that is executing is part of the current task or an interrupt handler. This is because *embOS* cannot perform a task switch during the execution of an interrupt handler; it can only do so at the end of an interrupt handler.

If a task switch were to occur during the execution of an ISR, the ISR would continue as soon as the interrupted task became 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* functions.

This leads us to the following rule:

- Interrupt functions that re-enable interrupts or use any *embOS* function need to call `OS_EnterInterrupt()` at the beginning, before executing any other command and, before they return, call either `OS_LeaveInterrupt()` or `OS_LeaveInterruptNoSwitch()` as last command.

If a higher priority task is made ready by the ISR, the task switch then occurs in the routine `OS_LeaveInterrupt()`. The end of the ISR 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 at the end of an interrupt service routine is not required, `OS_LeaveInterruptNoSwitch()` can be used instead.

14.5. Calling *embOS* routines from within an ISR

Before calling any *embOS* function from within an ISR, *embOS* has to be informed that an interrupt service routine is running.

14.5.1. OS_EnterInterrupt()

Description

Informs *embOS* that interrupt code is executing.

Prototype

```
void OS_EnterInterrupt(void);
```

Return value

Void.

Add. information

If `OS_EnterInterrupt()` is used, it should be the first function to be called in the interrupt handler. It must be used with either `OS_LeaveInterrupt()` or `OS_LeaveInterruptNoSwitch()` as the last function called.
The use of this function has the following effects:
disables task switches
keeps interrupts in internal routines disabled

14.5.2. OS_LeaveInterrupt()

Description

Informs *embOS* that the end of the interrupt routine has been reached; executes task switching within ISR.

Prototype

```
void OS_LeaveInterrupt(void);
```

Return value

Void.

Add. information

If `OS_LeaveInterrupt()` is used, it should be the last function to be called in the interrupt handler.
If the interrupt has caused a task switch, it is executed now (unless the program which was interrupted was in a critical region).

14.5.3. OS_LeaveInterruptNoSwitch()

Description

Informs *embOS* that the end of the interrupt routine has been reached but does not execute task switching within ISR.

Prototype

```
void OS_LeaveInterruptNoSwitch(void);
```

Return value

Void.

Add. information

If `OS_LeaveInterruptNoSwitch()` is used, it should be the last function to be called in the interrupt handler. If the interrupt has caused a task switch, it is not executed 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.

14.5.4. Example

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();
}
```

14.6. 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 cannot be interrupted. An example would be the access to a global volatile variable of type long on an 8/16-bit CPU. 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:

Bad example

```
volatile long lvar;

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

The problem with disabling and re-enabling interrupts is that functions that disable/enable the interrupt cannot be nested.

Your "C" compiler offers two intrinsic functions for enabling and disabling interrupts. These functions can still be used, but it is recommended to use the functions that *embOS* offers (to be precise, they only look like functions, but are macros in reality). If you do not use these 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 disabling interrupts 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 (the longer interrupts are disabled, the higher the interrupt latency). As long as you only call *embOS* functions with interrupts enabled, you may also safely use the compiler-provided intrinsics to disable interrupts.

14.6.1. OS_IncDI() / OS_DecRI()

The following functions are actually macros defined in RTOS.h, so they execute very quickly and are very efficient. It is important that they are used as a pair: OS_IncDI() first, then OS_DecRI().

OS_IncDI()

Short for **Increment and Disable Interrupts**
Increments the interrupt disable counter (OS_DICnt) and disables interrupts.

OS_DecRI()

Short for **Decrement and Restore Interrupts**
Decrements the counter and enables interrupts if the counter reaches 0.

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 in that any routine can be called and the interrupts will not be switched on before the matching OS_DecRI() has been executed.

If you need to disable 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 bit more efficient because the interrupt disable counter OS_DICnt is not modified twice, but only checked once. They have the disadvantage that they do not work with routines because the status of OS_DICnt is not actually changed, and they should therefore be used with great care. In case of doubt, use OS_IncDI() and OS_DecRI().

14.6.2. OS_DI() / OS_EI() / OS_RestoreI()

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 this function 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();
}
```

14.7. 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(); }
```

14.8. Nesting interrupt routines

Per default, interrupts are disabled in an 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. These are known as nested interrupts, illustrated in the diagram below:
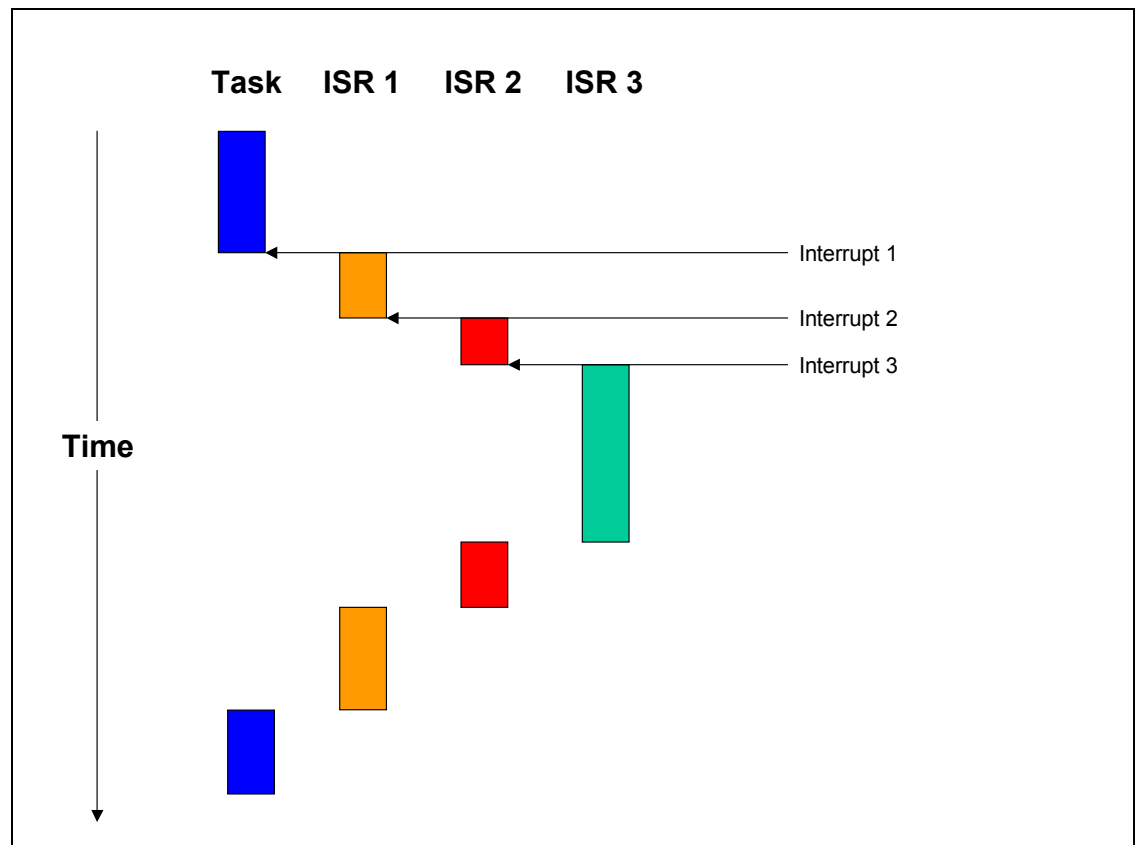
For applications requiring short interrupt latency, you may re-enable interrupts inside an ISR by using OS_EnterNestableInterrupt() and OS_LeaveNestableInterrupt() within the interrupt handler.

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, the enabling and disabling of interrupts from within an ISR has to be done using the functions that *embOS* offers for this purpose.

The routine OS_EnterNestableInterrupt() enables interrupts within an ISR and prevents further task switches; OS_LeaveNestableInterrupt() disables interrupts right before ending the interrupt routine again in order to restore the default condition. Re-enabling interrupts will make it possible for an *embOS* scheduler interrupt to shortly interrupt this ISR. In this case, *embOS* needs to know that an other ISR is still active and that it may not perform a task switch.

14.8.1. OS_EnterNestableInterrupt()

Description

Re-enables interrupts and increments the *embOS* internal critical region counter, thus disabling further task switches.

Prototype

```
void OS_EnterNestableInterrupt(void);
```

Return value

Void.

Add. information

This function should be the first call inside an interrupt handler when nested interrupts are required.
The function `OS_EnterNestableInterrupt()` is implemented as a macro and offers the same functionality as `OS_EnterInterrupt()` in combination with `OS_DecRI()`, but is more efficient, resulting in smaller and faster code.

Example

Refer to the example for `OS_LeaveNestableInterrupt()`.

14.8.2. OS_LeaveNestableInterrupt()

Description

Disables further interrupts, then decrements the *embOS* internal critical region count, thus re-enabling task switches if the counter has reached zero again.

Prototype

```
void OS_LeaveNestableInterrupt(void);
```

Return value

Void.

Add. information

This function is the counterpart of `OS_EnterNestableInterrupt()`, and has to be the last function call inside an interrupt handler when nested interrupts have been enabled before with `OS_EnterNestableInterrupt()`.
The function `OS_LeaveNestableInterrupt()` is implemented as a macro and offers the same functionality as `OS_LeaveInterrupt()` in combination with `OS_IncDI()`, but is more efficient, resulting in smaller and faster code.

14.8.3. OS_LeaveNestableInterruptNoSwitch()

Description

Disables further interrupts, informs *embOS* that the end of ISR is reached, but does not perform a task switch.

Prototype

```
void OS_LeaveNestableInterruptNoSwitch(void);
```

Return value

Void.

Add. information

If `OS_LeaveNestableInterruptNoSwitch()` is used, it should be the last
function to be called in the interrupt handler. If the interrupt has caused a task
switch, it is not executed 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.

Example of nestable interrupt handler

```
__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    */
}
```

14.9. Non-maskable interrupts (NMIs)

embOS performs atomic operations by disabling interrupts. However, a *non-maskable interrupt* (NIM) cannot be disabled, meaning it can interrupt these atomic operations. Therefore, NMIs should be used with great care and may under no circumstances call any *embOS* routines.

15. Critical regions

Critical regions are program sections during which the scheduler is switched off, meaning that no task switch and no execution of software timers are allowed except in situations where 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. 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. Software timers and interrupts are executed as critical regions anyhow, so it does not hurt but does not do any good either to declare them as such. If a task switch becomes due during the execution of a critical region, it will be performed right after the region is left.

15.1. OS_EnterRegion(): Enter critical region

Description

Indicates to the OS the beginning of a critical region.

Prototype

```
void OS_EnterRegion(void);
```

Return value

Void.

Add. information

`OS_EnterRegion()` is not actually a function but a macro. However, it behaves very much like a function with the difference that it is much 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()`; however, disabling interrupts will disable preemptive task switches.

Example

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

15.2. OS_LeaveRegion(): Leave critical region

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 it is much 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 the example for `OS_EnterRegion()`.

16. 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 are 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.

16.1. Time Variables

16.1.1. OS_Time

Description

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

Prototype

```
extern volatile OS_I32 OS_Time;
```

Add. information

The time variable has a resolution of one time unit, which is normally 1/1000 sec (1 ms) and is 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()` as explained in Chapter 18: "Time-related routines".

16.1.2. 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, OS_TimeDex will be assigned the time stamp of the next task or timer to be activated.

16.2. OS internal variables and data-structures

embOS internal variables are not explained here as they are 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.

17. 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 the optional embOSView.

The file `RTOSInit.c` is provided in source code form and can be modified in order to match your target hardware needs. It is compiled and linked with your application program.

17.1. Hardware-specific routines

Routine	Explanation
OS_InitHW()	Initializes the hardware timer used for generating interrupts. *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.
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. Cycle length 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_tx()	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.

17.2. Configuration defines

For most embedded systems, configuration is done by simply modifying the following defines, located at the top of the `RTOSInit.c` file:

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.

17.3. How to change settings

The only file which you may need to change is `RTOSInit.c`. This file contains all hardware-specific routines. The one exception is that some ports of *embOS* require an additional interrupt vector table file (details can be found in the *CPU & Compiler Specifics* manual of *embOS* documentation).

17.3.1. Setting the system frequency OS_FSYS

Relevant defines

OS_FSYS

Relevant routines

`OS_ConvertCycles2us()` (used with profiling only)

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()`. The `RTOSInit.c` file contains more information about in which cases this is necessary and what needs to be done.

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

Relevant routines

`OS_ InitHW()`

embOS usually generates 1 interrupt per ms, making the timer-interrupt, or *tick*, normally equal to 1 ms. This is done by a timer initialized in the routine `OS_InitHW()`. 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`.

17.3.3. 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. Please refer to the contents of the `RTOSInit.c` file for more information on which UARTS are supported for your CPU.

17.3.4. Changing the tick frequency

Relevant defines

OS_FSYS

As noted above, *embOS* usually generates 1 interrupt per ms. OS_FSYS defines the clock frequency of your system in Hz (times per second). The value of OS_FSYS is taken to calculate the desired reload counter value for the system timer for 1000 interrupts/sec. The interrupt frequency is therefore normally 1 kHz.

Different (lower or higher) interrupt rates are possible. If you choose an interrupt frequency different from 1 kHz, 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() (see below). The basic time unit does not have to be 1 ms; it might just as well be 100 us or 10 ms or any other value. For most applications, 1 ms is a convenient value.

17.4. OS_CONFIG()

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

Examples

1) The following will increment the time variable OS_Time by 1 per RTOS timer-interrupt. This is the default for *embOS*, so usage of OS_CONFIG() is not required.

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

2) The following will 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 500 Hz, which would result in an *embOS* timer-interrupt every 2 ms, a call of OS_Delay(10) would result in a delay of 20 ms, because all timing values are interpreted as ticks. A call of OS_CONFIG() with the parameter shown in example 2 would compensate for the difference, resulting in a delay of 10 ms when calling OS_Delay(10).

18. Time-related routines

embOS supports two basic types of run-time measurement which may be used to calculate the execution time of any section of user code. Low-resolution measurements use a time base of ticks, while high-resolution measurements are based on a time unit called a *cycle*. The length of a cycle depends on the timer clock frequency.

18.1. Low-resolution measurement

The system time variable OS_Time is measured in ticks, or ms. The low-resolution functions `OS_GetTime()` and `OS_GetTime32()` are used to return the current contents of this variable. The basic idea behind low-resolution measurement is quite simple: the system time is returned once before the section of code to be timed and once after, and the first value is subtracted from the second to obtain the time it took for the code to execute.

The term low-resolution is used because the time values returned are measured in completed ticks. Consider the following: With a normal tick of 1 ms, the variable OS_Time is incremented with every tick-interrupt, or once every ms. This means that the actual system time can potentially be more than what a low-resolution function will return (i.e. if an interrupt actually occurs at 1.4 ticks, the system will still have measured only 1 tick as having elapsed). The problem becomes even greater with run-time measurement since the system time must be measured twice. Each measurement can potentially be up to 1 tick less than the actual time, so the difference between two measurements could theoretically be inaccurate by up to two ticks.

The following diagram illustrates how low-resolution measurement works. We can see that the section of code actually begins at 0.5 ms and ends at 5.2 ms, which means that its actual execution time is (5.2 − 0.5) = 4.7 ms. However (with a tick of 1 ms), the first call to `OS_GetTime()` returns 0, and the second call returns 5. The measured execution time of the code would therefore result in (5 − 0) = 5 ms.
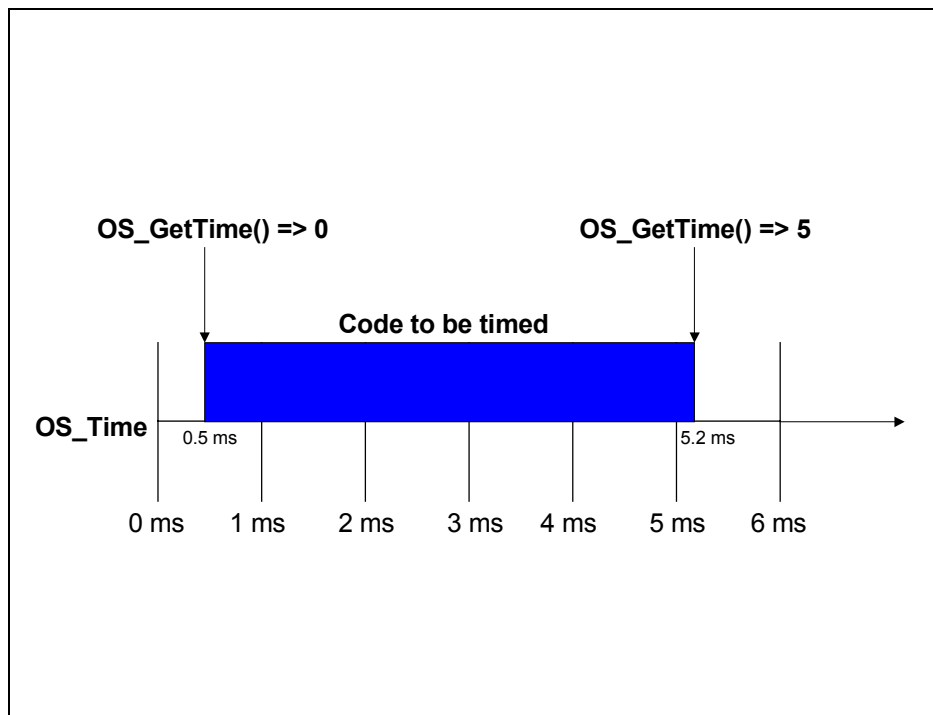
For many applications, low-resolution measurement may be fully sufficient for your needs. In some cases, it may be more desirable due to its ease of use and faster computation time than high-resolution measurement.

18.1.1. OS_GetTime()

Description

Returns the current system time in ticks.

Prototype

```
int OS_GetTime(void);
```

Return value

The system variable OS_Time as a 16- or 32-bit integer value.

Add. Information

This function returns the system time as a 16-bit value on 8/16-bit CPUs, and as a 32-bit value on 32-bit CPUs.
The OS_Time variable is a 32-bit value. Therefore, if the return value is 32-bit, it is simply the entire contents of the OS_Time variable. If the return value is 16-bit, it is the lower 16 bits of the OS_Time variable.

18.1.2. OS_GetTime32()

Description

Returns the current system time in ticks as a 32-bit value.

Prototype

```
long OS_GetTime32(void);
```

Return value

The system variable OS_Time as a 32-bit integer value.

Add. Information

This function always returns the system time as a 32-bit value. Since the OS_Time variable is also a 32-bit value, the return value is simply the entire contents of the OS_Time variable.

18.1.3. Example of typical use of low-resolution measurement

```
/*
* Measure the execution time with low resolution and return it in ms (ticks)
*/
int BenchmarkLoRes(void) {
   int t;
   t = OS_GetTime();
   UserCode();             /* Execute the user code to be benchmarked  */
   t = OS_GetTime() – t;
   return t;
}
```

18.2. High-resolution measurement

High-resolution measurement uses the same routines as those used in profiling builds of *embOS*, allowing for fine-tuning of time measurement. While system resolution depends on the CPU used, it is typically about 1 us, making high-resolution measurement about 1000 times more accurate than low-resolution calculations.

Instead of measuring the number of completed ticks at a given time, an internal count is kept of the number of cycles that have been completed. Look at the illustration below, which measures the execution time of the same code used in the low-resolution calculation. Fot this example, we assume that the CPU has a timer running at 10 MHz and is couting up. The number of cycles per tick is therefore (10 MHz / 1 kHz) = 10,000. This means that with each tick-interrupt, the timer restarts at 0 and counts up to 10,000.
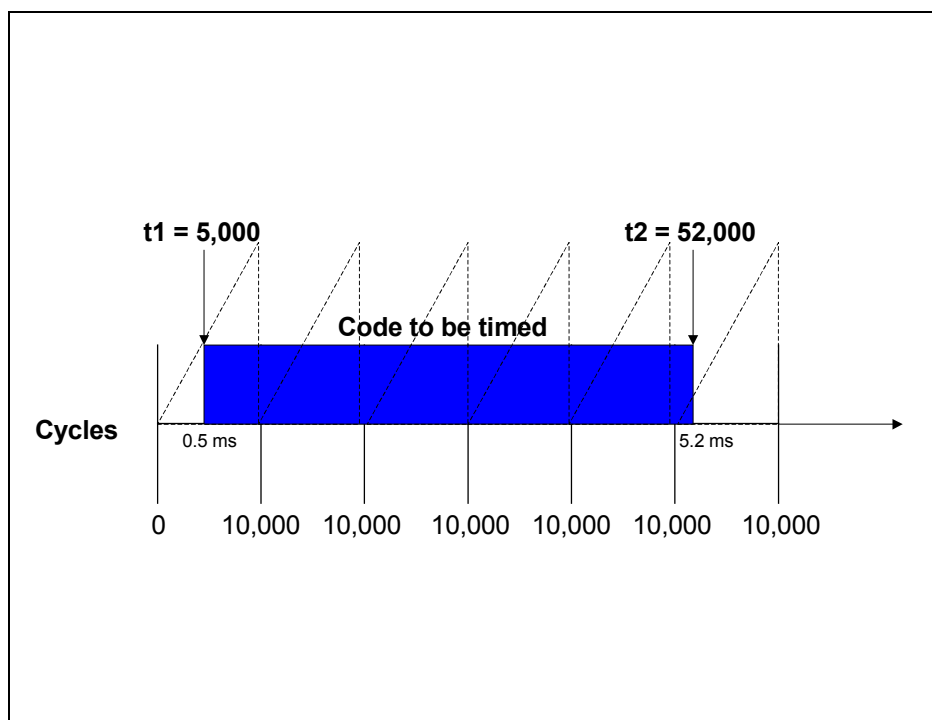
The call to `OS_Timing_Start()` calculates the starting value at 5,000 cycles, while the call to `OS_Timing_End()` calculates the ending value at 52,000 cycles (both values are kept track of internally). The measured execution time of the code in this example would therefore be (52,000 – 5,000) = 47,000 cycles, which corresponds to 4.7 ms.

Although the function `OS_Timing_GetCycles()` may be used to return the execution time in cycles as above, it is typically more common to use the function `OS_Timing_Getus()`, which returns the value in microseconds (us). In the above example, the return value would be 4,700 us.

<u>Data structure</u>

All high-resolution routines take as parameter a pointer to a data structure of type OS_TIMING, defined as follows:

```
#define OS_TIMING OS_U32
```

18.2.1. OS_Timing_Start()

Description

Marks the beginning of a section of code to be timed.

Prototype

```
void OS_Timing_Start(OS_TIMING* pCycle);
```

Parameter	Meaning
pCycle	Pointer to a data structure of type OS_TIMING.

Return value

Void.

Add. Information

This function must be used with `OS_Timing_End()`.

18.2.2. OS_Timing_End()

Description

Marks the end of a section of code to be timed.

Prototype

```
void OS_Timing_End(OS_TIMING* pCycle);
```

Parameter	Meaning
pCycle	Pointer to a data structure of type OS_TIMING.

Return value

Void.

Add. Information

This function must be used with `OS_Timing_Start()`.

18.2.3. OS_Timing_Getus()

Description

Returns the execution time of the code between `OS_Timing_Start()` and `OS_Timing_End()` in microseconds.

Prototype

`OS_U32 OS_Timing_Getus(OS_TIMING* pCycle);`

Parameter	Meaning
pCycle	Pointer to a data structure of type OS_TIMIING.

Return value

The execution time in microseconds (us) as a 32-bit integer value.

18.2.4. OS_Timing_GetCycles()

Description

Returns the execution time of the code between `OS_Timing_Start()` and `OS_Timing_End()` in cycles.

Prototype

`OS_U32 OS_Timing_GetCycles(OS_TIMING* pCycle);`

Parameter	Meaning
pCycle	Pointer to a data structure of type OS_TIMING.

Return value

The execution time in cycles as a 32-bit integer.

Add. Information

Cycle length depends on the timer clock frequency.

18.2.5. Example of typical use of high-resolution management

```
/*
*  Measure the execution time with hi resolution and return it in us
*/
OS_U32 BenchmarkHiRes(void) {
   OS_U32 t;
   OS_Timing_Start(&t);
   UserCode();                /* Execute the user code to be benchmarked */
   OS_Timing_End(&t);
   Return OS_Timing_Getus(&t);
}
```

18.3. Example

The following sample demonstrates the use of low-resolution and high-resolution measurement to return the execution time of a section of code:

```c
/*********************************************************
*        SEGGER MICROCONTROLLER SYSTEME GmbH
*  Solutions for real time microcontroller applications
*********************************************************
File       : SampleHiRes.c
Purpose    : Demonstration of embOS Hires Timer
--------------END-OF-HEADER-----------------------------*/

#include "RTOS.H"
#include <stdio.h>

OS_STACKPTR int Stack[1000]; /* Task stacks */
OS_TASK TCB;                 /* Task-control-blocks */

volatile int Dummy;
void UserCode(void) {
  for (Dummy=0; Dummy < 11000; Dummy++);  /* Burn some time  */
}

/*
* Measure the execution time with low resolution and return it in ms (ticks)
*/
int BenchmarkLoRes(void) {
  int t;
  t = OS_GetTime();
  UserCode();              /* Execute the user code to be benchmarked */
  t = OS_GetTime() - t;
  return t;
}

/*
* Measure the execution time with hi resolution and return it in us
*/
OS_U32 BenchmarkHiRes(void) {
  OS_U32 t;
  OS_Timing_Start(&t);
  UserCode();              /* Execute the user code to be benchmarked */
  OS_Timing_End(&t);
  return OS_Timing_Getus(&t);
}


void Task(void) {
  int tLo;
  OS_U32 tHi;
  char ac[80];
  while (1) {
    tLo = BenchmarkLoRes();
    tHi = BenchmarkHiRes();
    sprintf(ac, "LoRes: %d ms\n", tLo);
    OS_SendString(ac);
    sprintf(ac, "HiRes: %d us\n", tHi);
    OS_SendString(ac);
  }
}


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

void main(void) {
  OS_InitKern();        /* initialize OS                 */
  OS_InitHW();          /* initialize Hardware for OS    */
  /* You need to create at least one task here !         */
  OS_CREATETASK(&TCB, "HP Task", Task, 100, Stack);
  OS_Start();           /* Start multitasking            */
}
```

The output of the above sample is as follows:

```
LoRes: 7 ms
HiRes: 6641 us
LoRes: 7 ms
HiRes: 6641 us
LoRes: 6 ms
```

19. STOP / HALT / IDLE modes

Most CPUs support power-saving STOP, HALT or IDLE modes. Using these types of modes is one possible way to save power consumption during idle times. As long as the timer-interrupt will wake up the system with every *embOS* tick, or as long as 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.

20. embOSView: profiling and analyzing

20.1. 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 at www.segger.com.
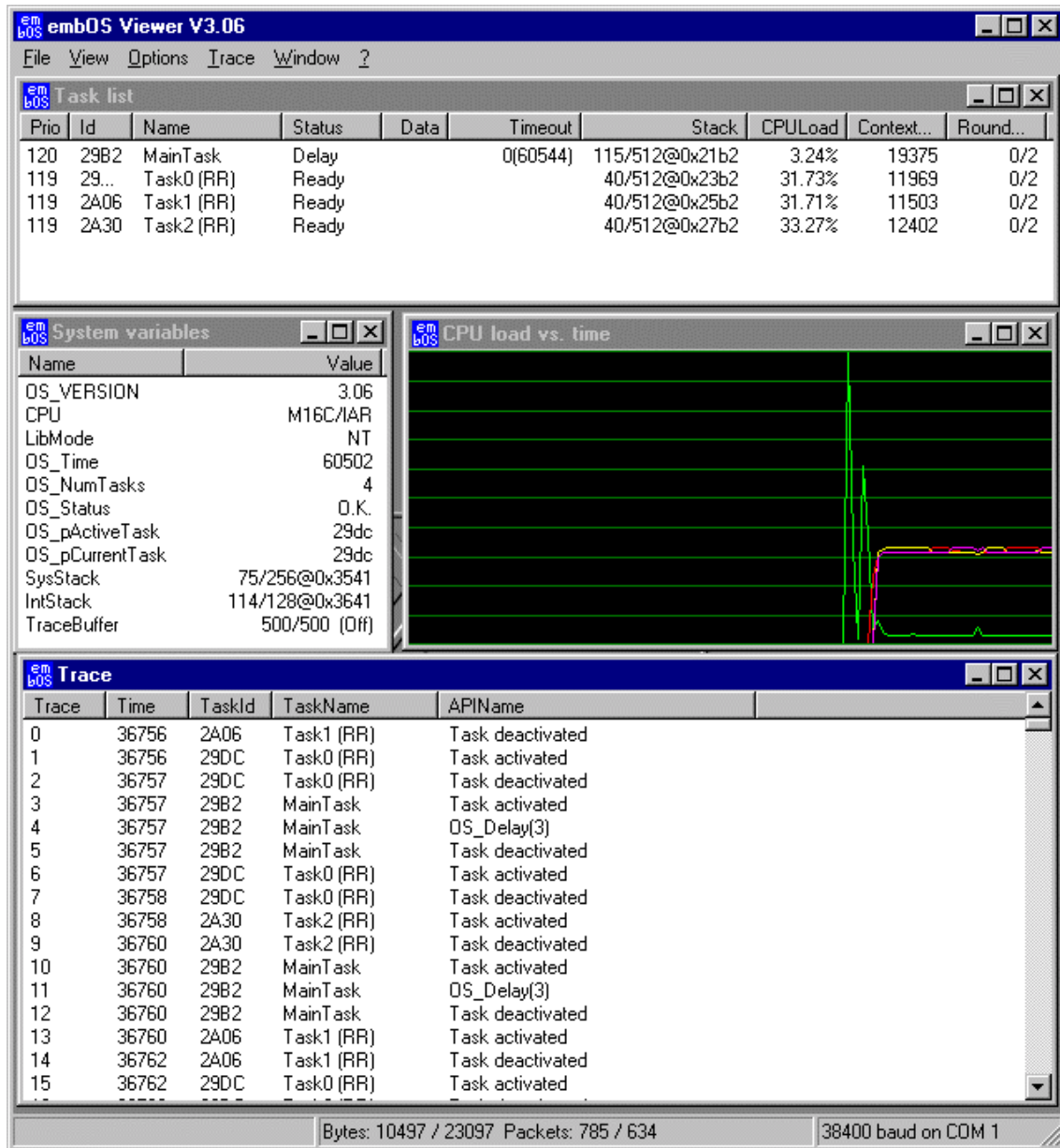
embOSView is a very helpful tool for analysis of the running target application.

20.2. 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	Current priority of task.	All
Id	Task ID, which is the address of the task control block.	All
Name	Name assigned during creation.	All
Status	Current state of task (ready, executing, delay, etc.) .	All
Data	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 helpful in analysis of stack usage and CPU load for every running task.

20.3. System variables window

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

20.4. Sharing the SIO for terminal I/O

The serial input/output (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 selecting `View/Terminal` from the menu.

To ensure communication via the terminal window in parallel with the viewer functions, the application uses the function `OS_SendString()` for sending a string to the terminal window and the function `OS_SetRxCallback()` to hook a reception routine that receives one byte.

20.4.1. OS_SendString()

Description

Sends a string over SIO to the terminal window.

Prototype

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

Parameter	Meaning
s	Pointer 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`.

20.4.2. 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 the 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);
}
```
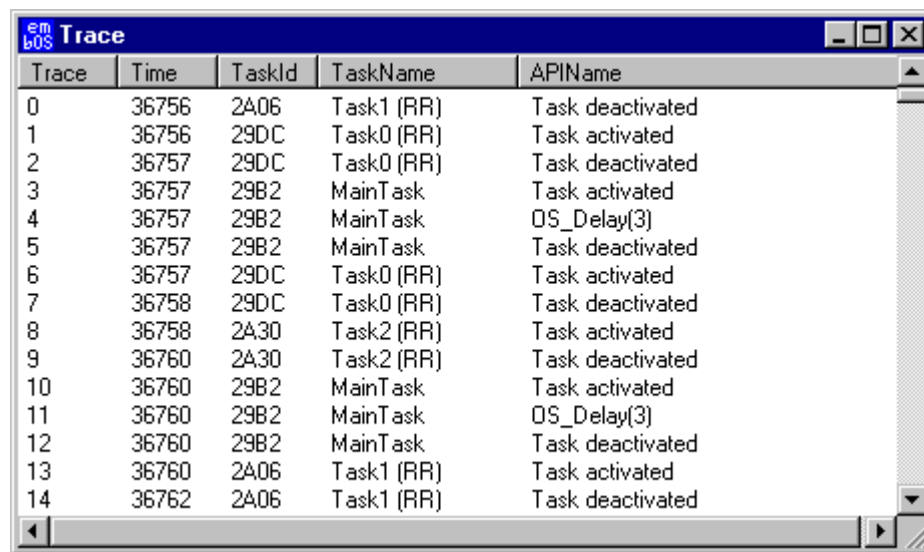
20.5. Using the API trace

embOS versions 3.06 or higher contain a trace feature for API calls. 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 the `Trace` menu, or from within the application by using the 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 the trace is 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	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) are also recorded. Parameters of API calls are recorded if possible, and are shown as part of the `APIName` column. In the example above, this can be seen with `OS_Delay(3)`.

Once the trace buffer is full, trace is automatically stopped. The trace list and buffer can be cleared from embOSView.

Setting up trace from embOSView

Three different kinds of trace filters are defined for tracing. These filters can be set up from embOSView via the menu `Options/Setup/Trace`.

Filter 0 is not task-specific and records all specified events regardless of the task. As the Idle loop is not a task, calls from within the idle loop are not traced.

Filter 1 is specific for interrupt service routines, software timers 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.

Filters 2 to 4 allow trace of API calls from named tasks.
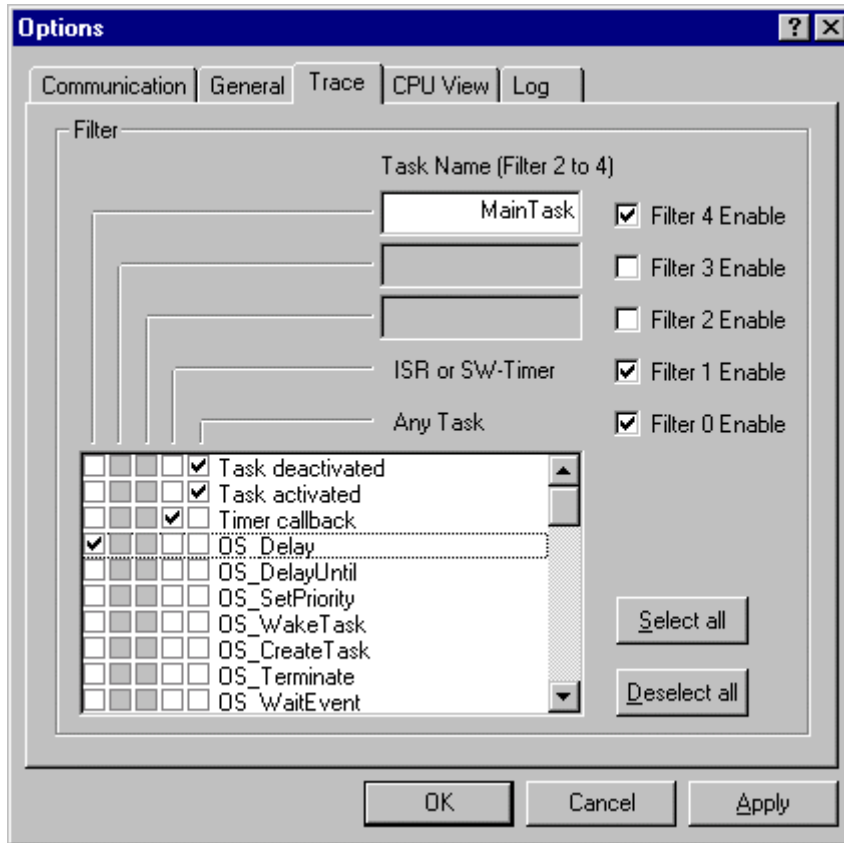
To enable or disable a filter, simply check or uncheck the corresponding check-boxes labeled `Filter 4 Enable` to `Filter 0 Enable`.

For any of these 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 currently enabled (checked) filters.
`Deselect all` disables trace of all API functions for the currently enabled (checked) filters.

Filter 2 to 4 allow tracing of task-specific API calls. A task name can therefore be specified for each of these filters. In the example above, Filter 4 is config-ured to trace calls of `OS_Delay()` from the task called `MainTask`.

After the settings are saved (via the `Apply` or `OK` button), the new settings are sent to the target application.

20.6. 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, using the following functions.

20.6.1. OS_TraceEnable()

Description

Enables tracing of filtered API calls.

Prototype

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

Add. information

The trace filter conditions should have been set up before calling this function. This functionality is available in trace builds only. In non-trace builds, the API call is removed by the preprocessor.

20.6.2. OS_TraceDisable()

Description

Disables tracing of API and user function calls.

Prototype

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

Add. information

This functionality is available in trace builds only. In non-trace builds, the API call is removed by the preprocessor.

20.6.3. OS_TraceEnableAll()

Description

Sets up Filter 0 (any task), enables tracing of all API calls and then enables the 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 non-trace builds, the API call is removed by the preprocessor.

20.6.4. OS_TraceDisableAll()

Description

Sets up Filter 0 (any task), disables tracing 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 non-trace builds, the API call is removed by the preprocessor.

20.6.5. OS_TraceEnableId()

Description

Sets the specified ID value in 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 must 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 non-trace builds, the API call is removed by the preprocessor.

20.6.6. OS_TraceDisableId()

Description

Resets the specified ID value in 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 must 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 non-trace builds, the API call is removed by the preprocessor.

20.6.7. OS_TraceEnableFilterId()

<u>Description</u>

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

<u>Prototype</u>

```
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*.

<u>Add. information</u>

To enable trace of a specific *embOS* API function, you must 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 non-trace builds, the API call is removed by the preprocessor.

20.6.8. OS_TraceDisableFilterId()

<u>Description</u>

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

<u>Prototype</u>

```
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*.

<u>Add. information</u>

To disable trace of a specific *embOS* API function, you must 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 non-trace builds, the API call is removed by the preprocessor.

20.7. 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 your own functions with your own parameters, you may call one of these routines.

All of these functions have the following points in common:

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

20.7.1. OS_TraceVoid()

Description

Writes an entry identified only 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: 0 <= 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.

20.7.2. 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: 0 <= 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 non-trace builds, the API call is removed by the preprocessor.

20.7.3. 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: 0 <= 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 non-trace builds, the API call is removed by the preprocessor.

20.7.4. 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: 0 <= 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 non-trace builds, the API call is removed by the preprocessor.

20.7.5. 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: 0 <= 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

This function may be used to record two pointers.

The values passed as parameter will be displayed in the trace list window of embOSView.

This functionality is available in trace builds only. In non-trace builds, the API call is removed by the preprocessor.

20.8. Application-controlled trace example

As described in the previous section, the user application can enable and set up the trace conditions without the need of a connection or command from embOSView. The trace record functions can also be called from any user function to write data into the trace buffer, using ID numbers from 100 to 127.

Controlling trace from the application can be very helpful for tracing API and user functions just after starting the application, when the communication to embOSView is not yet available or when the embOSView setup is not complete.

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

```
#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() */
}
```

Per default, embOSView lists all user function traces in the trace list window as `Routine`, followed by the specified ID and two parameters as hexadecimal values. The example above would result in the following:

```
Routine100(0xabcd, 0x01)
```

where 0xabcd is the pointer address and 0x01 is the parameter recorded from `OS_TraceDataPtr()`.

20.9. User-defined functions

In order to be able to use the built-in trace (available in trace builds of *embOS*) for application program user functions, embOSView can be customized. This customization is done in the setup file embOS.ini.

This setup file is parsed at the startup of embOSView. It is optional; you will not see an error message if it cannot be found.

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 following 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 preceeded by a colon.
#
API( 100, "Routine100")
API( 101, "Routine101", int)
API( 102, "Routine102", int, ptr)
```

21. Debugging

21.1. Run time errors

Some error conditions can be detected during run time. 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 with 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 that significant, but there is a difference). If *embOS* detects a run time error, it calls the following routine:

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

This routine is shipped in source as part of the module `OS_Error.c`. It simply disables further task 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);
}
```

If you are using embOSView, you can see the 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.
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. Simply reset this variable to 0 using your in circuit-emulator, and you can easily step back to the program sequence causing the problem. Most of the time, looking at this part of the program will make the problem clear.

21.2. 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, not initialized or overwritten.
129	OS_ERR_INV_TIMER	Timer control block invalid, not initialized or overwritten.
130	OS_ERR_INV_MAILBOX	Mailbox control block invalid, not initialized or overwritten.
132	OS_ERR_INV_CSEMA	Control block for counting semaphore invalid, not initialized or overwritten.
133	OS_ERR_INV_RSEMA	Control block for resource semaphore invalid, 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()`.
136	OS_ERR_MAILBOX_DELETE	`OS_DeleteMB()` was called on a mailbox with waiting tasks.
137	OS_ERR_CSEMA_DELETE	`OS_DeleteCSema()` was called on a counting semaphore with waiting tasks.
138	OS_ERR_RSEMA_DELETE	`OS_DeleteRSema()` was called on a resource semaphore which is claimed by a task.
140	OS_ERR_MAILBOX_NOT_IN_LIST	The mailbox is not in the list of mailboxes as expected. Possible reasons may be that one mailbox data structure was 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 inter-

Value	Symbolic name	Explanation
		rupts 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 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.
180	OS_ERR_NESTED_RX_INT	OS_Rx interrupt handler for embOSView is nested. Disable nestable interrupts.
190	OS_ERR_MEMF_INV	Fixed size memory block control structure not created before use.
191	OS_ERR_MEMF_INV_PTR	Pointer to memory block does not belong to memory pool on Release
192	OS_ERR_MEMF_PTR_FREE	Pointer to memory block is already free when calling OS_MEMF_Release(). Possibly, same pointer was released twice.
193	OS_ERR_MEMF_RELEASE	OS_MEMF_Release() was called for a memory pool, that had no memory block allocated (All available blocks were already free before).
200	OS_ERR_SUSPEND_TOO_OFTEN	Nested call of OS_Suspend() exceeded OS_MAX_SUSPEND_CNT
201	OS_ERR_RESUME_BEFORE_SUSPEND	OS_Resume() called on a task that was not suspended.

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

22. 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 compilers 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 and we will do our best to satisfy your needs in the shortest possible time.

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 it is 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 need to have non-reentrant routines 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 compiler performance). It is recommended not to use these options, although it is possible to do so 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.

23. Limitations

Max. no. of tasks:	limited by available RAM only
Max. no. of priorities:	255
Max. no. of semaphores:	limited by available RAM only
Max. no. of mailboxes:	limited by available RAM only
Max. no. of queues:	limited by available RAM only
Max. size. of queues:	limited by available RAM only
Max. no. of timers	limited by available RAM only
Event flags :	8 bits / task

We appreciate your feedback regarding possible additional functions and we will do our best to implement these functions if they fit into the concept.

Please do not hesitate to contact us. If you need to make changes to *embOS*, the full source code is available.

24. Source code of kernel and library

embOS is available in two versions:

1. Object version: Object code + hardware 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.

The source code distribution of *embOS* contains the following additional files:
- **CPU** folder contains all CPU and compiler specific source code and header files used to build the *embOS* libraries. It also contains the sample start project, workspace and source files for the *embOS* demo project delivered in the Start folder. Normally, you should not modify any of the files in the CPU folder.
- **GenOSSrc** folder contains all *embOS* sources and a batch file used to compile all of them in batch mode as described in the following chapter.

24.1. Building *embOS* libraries

The *embOS* libraries can only be built if you 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 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.

The whole build process is controlled with a few amount of batch files which are located in the root directory of your source code distribution:

- **Prep.bat**: Used to set up the environment for compiler, assembler and linker. Please ensure, that this file sets the path and additional include directories which are needed for your compiler. Normally, this batch file is the only one which might have to be modified to build the *embOS* libraries. Normally, this file is called from `M.bat` during the build process of all libraries.
- **Clean.bat:** Deletes the whole output of the *embOS* library build process. It is called automatically during the build process, before new libraries are generated. Normally it deletes the `Start\` folder. Therefore be careful not to call this batch file accidentally. Normally this file is called initially by `M.bat` during the build process of all libraries.
- **cc.bat**: This batch file calls the compiler and is used to compile one *embOS* source file without debug information output. Most compiler options are defined in this file and should normally not be modified. For your purposes, you might activate debug output and may also modify the optimization level. All modifications should be done with care. Normally this file is called from the *embOS* internal batch file `CC_OS.bat` and can not be called directly.
- **ccd.bat**: This batch file calls the compiler and is used to compile `OS_Global.c` which contains all global variables. All compiler settings are equal to those used in cc.bat, except debug output is activated to enable debugging of global variables when using *embOS* libraries. Normally this file is called from the *embOS* internal batch file `CC_OS.bat` and can not be called directly.
- **asm.bat**: This batch file calls the assembler and is used to assemble the assembly part of *embOS* which normally contains the task switch functionality. Normally this file is called from the *embOS* internal batch file `CC_OS.bat` and can not be called directly.
- **MakeH.bat**: Used to build the *embOS* header file `RTOS.h` which is composed from the CPU/compiler specific part `OS_Chip.h` and the generic part `OS_RAW.h`. Normally `RTOS.h` is output in the subfolder `Start\Inc`.
- **M1.bat**: This batch file is called from `M.bat` and is used to build one specific *embOS* library, it can not be called directly.
- **M.bat**: This batch file has to be called to generate all *embOS* libraries. It initially calls `Clean.bat` and therefore deletes the whole `Start` folder. The generated libraries are then placed in a new `Start\` folder which contains start projects, libraries, header and sample start programs.

24.2. Major compile time switches

Many features of *embOS* may be modified by compile time switches. All of them are predefined to reasonable values in the distribution of *embOS*.
The compile time switches must not be changed in `RTOS.h`.
When compile time switches should be modified to alter any of the *embOS* features, the modification has to be done in `OS_RAW.h` or has to be passed as parameter during the library build process. *embOS* sources have to be re-compiled and `RTOS.h` has to be rebuilt with the modified switches.

24.2.1. OS_RR_SUPPORTED

This switch defines whether round robin scheduling algorithm is supported. All *embOS* versions enable round robin scheduling per default. When you never use round robin scheduling and all of your tasks run on different individual pri-

orities, you may disable round robin scheduling by defining this switch to 0. This will save RAM, ROM and will also speed up the task switching process.
Please ensure, that none of your tasks ever run on the same priority when you disable round robin scheduling.
This compile time switch must not be modified in `RTOS.h`. It can only be modified in `OS_RAW.h` before *embOS* libraries are rebuilt.

24.2.2. OS_SUPPORT_CLEANUP_ON_TERMINATE

This compile time switch is new since version 3.26 of *embOS*. If enabled, it allows termination of tasks which are claiming resource semaphores or are suspended on any synchronization object.
Per default, this switch is activated for 16- and 32-bit CPUs.
For 8-bit CPUs it is disabled.
Even though the overhead is minimal and execution time is not affected significantly, you may define this switch to zero when you do not terminate tasks in your application, or if your application ensures, that tasks are never suspended on any synchronization object or claim any resource semaphores when they are terminated.

Disabling this switch will save some RAM in the task control structure and will also speed up the wait functions for synchronization objects.
When using an 8bit CPU, you have to enable this switch (define it to be unequal to 0) to enable termination of tasks which are suspended on synchronization objects or claim resource semaphores.

This compile time switch must not be modified in `RTOS.h`. It can only be modified in `OS_RAW.h` or has to be passed as "define" during the build process when *embOS* libraries are rebuilt.

25. Additional modules

25.1. 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 portion. It needs to be initialized on startup and creates a task that checks the keyboard 50 times per second.

Changes required for your hardware

```
void ReadKeys(void);
```

Example of how to implement into your program

```
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();
}
```

25.2. 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 on any target CPU for which an ANSI "C" compiler exists. 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. A 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, dec/hex/bin values, multiple windows
- ultra-fast, yet still very compact (typically between 8 and 20 kB ROM)

Everything you need for graphic displays!
Any LCD * Any LCD controller * Any CPU
Both monochrome and color versions available, as well as bitmapconverter, font converter, PC simulation and viewer. Check out our website!

emLoad

Boot-loader software.

26. 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 so 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.

Glossary

Some technical terms used in this manual are explained below.

Active Task	Only one task can execute at any given time. The task that is currently executing is called the active task.
Cooperative multitasking	A scheduling system in which each task is allowed to run until it gives up the CPU; an ISR can make a higher priority task ready, but the interrupted task will be returned to and finished first.
Counting semaphore	A type of semaphore that keeps track of multiple resources. Used when a task must wait for something that can be signaled more than once.
CPU	Central Processing Unit. The "brain" of a microcontroller; the part of a processor that carries out instructions.
Critical region	A section of code which must be executed without interruption.
Event	A message sent to a single, specified task that something has occurred. The task then becomes ready.
ISR	Interrupt Service Routine. The routine called automatically by the processor when an interrupt is acknowledged. ISRs must preserve the entire context of a task (all registers).
Mailbox	A data buffer managed by the RTOS, used to send a message to a task.
Message	An item of data (sent to a mailbox, queue, etc.).
Multitasking	The execution of multiple software routines independently of one another. The OS divides the processor's time so that the different routines (tasks) appear to be happening simultaneously.
NMI	Non-Maskable Interrupt. An interrupt that cannot be masked (disabled) by software. Example: Watchdog timer-interrupt.
Preemptive multitasking	A scheduling system in which the highest priority task that is ready will always be executed. If an ISR makes a higher priority task ready, that task will be executed before the interrupted task is returned to.
Processor	Short for microprocessor. The CPU core of a controller
Priority	The relative importance of one task to another. Every task in an RTOS has a priority.
Priority inversion	A situation in which a high priority task is delayed while it waits for access to a shared resource which is in use by a lower priority task. The lower priority task temporarily gets the

<table>
<tr><td></td><td>highest priority until it releases the resource.</td></tr>
</table>

Queue	Like a mailbox, but used to send one or more messages to a task.
Resource	Anything in the computer system with limited availability (e.g. memory, timers, computation time). Essentially, anything used by a task.
Resource semaphore	A type of semaphore used to manage resources by ensuring that only one task has access to a resource at a time.
RTOS	Real Time Operating System.
Scheduler	The program section of an RTOS that selects the active task, based on which tasks are ready to run, their relative priorities, and the scheduling system being used.
Semaphore	A data structure used for synchronizing tasks.
Software timer	A data structure which calls a user-specified routine after a specified delay.
Stack	An area of memory with FIFO storage of parameters, automatic variables, return addresses, and other information that needs to be maintained across function calls. In multitasking systems, each task normally has its own stack.
Superloop	A program that runs in an infinite loop and uses no real time kernel. ISRs are used for real time parts of the software.
Task	A program running on a processor. A multitasking 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) for which a task will be executed until a round-robin task change may occur.

Index