RTOS Library for PIC32

2015-08-09

Content

1 Important

- Since the any project using RTOS will include *.inc files and at least one project level defines (.pld) file it is necessary to have the "**Always build all files in the project**" compiler option (IDE menu: Tools, Options, Output settings) set to **ON**.
- In case you would like to make changes yourself in the RTOS library: see also section 11.3.1.

2 Overview of RTOS types, constants, variables and functions

3 Introduction

This RTOS is a **R**eal **T**ime **O**perating **S**ystem designed for and written specifically in mikroPascal from mikroElektronika. The system uses pre-emptive scheduling which means that the tasks with a high priority will interrupt (pre-empt) tasks of which the priority is lower.

Writing code for a RTOS requires a different mindset from that used when writing a single threaded application. However, once you have come to terms with this approach you will find that quite complex real time systems can be developed quickly using the services of the operating system.

The whole library is targeted at the **PIC32** series of MPUs. It is written mostly in mikroPascal (only small parts in assembler), so changes should be easy to implement.

3.1 Why should I use RTOS?

RTOS can give you the potential opportunity to squeeze more from your PIC than you might expect from your current single threaded application. For example, how often do your programs spend time polling for an input or an event. If you could have the Operating System tell you when an event has taken place you could use that polling time to do other things. This applies equally well to delays. By using RTOS you can write programs which appears to be doing many things all apparently at the same time.

Some of this can be achieved in a single threaded program by using interrupts but by using RTOS together with interrupts you will have be able to quickly develop responsive applications which are easy to maintain.

3.2 RTOS Fundamentals

This section describes the fundamentals of RTOS.

A typical program written in mE' mikroPascal would use a looping main program calling subroutines from the main loop. Time critical functions would be handled separately by interrupts. This is fine for simple programs but as the programs become more complex the timing and interactions between the main loop background and the interrupt driven foreground become increasingly more difficult to predict and debug.

RTOS gives you an alternative approach to this where your program is divided up into a number of smaller well defined functions or tasks which can communicate with each other and which are managed by a single central scheduler.

3.2.1 Some Basic Definitions

The fundamental building block of RTOS are **Tasks** (see also sections 3.2.2 and 7). Tasks are a discrete set of instructions that will perform a recognized function, e.g. Process a keypad entry, write to a display device, output to a peripheral or port etc. It can be considered in effect a small program in its own right which runs within the main program. Most of the functionality of a RTOS based program will be implemented in Tasks.

In RTOS a Task can have a **Priority** (see section 7.7) which determines its order of precedence with respect to other tasks. Thus you can ensure your most time critical tasks get serviced in a timely manner.

Interrupts are events which occur in hardware which cause the program to stop what it was doing and vector to a set of instructions (the Interrupt service routine ISR) which are written to respond to the interrupt. As soon as these instructions have been executed the control is returned to the main program at the point where it was interrupted. This is no different in RTOS.

A **Task switch** (see also section 7.9) occurs when one task is **Suspended** (made not runnable or not eligible) and another task is **Started or Resumed** (made runnable or eligible). This is core functionality to an RTOS. In the RTOS for PIC32 the action of task switching is both co-operative and time slice based. The first means that your tasks can Yield back to RTOS when e.g. they wait for something to happen.
The second one is a forced task switching every xx milliseconds (in RTOS this happens every 10 milliseconds).

Tasks can call for a Delay (see also section 7.9.2) which will suspend the task until the delay period has expired and will then resume from where it left off. This is similar to the Delay_ms or Delay_us functions in mP except that during the delay the processor can be assigned another task until that delay period is up. In practice it is most likely that delays will be defined in the ms or 10s of milliseconds as delays in the low microseconds would make Task switching (including its context switching) very inefficient.

An Event is the occurrence of something such as a serial data receipt, or an error has occurred or a long calculation or process has completed. An event can be almost anything and can be raised (Signaled) by any part of the program at any time. When a task waits on an event it can assign a Timeout so that the task can be released from being stuck waiting for an event which isn't going to happen for some reason.

Inter-task Communication provides a means for tasks to communicate with other tasks.
RTOS supports Semaphores (see section 8).
Semaphores can take 2 forms, **Binary and Counting Semaphore**. A binary semaphore can be used to signal actions like a button has been pressed or a value is ready to be processed. The task waiting on the event will then suspend until the event occurs when it will run. A counting semaphore can carry a value; typically it could be used to indicate the number of bytes in an input buffer. When the value of a counting semaphore has reached zero it becomes "not signaled".

3.2.2 More about Tasks

Example of a task in RTOS:

```
procedure UsefulTask:
begin
  while true do
  begin
     // do something useful(payload)
    OS_Yield; // yield to RTOS, context switch (optionally)
  end;
end;
```

This code will perform its operation and then Yield to the operating system. RTOS will then decide when to run it again. If there are no other tasks to run it will return to the original task. OS_Yield is one of a number of mechanisms for relinquishing control back to the operating system (next to the “timebase tick” based task switching, if not disabled by the OS_NO_AUTO directive).

In its simplest form a multitasking program could comprise just 2 or more tasks each taking their turn to run in a Round-Robin sequence. This is of limited use and is functionally equivalent to a single threaded program running in a main loop. However, RTOS allows Tasks to be assigned a priority which means you can ensure that the processor is always executing the most important task at any point in time.

Clearly, if all your tasks were assigned the highest priority you would be back to running a round-robin single loop system again but in real life applications, tasks only need to run when a specific event occurs. E.g. User entered data or a switch has changed state. When such actions occur the task which needs to respond to that action must run. The quicker the response needed then the higher the priority assigned to the task. This is where a multitasking RTOS starts to show significant advantages over the traditional single threaded structure.

More task details can be found in section 7.

4 RTOS Features

RTOS supports the following features:

- Task **priorities**
- **Pre-emptive** scheduling: a (low priority) task will be interrupted by a task with higher priority
- **Automatic task switching** very 10 milliseconds, under interrupt. Can be switched off, see OS_NO_AUTO. Switching it off this feature makes the RTOC behave as a CRTOS (Cooperative real time operating system).
- Round Robin scheduling within tasks of the same priority.
- Tasks can have local variables of which the values are preserved between task invocations
- Unconditional yielding to the RTOS Scheduler
- Timed yielding to the RTOS Scheduler (i.e. “delay” performed by the Scheduler)
- Binary semaphores
- Counting semaphores
- Waiting for semaphores with or without timeout
- Clearing semaphores
- Starting and stopping tasks
- Starting tasks with an initial delay
- Signaling semaphores from within an Interrupt Service Routine
- An “Idle” procedure can be called when no task is running (none is eligible for running)
- Critical Sections

5 RTOS Library files

The RTOS library consists of 1 pascal (.mpas) file: **RTOS.mpas**.

Make sure the RTOS.mpas library file is included in mP’s project manager in the IDE (if RTOS has not been installed as a “library”), or checked in the library manager (if RTOS has been installed as a “library” with the package manager).

6 RTOS Usage in a program

Important:

Since the project will include at least one include file (*.inc) and at least one project level defines (.pld) file it is necessary to have the “Always build all files in the project” compiler option (IDE menu: Tools, Options, Output settings) set to ON.

6.1 Overview

In a user application the following has to be done:

- The RTOS configuration for the project has to be defined: two project files need to be made and entered in the Project Manager: "RTOS_Defines.pld" and "RTOS_Sizes.inc", see section 10 for the details.
- The task variables and the semaphore variables (if used) have to be declared.
- The RTOS "Timebase": RTOS always uses the Timer1 interrupt.
  The only procedure the user has to provide is the routine "InitTimer1" (with that actual name). Make sure the IP bits are set for an interrupt priority of 6! (see section 6.2.3)

  You do not have to create a Timer1 interrupt routine. RTOS takes care of that.
  In all examples a Timebase tick of 1 millisecond is used. This means that automatic context switching will happen every 10 milliseconds (unless switched off, see OS_NO_AUTO).
- The "tasks" have to be defined. They are ordinary mP procedures, with no parameters, except for the following:
  - Each task must contain an endless loop (while true do begin... end)
  - In that loop the actual work of the task is done (the code before the loop is only executed the very first time the task is run)
  - In the loop there can be calls to RTOS functions that lead to yielding to the Scheduler. However this is not obligatory (automatic task switching every 10 timebase ticks).
  - The tasks are called indirectly by the Scheduler, so each task should be subject to a "SetFuncCall" statement. Do not call tasks directly.
- The Memory manager has to be started (library delivered with the compiler). This is needed when tasks are created to reserve space for the stack of each task.
- The RTOS has to be initialised.
- The tasks have to be "created" (added to the RTOS task list)
- The Semaphores (if any) have to be "created" (added to a RTOS semaphore list)
- The tasks have to be started (only the ones that are needed to be started when the Scheduler starts to run of course)
- The Scheduler has to be started. At this points the Timer1 will be initialised and tasks will be actually executed.

6.2 Example

6.2.1 The RTOS configuration for the project

Two project files need to be made and entered in the Project Manager: "RTOS_Defines.pld" and "RTOS_Sizes.inc", see section 10 for the details.

6.2.2 Task variables and semaphore variables

```
var T_LedOut, T_OscOut, T_Delayed: TOS_Task; // tasks
    E_LedCtrl: TOS_BinarySemaphore;          // semaphores
```

6.2.3 The Timer1 init routine

```
//Timer1 will give the timer tick for RTOS

// Values for Mini32 board (80 Mhz CPU clock and "Peripheral Clock Divisor" Pb_Clk is
Sys_Clk/1): //
// Prescaler 1:8; PR1 Preload = 10000; Actual Interrupt Time = 1 ms
procedure InitTimer1();      // <--- important: procedure name NOT to be changed!
begin
  T1CON         := 0x8010;
  T1IE_bit      := 1;
  T1IF_bit      := 0;
  T1IP0_bit     := 0;        // <--- important: interrupt priority 6 !!!!!!!!!!
  T1IP1_bit     := 1;        //
  T1IP2_bit     := 1;        //
  PR1           := 10000;
  TMR1          := 0;
end;
```

6.2.4 Definition of the tasks.

```
procedure LedOut;
begin
  while true do // endless loop
  begin
    LedOut_bit := not LedOut_bit; // payload of the task
    OS_Yield; // unconditional yield to the RTOS scheduler (optional)
  end;
end;

procedure OSCOut;
begin
  while true do // endless loop
  begin
    OscOut_bit := not OscOut_bit; // payload of the task
    OS_Yield;  // unconditional yield to the RTOS scheduler (optional)
  end;
end;

procedure DelayedTask;
begin
  while true do // endless loop
  begin
    DelayedOut_bit := not DelayedOut_bit; // payload of the task

    OS_StartTask(T_OscOut);               // start another task
    OS_DELAY(20);                         // yield, delay performed by the RTOS scheduler

    DelayedOut_bit := not DelayedOut_bit; // payload of the task

    OS_StopTask(T_OscOut);                // stop another task
    OS_DELAY(100);                        // yield, delay performed by the RTOS scheduler

    OS_SignalSemaphore(E_LedCtrl);        // signal a semaphore
```

```
  end;
end;

procedure BinSemTask;
begin
  while true do
  begin
    OS_WaitSemaphore(E_LedCtrl, OS_NO_TIMEOUT);

    OS_StartTask(T_LedOut); // starting of another task
    OS_DELAY(50);

    OS_StopTask(T_LedOut); // stopping of another task
  end;
end;
```

6.2.5 Initialisation of RTOS

```
OS_Init;
```

6.2.6 Signal indirect calling of tasks by the RTOS scheduler to the compiler

```
SetFuncCall(LedOut);
SetFuncCall(OSCOut);
SetFuncCall(DelayedTask);
SetFuncCall(BinSemTask);
```

6.2.7 Creation of tasks

```
T_LedOut    := OS_CreateTask(@LedOut, 3);
T_OscOut    := OS_CreateTask(@OscOut, 3);
T_Delayed   := OS_CreateTask(@DelayedTask, 2);
T_BinSem    := OS_CreateTask(@BinSemTask, 3);
```

The 2nd parameter of the function “OS_CreateTask” is the task priority, see section 7.7.

6.2.8 Creation of semaphores

```
E_LedCtrl := OS_CreateBinarySemaphore(False);

See section 8 for explanation of this routine.
```

6.2.9 Starting of tasks

```
OS_StartTask(T_Delayed);
OS_StartTask(T_BinSem);
```

6.2.10 Start the RTOS scheduler

```
OS_Run;                 // not in a loop!
                        // OS_Run is a blocking task (not left)
```

6.3 Explanation of example code

See the example code above (section 6.2).

2 tasks are running continuously: “T_Delayed” and “T_BinSem”.

“T_Delayed” starts task “T_OscOut” and stops it again after 20 millisecs. The total period of that is 120 millisecs. Furthermore “T_Delayed” signals semaphore “E_LedCtrl” every 120 millisecs.
Additionally “T_Delayed” toggles the “DelayedOut_bit” before starting and stopping “T_OscOut”.

The “T_BinSem” task waits for the semaphore signaled by the “T_Delayed” task. When “E_LedCtrl” is signaled “T_BinSem” starts the “T_LedOut” tasks and stops it again after 50 millisecs. This means that T_LedOut” will run for 50 millisecs with a total period of 120 millisecs.

The task “T_OscOut” simply toggles “OscOut_bit” in an endless loop until stopped by RTOS (i.e. 20 millisecs, every 120 millisecs).

The Task “T_LedOut” toggles “LedOut_bit” in every endless loop until stopped by RTOS (i.e. 50 millisecs, every 120 millisecs).

The toggling of Pic ports “DelayedOut_bit”, “LedOut_bit” and “OscOut_bit” acts here as payload of the tasks. Their status can be observed with an oscilloscope. They are defined e.g. as follows:

```
var LedOut_bit     : sbit at LatA.0;
    OSCOut_bit     : sbit at LatA.1;
    DelayedOut_bit : sbit at LatB.0;
```

Of course those ports have to be set to “output”:

```
var LedTris_bit    : sbit at TrisA.0;
    OSCTris_bit    : sbit at TrisA.1;
    DelayedTris_bit: sbit at TrisB.0;

...

// define ports as output
  LedTris_bit    := 0;
  OSCTris_bit    := 0;
  DelayedTris_bit:= 0;
```

On an oscilloscope the behavior can be observed:

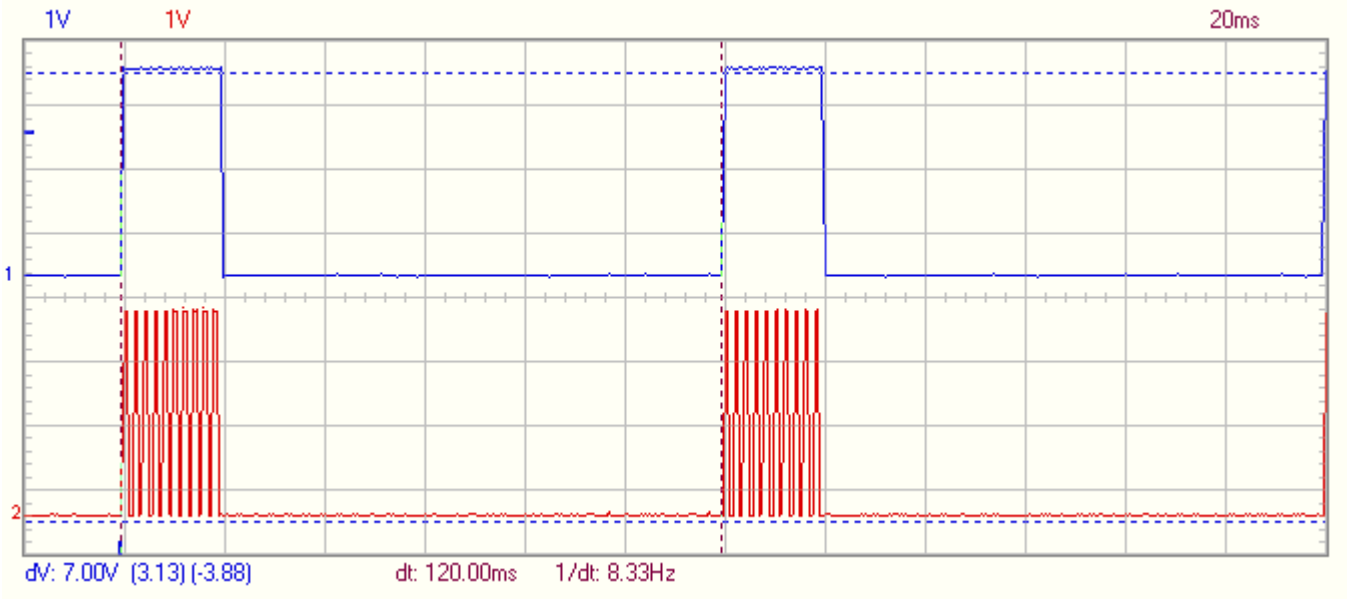


Figure 1: DelayedOut_bit (blue) and OscOut_bit (red)

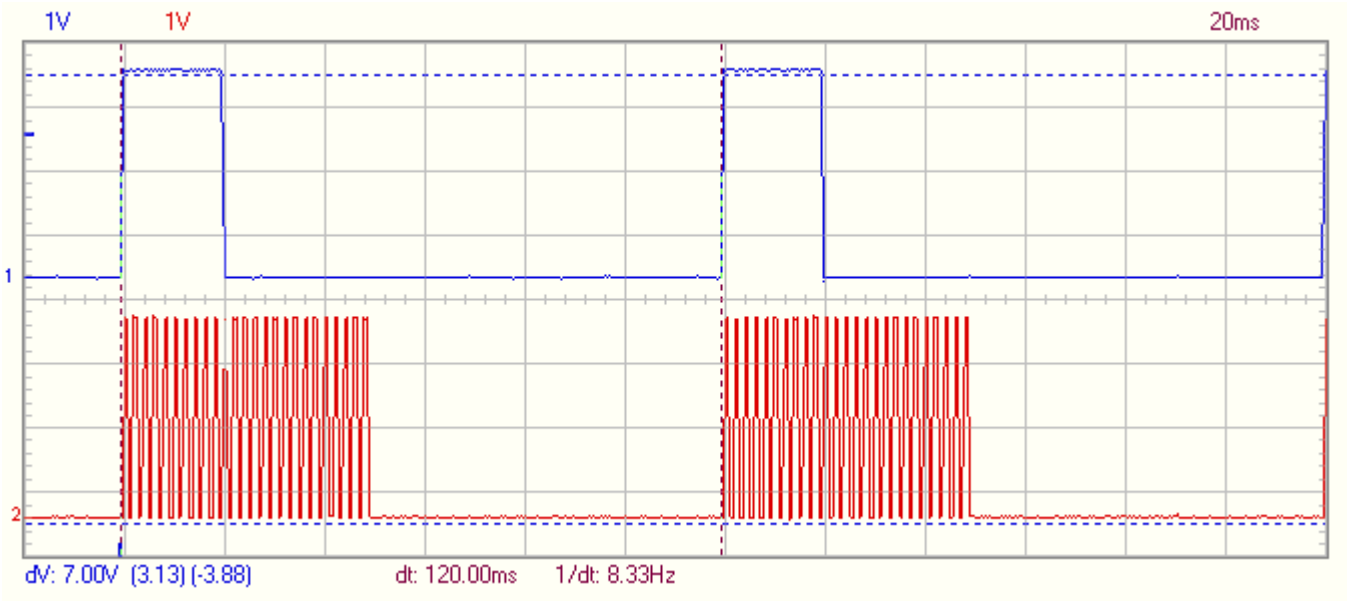


Figure 2: DelayedOut_bit (blue) and LedOut_bit (red)

As one can see the tasks “T_OscOut” and “T_LedOut” are started and stopped by other tasks and run simultaneously for a while.

7 Tasks

Tasks are the most important items in the application using RTOS; they contain the actual code to the executed (the payload). This requires the application author to analyze the application requirement and break the application into a set of inter-related co-operating tasks.

When using RTOS the tasks have to obey the following rules:

- they are Procedures (not Functions) with no parameters
- they must contain an infinite loop (e.g. “while true do begin ... end;”)
- they can contain, in that loop, one or more statement that can cause a yield to the RTOS Scheduler. This is optional (there is also the automatic task switching every 10 timebase ticks, unless this is disabled – see the OS_NO_AUTO directive).
- if the task is not the lowest priority and runs continuously, ensure that it actually yields once in a while to the Scheduler without becoming eligible immediately again (prevent starvation of tasks with a lower priority).
- Tasks can have local variables of which the values are to be preserved between task invocations.

Remark: The task can have code before the infinite loop. This code will be executed only once: the first time the task runs.

Tasks can be created, started, stopped and their priorities changed. They have a State (see section 7.6) and a Priority (see section 7.7).

The current task (the one executing) can be fetched with the function

```
CurTask := OS_CurrentTask;
```

Important:

- Tasks can call other subroutines without any problem, but those routines should:
  - NOT have endless loops
- Tasks can NOT be called directly from e.g. an other task, they can only be handled by the scheduler.
- Procedures or functions called from within tasks can have parameters and local variables.

7.1 Creation of a task

```
T_SomeTask    := OS_CreateTask(@SomeTask, 3);
```

Here “T_SomeTask” is of type TOS_Task, “@SomeTask” is the address of the task procedure, and 3 is the (initial) priority of the task.

Before the RTOS Scheduler is started a number of tasks can be started, but tasks can also be started while the RTOS Scheduler is running (provided there is some other task or event to start stopped tasks). Tasks will not actually run until RTOS is started.

7.2 Starting a task

```
OS_StartTask(T_SomeTask); // T_SomeTask is candidate to be eligible (candidate to be executed)
```

7.3 Starting a task with an initial delay

Sometimes a task is to be run “after a certain time”. This is achieved with:

```
OS_StartTask_Delay(T_SomeTask, 100); // T_SomeTask is candidate to be eligible (candidate to
be executed)after 100 timebase ticks
```

7.4 Stopping a task

```
OS_StopTask(T_SomeTask); // T_SomeTask will never be eligible for execution
```

When the task will be resumed (with OS_StartTask or OS_StartTask_Delay) then it will do so with the statement after OS_StopTask.

OS_StopTask can only be called from within an actual task procedure.

7.5 Stopping the current task and starting another one

```
OS_ReplaceTask(T_SomeOtherTask); // the current task is stopped and task T_SomeOtherTask is
                                 // started
```

OS_ReplaceTask can only be called from within an actual task procedure.

7.6 Task States

Tasks can be in different states:

- Destroyed: the task is not created yet
- Stopped: the task has been created but not started yet (or has been stopped)
- Delayed: the task has been started but is suspended for a period (waiting for n timebase ticks due to OS_DELAY)
- Waiting: the task has been started and is waiting a semaphore signaling (forever or with time-out)
- Running: the task is currently active (executing)

The state of a task can be fetched with:

```
OS_TaskState(SomeTask);
```

7.7 Task Priorities

Priorities of tasks range from 0 (zero) to n (n = OS_PRIORITY_COUNT – 1, see section 10). Zero is the highest priority, 1 is the second highest etc. and n is the lowest one.

Each task has a certain priority assigned (unless RTOS is configured otherwise).

Tasks with a higher priority always have precedence over tasks with a lower priority. Tasks with the same priority are executed in a "Round Robin" manner: each task is executed in the same sequence as it was created (the RTOS Scheduler remembers which task for a certain priority was executed last and takes the next in the same priority). After startup of the RTOS Scheduler the first task for each priority to be executed is the first one created with that priority.

The consequence of this is that as long as a task of a certain priority is "eligible" (candidate for execution) tasks with a lower priority are not run. This can lead to "starvation' of the lower priority tasks. This means that tasks with a higher priority should (at least once in a while) yield to the RTOS Scheduler (e.g. with OS_DELAY, or with OS_WaitSemaphore) without being immediately eligible again.

The priority of a task is defined at its creation time, but can be changed while the RTOS scheduler runs:

```
T_Task1 := OS_CreateTask(@Task1, 3);
OS_SetPriority(T_Task1, 2);
```

The first statements creates a task with priority 3, the second one changes its priority to 2.

The priority of a task can be fetched with:

```
 OS_Priority(SomeTask);
```

7.8 Execution Eligibility of a task

A started task is evaluated for eligibility by the RTOS Scheduler. The scheduler checks the following after a task yields to the RTOS scheduler:

- if a delay time has expired (task yielded with OS_DELAY)
- if a binary semaphore on which the task is waiting becomes signaled or a counting semaphore value on which the task is waiting becomes > 0 (task yielded with OS_WaitSemaphore)
- if waiting for a semaphore times out (task yielded with OS_WaitSemaphore)
- if the task unconditionally yields to the RTOS Scheduler with OS_Yield

If one of the above happens the task is (again) ready for execution, the task is **eligible** for execution.

This does not mean the task will be executed immediately, there could be more tasks that are eligible at the same time. So, all eligible tasks are waiting to be executed (as soon as possible).
The actual RTOS Scheduler will pick one of them according their priority, and that one will actually be executed.

A task is always eligible for execution after it has just been started with OS_StartTask.

7.9 Task Yielding to the scheduler

Each task can yield to the operating system inside its endless loop. This is not obligatory: there is also the automatic task switching mechanism every 10 timebase ticks.

BUT: In case the automatic task switching every 10 Timebase ticks is disabled (see the OS_NO_AUTO directive) then each task HAS to yield to the scheduler at some moment in its infinite loop.

This can be done in a number of ways:

7.9.1 Unconditionally

```
OS_Yield;
```

This type of yielding causes the scheduler suspends the task's execution and makes the task immediately eligible again for execution. Tasks with lower priority than this one have no chance to execution any more.

OS_Yield can only be called from within an actual task procedure.

7.9.2 Delayed

```
OS_DELAY(Ticks);
```

This causes the scheduler to suspend the task's execution for "Ticks" timebase ticks. After the ticks have expired, the task is made eligible again for execution. Here tasks with a lower priority have a chance for execution, since the scheduler waits a while before the task is made eligible again.

Also the function OS_StartTask_Delay yields to the scheduler during its initial delay.

OS_Delay can only be called from within an actual task procedure.

7.9.3 Waiting (for a semaphore)

```
OS_WaitSemaphore(SomeSem, 15);
```

Above method causes the scheduler to suspend the task's execution until the semaphore becomes signaled or 15 timebase ticks have passed (timeout). After one of both occurred the task is made eligible again for execution. Here tasks of a lower priority have a chance for execution (see the "Delayed" version above for the reason), provided the semaphore is not actually signaled when OS_WaitSemaphore is called, in which case the scheduler will make the task not eligible for a while.

OS_WaitSemaphore can only be called from within an actual task procedure.

7.9.4 Stopping the current task

If the current task is stopped then a yield to the scheduler will occur. This can be invoked by calling either one of these commands:

```
OS_StopTask(OS_CurrentTask); // or
OS_ReplaceTask(NewTask);
```

OS_StopTask and OS_ReplaceTask can only be called from within an actual task procedure.

7.10 Critical Sections

Critical sections in tasks can occur when several tasks do use the same resource, e.g. the same memory/variables, the same hardware resource –e.g. Uart, SPI -, etc...

In an interrupt driven OS, as this one is, a Timer1 interrupt can cause a task switch, which is undesirable in a critical section as described above, because those task switching's can result in unpredictable behavior.

There are two routines to mark such an critical section:
OS_EnterCriticalSection and OS_LeaveCriticalSection. They respectively disable and re-enable Timer1 Interrupts, so no task switching can occur within the critical section.
This also means that statements causing a task switching (e.g. OS_Delay etc...) will have no effect until after the critical section is left.

7.11 Subroutines called in tasks

Calling other routines in tasks is allowed, but there is one point of attention: if a routine is called by more than one task (directly or indirectly), then the routine should be thread safe, or it should be considered a critical resource (see above section).

7.12 Tasks are called indirectly

Important:
Tasks are always called indirectly, so mikroPascal has to be warned by inserting the statement "SetFuncCall", e.g.:

```
SetFuncCall (Task1, Task2, Task3, ...);
```

The SetFunCall statements have to be placed before creation of the tasks in the main procedure.
Do not call as task directly.

8 Semaphores

Semaphores are used to communicate between tasks, e.g. the finishing of a process, the availability of data, a time that has expired etc. Also shared resources (e.g. an SDcard) can be waited for and "grabbed" and "freed" after usage.

There are 2 types of semaphores: binary and counting:

- A binary semaphore can only be in the state "signaled" or "not signaled",
- a counting semaphore can be signaled more than once (it counts the number of times it was signaled).

8.1 Creation of a Semaphore

A binary semaphore is created as follows:

```
var  E_BinSem: TOS_BinarySemaphore;
...
E_BinSem := OS_CreateBinarySemaphore(False); // initial state = "not signaled"
```

A counting semaphore is created with:

```
var E_SomeSem : TOS_CountingSemaphore;
...
E_SomeSem := OS_CreateCountingSemaphore(5);  // initial count = 5 (normally zero!)
```

8.2 Signaling of a semaphore

Both types of semaphores are signaled by a task with

```
OS_SignalSemaphore(E_SomeSem);
```

In the case of a binary semaphore it will change state to "signaled", in case of a counting semaphore it will increment its count.

It is also possible to signal a semaphore from an ISR (Interrupt service routine) with

```
OS_SignalSemaphore_ISR(E_SomeSem);
```

8.3 Waiting for a Semaphore

Both types of semaphores can be waited for in (other) tasks with:

```
OS_WaitSemaphore(E_SomeSem, 20); // with a timeout of 20 timebase ticks
```

In case of a binary semaphore the statement will operate as follows:

- if the semaphore is already signaled at the time "OS_WaitSemaphore" is executed then the semaphore is cleared and the task is continued (no yield to the RTOS Scheduler)
- if the semaphore is not signaled at the time "OS_WaitSemaphore" is executed, it will yield to the RTOS Scheduler until the semaphore becomes signaled. When that happens, the semaphore will be cleared and the task that called "OS_WaitSemaphore" will be made "eligible" again (candidate for execution).

The behavior of a counting semaphore is very similar:

- If the value (count) of the semaphore is already >0 ("signaled") at the time "OS_WaitSemaphore" is executed, then the semaphore count is decremented and the task is continued (no yield to the RTOS Scheduler)
- If the semaphore count is zero (not signaled) at the time "OS_WaitSemaphore" is executed, it will yield to the RTOS Scheduler until the semaphore count becomes > 0. When that happens, the semaphore count will be decremented and the task that called "OS_WaitSemaphore" will be made "eligible" again (candidate for execution).

As one can see "signaling" a semaphore sets some kind of flag, while "wait for" suspends the execution of a task until that flag is set, and if so, clears it and continues execution. In case of counting semaphores the "flag" can be set multiple times.

OS_WaitSemaphore can only be called from within an actual task procedure.

8.4 Semaphore timeout

Waiting for a semaphore can be subject to a time limit. The procedure OS_WaitSemaphore stops execution of a task until the binary semaphore is signaled (or the counting semaphore value is > 0), or the timeout has expired. The occurrence of a timeout when the task continues executing can be tested with the boolean function OS_TIMEOUT.

Example:

```
...
OS_WaitSemaphore(E_SomeSem, 50);
if OS_TIMEOUT
then ... // the semaphore was not signaled in time
else ... // the semaphore was signaled in time
```

In above example the timeout is 50 ticks of the RTOS Timebase.

- If the semaphore E_SomeSem does not become signaled within those 50 ticks the RTOS Scheduler will make the calling task eligible for execution again and, when it actually runs again, the function OS_TIMEOUT will return “true”. The timeout flag will be reset automatically when calling OS_TimeOut.
- If the semaphore E_SomeSem becomes signaled within to Timebase ticks the RTOS Scheduler will make the calling task eligible for execution again and, when it actually runs again, the function OS_TIMEOUT will return “false”.

8.5 Clearing a semaphore

A semaphore can be cleared also without using “OS_WaitSemaphore”.
Example:

```
If OS_TrySemaphore(SomeSem) then // test if the semaphore is signaled
begin // if so, then...
  // do something
  OS_ClearSemaphore(SomeSem); //and clear the semaphore
end;
```

9 The “Idle” procedure

The possibility exists to let the scheduler call a procedure when no tasks can be run (all tasks are not eligible for running somehow, e.g. waiting for the end of an OS_Delay) after a task switch.
The “Idle” procedure:

- Is a “**procedure**”, not a “task”.
  - This means it has NO endless loop and
  - it does NOT yield to the scheduler, so, all RTOS routines mentioned in section 7.9 are forbidden.
- Can use other RTOS commands like starting of stopping a task.
- Must have a very low time consumption, since it is called continuously by the scheduler (it will delay task execution).
- Can call other routines (no tasks however).
- Only available when the OS_IDLE compiler directive is used.
- Is called indirectly (so, “SetFuncCall” is needed)

The idle procedure is “attached” to RTOS trough a procedure pointer called “OS_IdleProcedure”.

Example:

```
procedure MyIdleProcedure;
begin
   LatA.0 := not LatA.0; // payload
end;

...
SetFuncCall(MyIdleProcedure); // will be called indirectly

OS_IdleProcedure := @MyIdleProcedure; // attach the procedure to the “idle” hook.
```

```
...
OS_IdleProcedure := nil; // unhook the procedure
```

10 RTOS Configuration

RTOS uses a number of compiler directives which define the RTOS functionalities and *constants* which define the *list sizes*:

10.1 RTOS Functionalities

In RTOS executing tasks is always a supported functionality, without tasks there is no RTOS whatsoever. For other features there are the following compiler directives, to be found in the file "`RTOS_Defines.pld`" (to be present in the main project file's directory):

RTOS_Defines.pld	meaning
OS_SEMAPHORES	allows semaphores
OS_PRIORITIES	if there is more than one priority
OS_DELAY	needed for "OS_DELAY"
OS_TIMEOUT	needed for "OS_WaitSemaphore" with timeout
OS_SIGNAL_ISR	To enable "OS_SignalSemaphore_ISR"
OS_IDLE	enable the "idle" procedure, see section 9
OS_STACK_CHECK	Enable the task stack usage calculation

The above defines can be "undefined" by either deleting the definition or adding a minus sign ("-") just before it.

Example of a valid content of the file "RTOS_Defines.pld" (only one priority, no semaphore signaling from ISR's, no "Always procedure"):

```
OS_SEMAPHORES
-OS_PRIORITIES
OS_DELAY
OS_TIMEOUT
-OS_SIGNAL_ISR
-OS_IDLE
OS_STACK_CHECK
```

Important:

- Since the RTOS_Defines.pld file contents must be known by all files used in the project it is obligatory to set the "Always build all files in the project" compiler option (IDE menu: Tools, Options, Output settings).
- The RTOS_Defines.pld file must be entered in the Project Manager, section "Project Level Defines".

Some OS_... functions have another signature when certain defines are not there:

- When OS_PRIORITIES is undefined:
  The signature of OS_CreateTask becomes:
  SomeTask := OS_CreateTask(@TaskProcedure); // no priority parameter present

The OS_Priority and OS_SetPriority functions will not exist.

- When OS_DELAY is undefined:
  The OS_DELAY and the OS_StartTask_Delay functions will not exist.

- When OS_TIMEOUT is undefined:
  The signature of OS_WaitSemaphore becomes:
  OS_WaitSemaphore(SomeSem); // no timeout parameter

  The OS_TIMEOUT function will not exist.

- When OS_Signal_ISR is undefined:
  The OS_SignalSemaphore_ISR function will not exist.

The variable OS_IdleProcedure (the hook for the Idle procedure, see section 9) only exists when OS_IDLE is defined.
The function OS_Stack_Usage is only defined when OS_STACK_CHECK is defined.

10.2 RTOS list sizes

The size of some lists is defined by setting the following constants to be found in the file "RTOS_Sizes.inc":

```
const OS_TASKS_COUNT    = 10; // maximum number of tasks
const OS_PRIORITY_COUNT = 3;  // maximum number of priorities
const OS_EVENTS_COUNT   = 5;  // maximum number of semaphores
const OS_TASK_STACKSIZE = 100; // stack size in bytes allocated to each task and the Idle routine (if the latter exists)
```

Above description is self explanatory. The values are "maxima", meaning you can actually use (create) less in your application.

The constant **OS_PRIORITY_COUNT** is only needed when the compiler directive **OS_PRIORITIES** has been defined.
The constant **OS_EVENTS_COUNT** is only needed when the compiler directive **OS_SEMAPHORES** has been defined.

Important:

- The minimum value for all three constants is 1 (unless not needed of course). The maximum value of OS_TASKS_COUNT and OS_EVENTS_COUNT is 254, the maximum value of OS_PRIORITY_COUNT is 255.
- Memory for lists is always reserved according the values of the 3 OS_..._COUNT constants, irrespective of how many tasks or events are actually created, or how many priorities are actually used.
- OS_TASK_STACKSIZE: **Make sure the number is a multiple of 4!** Increase the size of the task stacks if the routine OS_Stack_Usage shows a too high percentage of usage.
- The file RTOS_Sizes.inc must be entered in the Project Manager, section "Other Files".

11 Appendixes

11.1 Programmers Reference

In this section all RTOS interface items (except configuration, see section 10) are discussed.

11.1.1 Types

TOS_Task
Represents a handle to a task.

Example:
`Var SomeTask : TOS_Task;`
`SomeTask := OS_CreateTask(@Task1, 3);`

TOS_BinarySemaphore
Represents a handle to a binary semaphore. This type only exists if the OS_SEMAPHORES configuration directive is defined, see section 10.1.

Example:
`Var SomeBinSem: TOS_BinarySemaphore;`
`SomeBinSem := OS_CreateBinarySemaphore(false);`

TOS_CountingSemaphore
Represents a handle to a counting semaphore. This type only exists if the OS_SEMAPHORES configuration directive is defined, see section 10.1.

Example:
`Var SomeCountSem : TOS_CountingSemaphore;`
`SomeCountSem := OS_CreateCountingSemaphore(0);`

TOS_Semaphore
Represents a handle to both binary and counting semaphores. This type only exists if the OS_SEMAPHORES configuration directive is defined, see section 10.1.
Example: `OS_SignalSemaphore(SomeCountSem);`

TOS_Priority
A value to be used as task priority. Value 0 (zero) is the highest priority, 1... the lower ones. This type only exists if the OS_PRIORITIES configuration directive is defined, see section 10.1.
Example: `SomeTask := OS_CreateTask(@Task1, 3); // priority 3`

TOS_State
A value type representing the state a task is in. See the Constants in section 11.1.3 for the possible values.

Example:
`Var State: TOS_State;`
`State := OS_State(SomeTask);`

11.1.2 Variables

var OS_IdleProcedure : ^TOS_IdleProcedure;
This is the “hook” to the Idle procedure, see section 9. To activate an Idle procedure set the variable to the

address of that procedure, to deactivate an idle procedure, set the variable to nil.

Example:

```
OS_IdleProcedure := @MyIdleProcedure; // activate the idle procedure
OS_IdleProcedure := nil;              // deactivate the idle procedure
```

11.1.3 Constants

OS_TASKS_COUNT

Maximum number of tasks. Should always be 1..255. To be placed in project file "RTOS_Sizes.inc", see section 10.2.

Example:

```
OS_TASKS_COUNT = 10; // maximum 10 tasks are to be created.
```

OS_PRIORITY_COUNT

Maximum number of priorities. Should always be 1..255 (if used). To be placed in project file "RTOS_Sizes.inc", see section 10.2.
Only exists if the OS_PRIORITIES configuration directive is defined, see section 10.1.

Example:

```
OS_PRIORITY_COUNT= 5; // priorities are numbered from 0..4
```

OS_EVENTS_COUNT

Maximum number of semaphores (both binary and counting together). Should always be 1..255 (if used). To be placed in project file "RTOS_Sizes.inc", see section 10.2.
Only exists if the OS_SEMAPHORES configuration directive is defined, see section 10.1.

Example:

```
OS_EVENTS_COUNT = 3; // max 3 events are to be created
```

OS_TASK_STACKSIZE

Size (in bytes) of the stack that will be allocated to each task and the Idle procedure (if the latter exists). To be placed in project file "RTOS_Sizes.inc", see section 10.2.
Make sure the number is a multiple of 4! Increase the size of the task stacks if the routine OS_Stack_Usage shows a too high percentage of usage.

Example:

```
OS_TASK_STACKSIZE = 100; // 100 bytes stack for each task (this means 25 stack levels for the
PIC32)
```

OS_NO_TASK

Value returned by OS_CreateTasks if the OS_TASKS_COUNT has been reached previously.

Example:

```
Var Task: TOS_Task;
Task := OS_CreateTask(@routine, 3);
If Task <> OS_NO_TASK then // task creation successful
...
```

OS_NO_EVENT

Value returned by OS_Create_xxx_Semaphore if the OS_EVENTS_COUNT has been reached previously.
Only exists if the OS_SEMAPHORES configuration directive is defined, see section 10.1.

Example:

```
Var Sem: TOS_Semaphore;
Sem := OS_CreateBinarySemaphore(false);
If Sem <> OS_NO_EVENT then // event creation was successful
...
```

OS_NO_TIMEOUT

Value to be used in OS_WaitSemaphore if no timeout is wanted (= infinite timeout). Only exists if the OS_TIMEOUT configuration directive is defined, see section 10.1.
Example: `OS_WaitSemaphore(SomeCountSem, OS_NO_TIMEOUT);`

st_DESTROYED
TOS_State type value: the task is not created.

st_STOPPED
TOS_State type value: the task is created, but not started (yet) or has been stopped.

st_DELAYED
TOS_State type value: the task has been started but is suspended for a period (waiting for n timebase ticks).

st_WAITING
TOS_State type value: the task has been started and is waiting a semaphore (forever or with time-out).

st_ELIGIBLE
TOS_State type value: the task has been started and is eligible to run (the wait time is over or semaphore is signaled/timeout or yielding to the Scheduler was with OS_Yield).

st_RUNNING
TOS_State type value: The task is the currently active (executing) Task.

11.1.4 Procedures and functions

procedure OS_Init;
To be called before any other RTOS routine can be called.
Example: `OS_Init;`

procedure OS_Run;
Starts the RTOS Scheduler. Should be the last routine executed in the main program, the task is blocking (never left).
Example: `OS_Run;`

function OS_CreateTask(TaskProc: ^TOS_TaskProc; Priority: TOS_Priority): TOS_Task; function OS_CreateTask(TaskProc: ^TOS_TaskProc): TOS_Task;
Adds a task to the tasks list. The "TaskProc" parameter is the start address of the task. The “Priority” parameter gives the initial priority to the task (only available when configuration directive OS_PRIORITIES is defined, see section 10.1). The function returns the number of the task in the tasks list or "OS_NO_TASK" when the tasks list is full.

Example:
`Var Task: TOS_Task;`
 	
`Procedure MyTaskProcedure;`
`Begin`
`  ...`
`End;`
 	
`Task := OS_CreateTask(@MyTaskProcedure, 2); // create task with priority 2`

procedure OS_StartTask(TaskID: TOS_Task);
Starts a (stopped) task identified by "TaskID".
Example: `OS_StartTask(Task);`

procedure OS_StartTask_Delay(TaskID: TOS_Task; Ticks: word);
Starts a (stopped) task identified by "TaskID" after an initial waiting time of "Ticks" timebase ticks. “Ticks” should have a value of 1..65535. OS_Delay can only be called from within an actual task procedure. Only available if the OS_DELAY directive is defined, see section 10.1.
Example: `OS_StartTask_Delay(Task, 30);`

procedure OS_StopTask(TaskID: TOS_Task);
Stops the task "TaskID" and returns to the Scheduler if "TaskID" is the current task. The task can only be restarted with "OS_StartTask" which will make the task resume where it was when stopped. OS_StopTask can only be called from within an actual task procedure.

Example:
`OS_StopTask(Task; // stop any task` or
`OS_StopTask(OS_CurrentTask);// stop the current task`

procedure OS_ReplaceTask(TaskID: TOS_Task);
Stops the current executing task and starts the task with "TaskID". OS_ReplaceTask can only be called from within an actual task procedure.
Example: `OS_ReplaceTask(Task2);`

function OS_CurrentTask: TOS_Task;
Returns the ID of the currently active (executing) task.
Example: `OS_SetPriority(OS_CurrentTask, 5);`

function OS_TaskRunning(TaskID: TOS_Task): boolean;
Returns true if the task "TaskID" is *started*, otherwise false.
Example: `If OS_TaskRunning(Task) then ...`

function OS_TaskState(TaskID: TOS_Task): TOS_State;
Returns the state of "TaskID".

Example:
`Var State: TOS_State;`
`State := OS_TaskState(Taskx);`

procedure OS_Yield;
Unconditionally yields to the Scheduler. If no other task is waiting to run then the current task will resume at the next instruction after "OS_Yield". OS_Yield can only be called from within an actual task procedure.
Example: `OS_Yield;`

procedure OS_Delay(Ticks: word);
Suspends the current task and returns to the Scheduler. The latter will resume the task after "Ticks" time. “Time” should have a value of 1..65535. This procedure only exists if the OS_DELAY configuration directive is defined, see section 10.1.

Example:
```
OS_Delay(100); // 100 timebase ticks delay
```

function OS_CreateBinarySemaphore(InitialValue: boolean): TOS_BinarySemaphore;

Adds a binary semaphore to the events list.
The function returns the semaphore's number or "OS_NO_EVENT" if the events list is full "InitialValue" is the initial value assigned to the semaphore (true = "signaled").
This procedure is only defined if the OS_SEMAPHORES configuration directive is defined, see section 10.1.

Example:
```
Var BinSem: TOS_BinarySemaphore;
OS_CreateBinarySemaphore(BinSem, false);
```

function OS_CreateCountingSemaphore(InitialValue: word): TOS_CountingSemaphore;

Adds a counting semaphore to the events list.
The function returns the semaphore's number or "OS_NO_EVENT" if the events list is full "InitialValue" is the initial count value assigned to the semaphore.
This procedure is only defined if the OS_SEMAPHORES configuration directive is defined, see section 10.1.

Example:
```
Var CntSem: TOS_CountingSemaphore;
CntSem := OS_CreateCountingSemaphore(CntSem, 0)
```

procedure OS_SignalSemaphore(Event_: TOS_Semaphore);

Sets the semaphore to "signaled" (binary semaphore) or increments the semaphore count by one (counting semaphore).
This procedure is only defined if the OS_SEMAPHORES configuration directive is defined, see section 10.1.

Example:
```
OS_SignalSemaphore(BinSem); // or
OS_SignalSemaphore(CntSem);
```

procedure OS_SignalSemaphore_ISR(Event_: TOS_Semaphore);

To be used from within interrupt service routines.
Sets the semaphore to "signaled" (binary semaphore) or increments the semaphore count by one (counting semaphore).
This procedure is only defined if the OS_SEMAPHORES and OS_SIGNAL_ISR configuration directives both are defined, see section 10.1.

Example:
```
OS_SignalSemaphore_ISR(BinSem); // or
OS_SignalSemaphore_ISR(CntSem);
```

procedure OS_WaitSemaphore(Event_: TOS_Semaphore; TimeOut: word);
procedure OS_WaitSemaphore(Event_: TOS_Semaphore);

Suspends the current task until the binary or counting semaphore referenced in "Event_" has been signaled (binary semaphore) or the semaphore count has a value > 0 (counting semaphore) or the TimeOut (ticks) has been elapsed.
If the event is already signaled (binary) or the semaphore count has a value > 0 (counting) when the wait routine is called then the current task will continue (no task switching).
“TimeOut” should have a value of 1..65535.
This procedure is only defined if the OS_SEMAPHORES configuration directive is defined, see section 10.1.
The TimeOut parameter is only there if the OS_TIMEOUT configuration directive is defined, see section 10.1.
OS_WaitSemaphore can only be called from within an actual task procedure.

Example:
```
OS_WaitSemaphore(BinSem, 50); // timeout of 50 timebase ticks
```

function OS_ReadSemaphore(Event_: TOS_Semaphore): boolean;

Returns true if the binary semaphore signaled or the counting semaphore's value is > 0, otherwise false. The

value of the semaphore is not changed (signaling or count value).

Example:

```
If OS_ReadSemaphore(BinSem) then ...  //or
If OS_ReadSemaphore(CntSem) then ...
```

function OS_ReadCountingSemaphore(Event_: TOS_Semaphore): word;

Returns the counting semaphore's value. That value remains unchanged.

Example:

```
Var CntSemValue: word;
CntSemValue := OS_ReadCountingSemaphore(CntSem);
```

function OS_TrySemaphore(Event_: TOS_Semaphore): boolean;

Returns true if the binary semaphore is signaled or the counting semaphore's value is > 0, otherwise false.
The value of the semaphore remains unchanged.

Example:

```
If OS_TrySemaphore(BinSem) then ... // or
If OS_TrySemaphore(CntSem) then ...
```

function OS_TimeOut: boolean;

Returns "true" if a timeout occurred, or false if no timeout occurred (e.g. after "OS_WaitSemaphore").
After calling "OS_TimeOut" the timeout flag is cleared. The flag is also initially cleared when an "OS_Wait..." routine is called.
This procedure is only defined if the OS_TIMEOUT configuration directive is defined, see section 10.1.

Example:

```
OS_WaitSemaphore(CntSem, 100);
If OS_TimeOut
then ...
else ...
```

procedure OS_ClearSemaphore(Event_: TOS_Semaphore);

Sets a binary semaphore to "not signaled" or sets a counting semaphore's count to zero.
Also pending signaling from interrupts are cleared.

Example:

```
If OS_TrySemaphore(SomeSem) then
Begin
  // do something
  OS_ClearSemaphore(SomeSem); //and clear the semaphore
End;
```

procedure OS_SetPriority(TaskID: TOS_Task; Priority: TOS_Priority);

Sets the priority of "TaskID" to "Priority".
This procedure is only defined if the OS_PRIORITIES configuration directive is defined, see section 10.1.

Example:

```
OS_SetPriority(SomeTask, 5);
```

function OS_Priority(TaskID: TOS_Task): TOS_Priority;

Returns the priority of "TaskID".
This procedure is only defined if the OS_PRIORITIES configuration directive is defined, see section 10.1.

Example:

```
Var Priority : TOS_Priority;
Priority := OS_Priority(SomeTask);
```

function OS_Stack_Usage(TaskID: TOS_Task): byte;

Returns the task usage of "TaskID" in percent.
This procedure is only defined if the OS_CHECK_TASK configuration directive is defined, see section 10.1.

Example:

```
bytetostr(OS_Stack_Usage(T1), Str); // T1 is a task (type TOS_Task)
Uart2_write_text('T1-Stack Usage: ');
Uart2_write_text(Str + '%'+ #13 + #10);
```

procedure OS_EnterCriticalSection ;
Will prevent task switching (until a call to OS_LeaveCriticalSection). This is done by preventing Timer1 interrupts (the timebase tick is stopped), see section 7.10.

Example:
`OS_EnterCriticalSection;`
`...`
`OS_LeaveCriticalSection;`

procedure OS_LeaveCriticalSection ;
will allow task switching again after a call to OS_EnterCriticalSection, see section 7.10.

Example:
`OS_EnterCriticalSection;`
`...`
`OS_LeaveCriticalSection;`

11.1.5 Directives

These are placed in the project file "`RTOS_Defines.pld`", see section 10.1.

OS_SEMAPHORES
Allows semaphores

OS_PRIORITIES
If there is more than one task priority needed

OS_DELAY
needed for OS_DELAY

OS_TIMEOUT
needed for OS_WaitSemaphore with timeout

OS_SIGNAL_ISR
To enable OS_SignalSemaphore_ISR

OS_IDLE
Enables the "Idle" procedure possibility

OS_STACK_CHECK
Enables the "OS_Stack_Usage" procedure

OS_NO_AUTO
Disables the automatic task switching every 10 timebase ticks

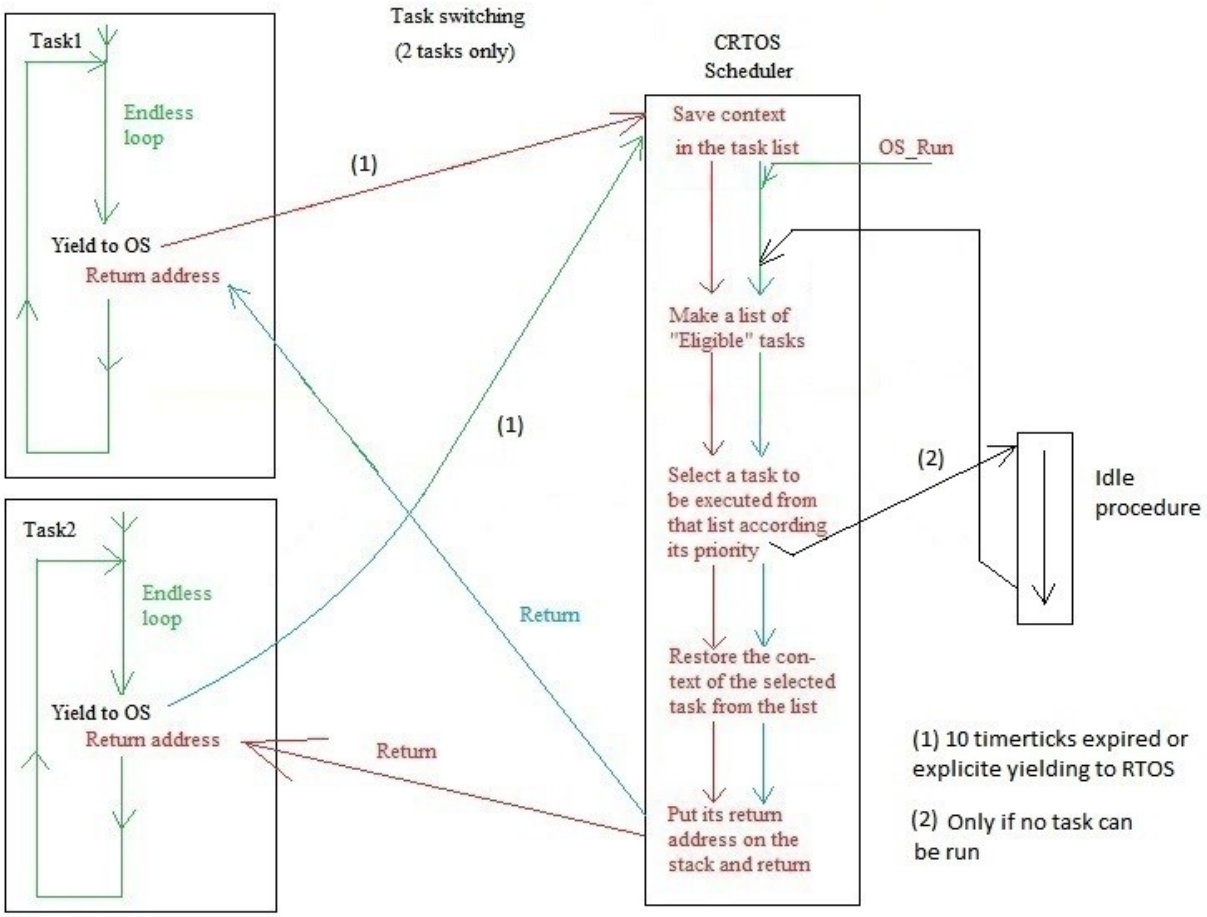
11.2 Nomenclature

Expression	Meaning
RTOS Scheduler	The main task of the scheduler is selecting a task for execution after another task yields to it. It does this by:

	• saving the context of the yielding task • calling the “Idle” procedure (optionally) • making a list of tasks eligible to be executed • Choosing one of them according the priority rules • restoring the context of the task chosen, and • finally executing the newly chosen task
RTOS Timebase	RTOS has to keep track of time when OS_DELAY or OS_WaitSemaphore timeouts are used. Additionally the automatic task switching mechanism (every 10 timer ticks) must be driven (if not disabled with OS_NO_AUTO).
RTOS Idle Procedure	A procedure called id no tasks are eligible to run, see section 9

11.3 Task (context) switching mechanism

Graphically (switching alternating between 2 tasks is shown):



Task switching is done as follows:

- The current (executed) task yields (i.e. gives control) to the RTOS Scheduler, or 10 timer ticks have elapsed
- The Scheduler saves the context of the task above,
- The Scheduler selects another task, and
- If a new task is selected the Scheduler first restores the context of that new task and gives control to it (= continue with the next statement after its yield to the OS).
  if no task is eligible to run then the “Idle” procedure is called (see section 9).
- The initial return address of a tasks is its start address. This means that code above the endless loop in a task will be executed only the very first time the task runs.

As can be seen, the scheduler does not “call” the tasks, the tasks “call” the scheduler. The scheduler simply chooses which task it “returns” to.

11.3.1 Task Context

The task context for the **PIC32** used with **mP PRO** is the following:

- Registers 2..31 (including Stack Pointer), R1 is the global pointer, which never changes
- the coprocessor's "Status"
- the coprocessor's "SRSCtl"
- The HI and LO registers
- The return address of where the Timebase interrupt occurred.

IMPORTANT:
In case you would like to make changes yourself in the RTOS library: Make sure no routine calls are done in the timer1 interrupt procedure.
One can check this to be true in the list file: the first machine statement of this routine should not be "ADDIU SP, SP, -xx".
If routines are called then place their code inline in the interrupt procedure.

11.4 Examples

11.4.1 Timers

Timers are tasks that are executed on a regular time interval, e.g. 100 milliseconds.

In RTOS there are no special "timers", so, they have to be implemented as regular tasks.
An example:

```
procedure Timer_1;
begin
  while true do
  begin
    // do here the actions that are to be executed regularly (payload)
    OS_DELAY(100);             // cyclic delay
  end;
end;
...
Task1 := OS_CreateTask(@Timer_1, 0);
...
OS_StartTask_Delay(Task1, 50); // initial delay
```

As you can see in above example there is also an "initial" delay of 50 timebase ticks. The other one makes sure the payload "do here the actions..." will be executed every 100 timebase ticks.
Of course, in stead of the constant values 50 and 100, also variables (byte or word) can be used.

There are also “one shot” timers: their payload is executed only once after a waiting time.
In code:

```
procedure OneShot; // One-shot with extra initial delay
begin
  while true do
  begin
    OS_DELAY(100);
    // do something useful here (payload)
    OS_StopTask(OS_CurrentTask); // and stop this task
  end;
end;
```

or

```
procedure OneShot;
begin
  while true do
  begin
    // do something useful here (payload)
    OS_StopTask(OS_CurrentTask); // and stop this task
  end;
end;
...
Task1 := OS_CreateTask(@OneShot, 0);
...
OS_StartTask_Delay(Task1, 50); // one shot delay
```

11.4.2 Critical resources

Critical resources are e.g. SD/MMC cards. Two tasks cannot gain access to such a resource at the same time, one task needs to release the resource before another can gain access to it. This is no big problem in a cooperative RTOS provided the actions on the critical resources are always properly completed before yielding.

Nevertheless, there is a possibility, using semaphores to provide protection for critical resources.
This is done by giving the binary semaphore used for the protection an initial value of “true” (signaled), meaning “the critical resource is available”.

The task that wants to access the resource first waits for the semaphore to be signaled (resource is available), which will make the semaphore non signaled (meaning the resource is now not available any more). After the task has done its activities with the critical resource if will make it “available” again (or “free” it) by signaling the semaphore. In code:

```
procedure MyTask;
begin
  while true do
  begin
    ...
    // here the tasks wants to use some critical resource, so
    OS_WaitSemaphore(Res, OS_NO_TIMEOUT);
    ...
    // here the critical resource can be used by this task, and, since the
    // semaphore is now unsignaled, it can not be used by other tasks
    OS_DELAY(10); // for the example, to make things difficult (*)
    ...
    // here the task is done with the critical resource, so
```

```
    OS_SignalSemaphore(Res); // release the critical resource
  end;
end;
```

(*): without the “protection” some other task could try to access the critical resource.

The semaphore is initially set to “signaled” while it is created:

```
var Res: TOS_BinarySemaphore;
Res := OS_CreateBinarySemaphore(true);
```

[end of document]