



Bâtiment EARHART
ZAC Grenoble Air Parc
38590 St Etienne de St Geoirs -
France
www.pragmatec.net

Real Time Kernel for PIC18
PICos18 v 2.xx

Pragmatec

Products and services dedicated to real time embedded systems

PICos18

Real Time Kernel for PIC18

Kernel Interface API
Version 2.xx

(engl.) 17th May 2006 – Rev 1.03

Translated by Bruce Elliott (bruce.t.elliott@t-online.de)



Bâtiment EARHART
ZAC Grenoble Air Parc
38590 St Etienne de St Geoirs -
France
www.pragmatec.net

Real Time Kernel for PIC18
PICos18 v 2.xx



TO ANY PICos18 USER

Distribution:

PICos18 is free software; you can redistribute it and/or modify it under the terms of the GNU General License as published by the Free Software Foundation; either version 2, or (at your option) any later version.

PICos18 is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with PICOS18; see the file COPYING.txt. If not, write to the Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

PICos18 is a product of the PRAGMATEC S.A.R.L. company distributed freely under the GPL licence. This licence is the warranty PICos18 will be always available for free.

PICos18 is a real time kernel based on the OSEK/VDX™ standard for the automotive industry and is dedicated to the PIC18 family of Microchip Technology Inc.

This document is the property of PRAGMATEC S.A.R.L. It is an API document to help people to use the PICos18 operating system. For technical and practical reasons it has been realized for MPLAB® and compiled with the C18 compiler for the PIC18F452 target.

TABLE OF CONTENT

1. PREAMBLE	6
A MULTI-TASK KERNEL.....	6
PICOS18 : A REAL TIME KERNEL FOR PIC18.....	7
THE OSEK/VDX™ STANDARD.....	7
THE PICOS18 FEATURES.....	8
THE GPL LICENCE.....	9
THE MICROCHIP COMPILER TOOLSUITE.....	10
THE PRAGMATEC COMPANY.....	11
2. PICOS18 API: INTRODUCTION	12
TASK MANAGEMENT.....	12
EVENT MANAGEMENT.....	12
ALARM MANAGEMENT.....	12
KERNEL SERVICES.....	12
INTERRUPT MANAGEMENT.....	12
RESOURCE MANAGEMENT.....	13
HOOK ROUTINES.....	13
3. TASK MANAGEMENT	14
DECLARETASK.....	15
ACTIVATETASK.....	16
TERMINATETASK.....	17
CHAINTASK.....	18
SCHEDULE.....	19
GETTASKID.....	20
GETTASKSTATE.....	21
4. EVENT MANAGEMENT	22
DECLAREEVENT.....	23
SETEVENT.....	24
CLEAREVENT.....	25
GETEVENT.....	26
WAITEVENT.....	27
5. ALARM MANAGEMENT	28
DECLAREALARM.....	29
GETALARMBASE.....	30
GETALARM.....	31
SETRELAARM.....	32
SETABSALARM.....	33
CANCELALARM.....	34
6. KERNEL SERVICES	35
GETACTIVEAPPLICATIONMODE.....	36
STARTOS.....	37
SHUTDOWNOS.....	38



7. INTERRUPT MANAGEMENT	39
RESUMEALLINTERRUPTS.....	40
SUSPENDALLINTERRUPTS.....	41
ENABLEALLINTERRUPTS.....	42
DISABLEALLINTERRUPTS.....	42
RESUMEOSINTERRUPTS.....	43
SUSPENDOSINTERRUPTS.....	45
8. RESOURCE MANAGEMENT	46
DECLARERESOURCE.....	47
GETRESOURCE.....	48
RELEASERESOURCE.....	49
9. HOOK ROUTINES	50
ERRORHOOK.....	51
PRETASKHOOK.....	52
POSTTASKHOOK.....	53
STARTUPHOOK.....	54
SHUTDOWNHOOK.....	55

1. Preamble

A multi-task kernel

In the embedded world we hear so many people talking about real time kernels without knowing exactly what it is. Actually to understand easily what a multi-task real time kernel is, we need to talk about 3 different aspects:

A kernel...

A kernel is a set of functionalities, regroup under the term **SERVICES** for most of them. In the case of Linux for instance the kernel is composed of the task manager, the hardware access manager, the file system manager... The shell is a program and then is not included on the kernel itself. The malloc function that reserve memory dynamically calls a kernel service because the kernel is in charge of the resource management, and memory is a resource.

One of the most famous functionality of the kernel (without being a service) is the **SCHEDULER** in charge of the task processing in parallel.

... multi-task ...

So the kernel controls the resources and lets the applications use them safety and efficiently through a set of **SERVICES**. Because the kernel warranty the stability of the system many independent applications can be ready to run, when the kernel decide it. This **MULTI-PROGRAMMING** capability let many developers create its own program without thinking about what is developed in the other tasks of the system. Each developer has the feeling to be alone to develop.

If the kernel can do it, it's also possible to make different programs running in parallel with the feeling each task is alone (but is slower than if the task will be really alone).

When the kernel is a perfect task manager (the task scheduling is the job of the kernel) we say the kernel is a **MULTI-TASK AND PREEMPTIVE** kernel. If not, each task needs to manage itself this process scheduling by calling the kernel scheduler some time to time. We say the kernel is a **MULTI-TASK AND COOPERATIVE** kernel.

PICos18 is a multi-task and preemptive kernel.

... in real time

The multi-task kernel can just manage the task parallelism by splitting the global time into small time slices for each task present in memory. The problem is that each task doesn't need the same CPU availability and some task running very seldom needs all the CPU capability when they are running.

The tasks then have different **PRIORITIES** and need to start immediately if necessary. Rather than warranty a null time (that is not possible), the kernel has to warranty a constant **LATENCY TIME**: that is called the **DETERMINISM**.

The latency time of PICos18 is 50 us.

PICos18 : a real time kernel for PIC18

With the old PIC16 family it was not possible to create such a kernel with PICmicros. Indeed the main feature of a multi-task kernel is to make run different tasks together that means we need to control the function call stack [cf *datasheet of the PIC18F452 DS39564A, page 37, chapter 4.2 « Return Address Stack »*]. Imagine what would happen if all the tasks presents in the system share the same stack: the kernel would stop the current task to activate another one and at the next RETURN instruction the address pointer will jump in the first task where the previous task stopped!

The PIC18 can manage the hardware stack dedicated to the function calls (thanks to the PUSH and POP instructions and the free motion of the stack pointer), so it is possible now to put the stack in a proper state before jumping to the next task.

It was just needed then to define the list of kernel services and how the kernel manages the tasks and the resources. Rather than designing the kernel as a proprietary solution, the Pragmatec company decided to base the kernel on a standard: the OSEK/VDX™ standard.

The OSEK/VDX™ standard

PICos18 is based on the OSEK-VDX™ standard (www.osek-vdx.org).

OSEK-VDX™ is a wide project from the automotive industry supported by the most French and German car manufacturers. The project goal is to define a standard for the control and process of embedded architectures in modern vehicles. The actual vehicles can use up to 30 calculators (motor control, onboard computer, door controller, ABS, ESP...) that transfer data through a global network (CAN, VAN, LIN, MOST buses...).

Describe the way these calculators are working all together is:

- Define the same development platform, same development process and same validation process;
- Specify a common language for manufacturers, subcontractors et third parties;
- To use a common architecture for the development, validation and intergation sequences in a project.

The term OSEK means « Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug » (Open system and related interfaces for embedded automotive electronic ». The term VDX means « Vehicle Distributed eXecutive ».

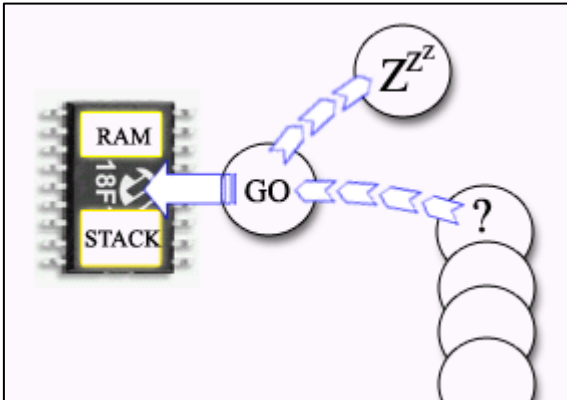
The OSEK operating system standard is included now in VDX.

Nowadays this standard is used in the automotive and the robotic industries and defines the kernel features around 3 axes: Operating System (OS), Communication (COM) and Network Management (NM). At this moment only the OS specifications has been implemented in PICos18.

The OSEK-VDX™ standard is really suitable for a PIC18 kernel. A PICos18 application is composed of many tasks symbolized by circles on the following picture.

The major feature is that only one task can have access to the PIC18 to run (more precisely to the processor, to the RAM memory and to the stack).

In order to decide which task is allowed to run at a certain time the PICos18 kernel inspects all the task of the application and chooses the task in a ready state with the highest priority.



After being activated a task can wait for a specific event and then falls asleep temporarily to let another task with a smallest priority to run. When the event will be detected the kernel will wake up the task previously put in a waiting state.

The different states of a PICos18 task are: **READY**, **SUSPENDED**, **WAITING** and **RUNNING**.

The PICos18 features

The PICos18 kernel is composed of different layers:



- ✓ The **kernel core** (Init + Scheduler + Task Manager) in charge of managing the tasks present in the application and so decide the next task to activate function to the state and the priority of the tasks.
- ✓ The **alarm and counter manager** (Alarm Manager). Attached to the kernel core it uses the TIMERO interrupt in order to update periodically the alarms and counters used by the application.
- ✓ The **Hook routines** are partially included in the kernel core and let the developer to jump into additional and personal routines. Doing so it is possible to take the control of the kernel process during a short time to debug the application for instance.
- ✓ The **task manager** (Process Manager) is a set of kernel services which proposes the necessary functions to manage the task state (to change the state of a task, chain two tasks, activate a task...).
- ✓ The **event manager** (Event Manager) is a set of kernel services which proposes the necessary function to manage the events waited or posted by/to a task (to wait for an event, to post an event, to clear an event, to read the set of events received..).
- ✓ The **interrupt manager** (INT Manager) is a set of kernel services to enable or disable the interrupts (high and low interrupts).

PICos18 is a modular kernel that means the access to the PIC18 peripherals (drivers, file system manager, etc.) are written as a task independent of the kernel.

Then it is possible for you to customize your PICos18 application by using different software layers (different tasks and libraries like those proposed by Pragmatec) to obtain something specific to your needs and also fast and easy to design.



The GPL licence

The PICos18 kernel is distributed in *open-source* under the GPL licence (*General Public Licence*). It means all of the sources of the kernel written in C code and assembling language is always available without any limitation. It means also it is totally free and you don't have to pay for any royalties to the authors.

At the top of each PICos18 source file you will find the same text with a GPL banner. If you need to modify or to use one of the PICos18 file you have to keep this banner or eventually to complete it:

```
/*
/* *****
/* File name: filename of the source file
/*
/* Since: date of creation
/*
/* Version: 2.xx (current release of PICos18)
/*
/* Author: Designed by Pragmatec S.A.R.L. www.pragmatec.net
/* MONTAGNE Xavier [XM] xavier.montagne@pragmatec.net
/* LASTNAME Firstname [xx]
/*
/* Purpose: file content explanation
/*
/* The GPL licence from Boston (USA)(cf « Free Software Foundation)
/* Distribution: This file is part of PICos18.
/* PICos18 is free software; you can redistribute it
/* and/or modify it under the terms of the GNU General
/* Public License as published by the Free Software
/* Foundation; either version 2, or (at your option)
/* any later version.
/*
/* PICos18 is distributed in the hope that it will be
/* useful, but WITHOUT ANY WARRANTY; without even the
/* implied warranty of MERCHANTABILITY or FITNESS FOR A
/* PARTICULAR PURPOSE. See the GNU General Public
/* License for more details.
/*
/* You should have received a copy of the GNU General
/* Public License along with gpsim; see the file
/* COPYING.txt. If not, write to the Free Software
/* Foundation, 59 Temple Place - Suite 330,
/* Boston, MA 02111-1307, USA.
/*
/* > A special exception to the GPL can be applied should
/* you wish to distribute a combined work that includes
/* PICos18, without being obliged to provide the source
/* code for any proprietary components.
/*
/* History:
/* 2004/09/20 [XM] Create this file.
/*
/* *****

```

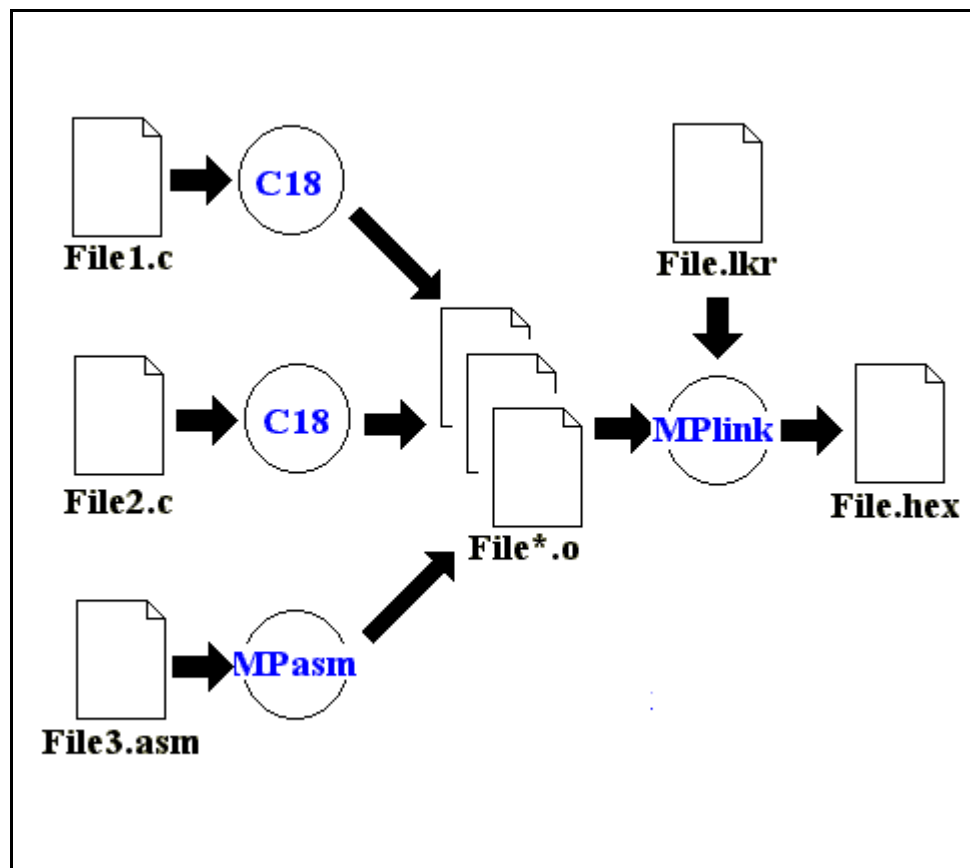
The GPL licence warranty the PICos18 sources can be get freely without any restriction or patent of any company or organisation. Moreover the PRAGMATEC company which has created PICos18 accepts to maintain the kernel and to update as often as possible always with the respect to the GPL licence. Of course anybody can modify the PICos18 sources for its own needs or can participate to the kernel evolution.

A special paragraph has been added to the GPL licence in then banner. This paragraph written in English specify it is possible to add any files to PICos18 without being forced to apply the GPL licence to these new files, it means that you don't have to provide your own sources. But keep in mind you have to provide the kernel sources itself because it is still protected by the GPL licence.

The Microchip compiler toolsuite

You're going to program your first application in C with PICos18. The kernel itself is composed of C files and a file in assembling language (kernel.asm).

These different files will be compiled and link all together in order to get a unique file: the HEX file that can be loaded into your PIC18.



The Microchip compiler toolsuite is composed of 3 elements:

- The MPASM assembler to transform the ASM files into O files
- The MCC18 compiler to transform C files into O files
- The MPLINK linker to link all the O files to a unique HEX file

The file called "linker script" (File.lkr above) is very important to generate the HEX file. Indeed the O files can not place the code and variables into memory. For instance the main() function has been translated into assembling instructions that the PIC18 can interpret but is not located at a specific place.



It is the goal of the linker script to specify the position of the variables and code into memory (into RAM and ROM). PICos18 is provided with some linker script for the most famous PIC of the PIC18 family. You can adapt them to a specific application or port one file to manage a new PIC18.

There are different other files generated after the compilation process and the link of the project (*.map, *.lst, *.cod). Have a look at the Microchip documentation for more detail about these files.

This tutorial has been written to help the users to understand and use properly the PICos18 real-time kernel. For technical reasons it has been done with MPLAB[®] and compiled with the C18 compiler in free version for a PIC18F452 target.

This tutorial has been tested with the Microsoft Windows[™] 98/NT/2K/XP and the MPLAB[®] v7.00 development environment and the C18 v2.40 compiler from Microchip.

The Microchip toolsuite can be download for free from the Microchip Technology Inc. web site: www.microchip.com.

[The Pragmatec company](#)

This tutorial is the property of PRAGMATEC S.A.R.L.

PICos18 is a product of the PRAGMATEC company and is distributed freely under the GPL licence.

The PRAGMATEC company develops and proposes PICos18 extensions to let the developer design their own application based on free software layers (RS232 driver, CAN bus driver to transfer data with the PIC18F458 CAN bus peripheral, USB, I2C, etc.).

PRAGMATEC provides also a set of additional software and hardware tools to use with PICos18 and the PIC18 to let the user debug and control the application from a Windows application, program the PIC18, convert data on a CAN bus...

2. PICOS18 API : Introduction

The objective of this document is to describe the programming interface for PICos18. Standard OSEK-VDX™ not only defines the functionalities of the kernel, but also the way in which it must be interfaced with the user's programs. PICos18 does not adhere to all of these specifications as yet, but we present the complete interface of the kernel as well as examples for using these interfaces.

Task Management

This chapter describes all of the functions which refer to handling tasks, like the activation of a task or the reading of its state or its ID. In PICos18, tasks are declared statically, i.e. they are declared with the kernel at the time of compilation and do not change dynamically. The functions referred to here as "management tasks" are for the handling of the tasks in the course of operation.

Event Management

Events in OSEK/VDX™ are equivalent to the semaphores of standard POSIX : a means of inter-process communication. Events have an essential role, because they make it possible for a task to put itself on standby and wait for an event to occur. It is also possible for a task or an interrupt to post events in order to awake tasks or to synchronize actions between several tasks.

Alarm Management

The synchronization of the tasks is an essential, but not the only element of multi-tasking systems. Specific tasks may require a timed synchronization, for example software alarms. A task using an alarm could then be awakened at fixed intervals for a fixed or infinite number of cycles. With alarms you could easily create a task which would, for example, toggle a LED every 300 ms.

Kernel Services

The kernel does not start automatically with the start of the application, according to standard OSEK/VDX™. Rather, it is the responsibility of the main() function of the application to first initialize the system. After that the kernel is started and it takes over the management of the tasks in the application.

Interrupt Management

Why use a microcontroller if you can't use its peripherals? PICos18 is conceived to enable you to use the interrupts, which can be interfaced to your tasks through the posting of events. The interrupt routines, called ISRs, can also call the kernel services and thus act as a pre-processor for the associated tasks.



Resource Management

PICos18 allows free access to all of the resources of a PIC18 for the tasks in your application. This makes it possible for you to simplify the access to the peripherals, but this can cause some problems concerning the right of access when this same peripheral is used by several tasks at the same time. This problem is solved by the resource management in the "ceiling protocol" mode of standard OSEK/VDX™. This feature is supported now by the release v2.xx of PICos18.

Hook routines

During the execution of your application it is always possible to check for correct execution thanks to the « software probes », the hook routines, linked to your tasks and working like true sensors.

3. Task Management

This chapter describes all of the functions which refer to handling tasks, like the activation of a task or the reading of its state or its ID. In PICos18, tasks are declared statically, i.e. they are declared with the kernel at the time of compilation and do not change dynamically. The functions referred to here as "management tasks" are for the handling of the tasks in the course of operation.

Types used for the management of the tasks:

TaskType	Describes the ID of a task. Value: 8 bits signed (char) ranging from 0 to 7.
TaskRefType	Reference to the ID of a task. Reference to 8 bit signed (char *) ranging from 0 to 7.
TaskStateType	The possible values of the task state : <ul style="list-style-type: none"> ✓ SUSPENDED 0x00 ✓ READY 0x20 ✓ RUNNING 0x40 ✓ WAITING 0x80 This type is not used by the API OSEK-VDX™.
TaskStateRefType	Reference to the state of a task. Reference to 8 bit signed (char *) which has the values: <ul style="list-style-type: none"> ✓ SUSPENDED 0x00 ✓ READY 0x20 ✓ RUNNING 0x40 ✓ WAITING 0x80

DeclareTask

Prototype	DeclareTask(TaskIdentifier)
Description	Used to declare the name of a task during the compilation of an application. Used by the linker during linkage editing. .
Parameters [in]	TaskIdentifier: name of the task.
Return code	None
Comments	This function of the OSEK-VDX™ API is not used by PICos18. The name of the task is not used by the Microchip MPLINK linker. For compatibility reasons, the function is implemented here as a macro : <code>#define DeclareTask(TaskIdentifier) extern TASK(TaskIdentifier)</code>
File	pro_man.h
Example	<p>The file tascdesc.c has the function of describing your application to PICos18.</p> <p>In order for the kernel to know the requirements of your tasks, you must declare them all in the file tascdesc.c, as indicated in the example.</p> <pre> ... #define DEFAULT_STACK_SIZE 128 DeclareTask(TASK0); DeclareTask(HD4478_DRV); volatile unsigned char stack0[DEFAULT_STACK_SIZE]; volatile unsigned char stack_hd4478[DEFAULT_STACK_SIZE]; ... </pre>

ActivateTask

Prototype	<code>StatusType ActivateTask (TaskType TaskID)</code>
Description	Used to change the state of a task from SUSPENDED to READY. If the task to be activated has a task ready priority it immediately takes over from the task in progress.
Parameters [in]	TaskID: ID of the task to be activated.
Return code	None if the ID corresponds to a task in the application. E_OS_ID if ID is not correct.
Comments	<p>If the ID passed as parameter corresponds to a task in the application, the ActivateTask function requests the scheduler in order to determine the next active task. The events previously posted for the task activated are not set to zero.</p> <p>The kernel guarantees the execution of the task activated from the first line of code of the task.</p> <p>You can activate a task several times before it finishes. In this case there are multiple activations of the task. When a task finishes, it is able to reactivate itself as long as its activation counter is not null (maximum value of 7).</p>
File	pro_man.c
Example	<p>Task MY_TASK awaits event KEY_PRESSED and then activates a task to manage an LCD when this event occurs.</p> <p>ActivateTask does not change the state of the active task. The scheduler is automatically called in order to determine if the priority of the new task is sufficient to allow its immediate activation.</p> <pre> TASK(MY_TASK) { unsigned char Key; while (1) { WaitEvent(KEY_PRESSED); ClearEvent(KEY_PRESSED); KeyDetection(&Key); ActivateTask(LCD_TASK_ID); /* ... */ } } </pre>

TerminateTask

Prototype	<code>StatusType TerminateTask (void)</code>
Description	Used to change the state of a task from RUNNING to SUSPENDED. The task is then no longer scheduled by the kernel. To restart it again, it is necessary to activate the task using the ActivateTask function.
Parameters [in]	None
Return code	None
Comments	<p>The TerminateTask function requests the scheduler in order to determine the next active task. A task can choose to terminate itself but cannot choose to terminate another task.</p> <p>The TerminateTask function does not return a value. Never call it as a function.</p> <p>The task finishing should never lock resources such as a peripheral or an alarm. Thus, it is not possible to create alarms within a task which calls TerminateTask. In this case it is necessary to declare and parameterize the alarm outside the task (in the function main() for example).</p> <p>If the activation counter is not null, the TerminateTask function causes the task to start again immediately (see ActivateTask).</p>
File	pro_man.c
Example	<pre> TASK(MY_TASK) { My_Task_Init(); while (1) { WaitEvent(RS232_EVENT TIMEOUT); GetEvent(my_task_ID, &my_task_event); if (my_task_event & TIMEOUT) { ClearEvent(TIMEOUT); TerminateTask(); } /* ... */ } } </pre> <p>TerminateTask makes it possible to place a task in the SUSPENDED state, in order to deactivate it.</p>

ChainTask

Prototype	<code>StatusType ChainTask (TaskType TaskID)</code>
Description	Used to change the state of the current task from RUNNING to SUSPENDED. The task whose ID is TaskID changes its state to READY. If the task thus activated has the task ready priority, it immediately takes over from the task in progress.
Parameters [in]	TaskID: ID of the chained task.
Return code	None
Comments	<p>The ChainTask function requests the scheduler in order to determine the next active task.</p> <p>A task can choose to be chained with itself, which causes it to start again. The ChainTask function does not return a value. Never call it as a function.</p> <p>The task being suspended should never lock resources such as a peripheral or an alarm. Thus, it is thus not possible to create alarms within a task which calls ChainTask. In this case, it is necessary to declare and parameterize an alarm outside the task (in the function main() for example).</p> <p>The ChainTask function suspends the running task and reduces the value of its activation counter in order to guarantee the chaining the two tasks. To preserve the concept of multiple activation, the preferable way is to use the ActivateTask(TASK_ID) function followed by TerminateTask() (see the ActivateTask function).</p>
File	pro_man.c
Example ChainTask not only makes it possible to activate a forthcoming task but also to prolong the current task. Warning: any code following the call to ChainTask will not be carried out!	<pre> TASK(MY_TASK) { TaskStateType the_task_state; My_Task_Init(); While (1) { WaitEvent(ALARM_EVENT); ClearEvent(ALARM_EVENT); GetTaskState(TARGET_TASK_ID, &the_task_state); If (the_task_state == SUSPENDED) { ChainTask(TARGET_TASK_ID); /* The current task is suspended now ...*/ } } } </pre>

Schedule

Prototype	<code>StatusType Schedule(void)</code>
Description	Call the scheduler of the kernel in order to determine the next active task. If the selected task has a priority state READY, the execution continues immediately after the call to the Schedule function.
Parameters [in]	/
Return code	None.
Comments	<p>The PICos18 kernel is a preemptive and not a cooperative kernel. Thus it is not necessary to call Schedule to force the rescheduling of the tasks, as this is a basic function of the kernel.</p> <p>The call to the Schedule function can be used to leave a critical region, i.e. leave a portion of code where no other possibilities of interruption are available. With critical regions the kernel is not able to take control on your tasks, it is thus of the responsibility for the developer to call Schedule() to force a task switching if necessary.</p>
File	pro_man.c
<p>Example</p> <p>Certain accesses to internal or external peripherals require to return in critical area, i.e. of dévalider all the interruptions. The call in Schedule makes it possible to force the regrouping of the other tasks if need be.</p> <p>Warning : Schedule enables the interrupts then it is necessary to disable them to remain in critical area (second "GIEL=0" after the Schedule() call).</p>	<pre> TASK(MY_TASK) { unsigned char table[10]; Init_Table(&table[0], 10); while (1) { /* ... */ /* Enter critical region */ INTCONbit.GIEL = 0; For (i = 0 ; i < 10 ; i++) { SendEEPROMAddr(i); WriteData(table[i]); /* GIEL = 1 when coming back from Schedule */ Schedule(); INTCONbit.GIEL = 0; } INTCONbit.GIEL = 1; /* Leave critical region */ /* ... */ } } </pre>

GetTaskID

Prototype	<code>StatusType GetTaskID (TaskRefType TaskID)</code>
Description	Return the ID of the current task.
Parameters [out]	TaskID : ID of the current task.
Return code	E_OK systematically.
Comments	<p>This function is intended to be used by Hook Routines and ISRs.</p> <p>Called since the function hand, it cannot bring any useful information and turns over value INVALID_TASK (ID except limits).</p> <p>If called from a task, this function returns the ID of that task. If called from an ISR, it returns the ID of the interrupted task. Since the function hand, it turns over INVALID_TASK.</p>
File	Pro_man.c
Example	<p>GetTaskID is used here to avoid using the constant ones representing the ID of the task (id_tsk_run). This system call is also extremely useful within an ISR to find the ID of the interrupted task.</p> <pre> TASK(MY_TASK) { TaskType my_task_ID; while (1) { WaitEvent(ALARM_EVENT AN_EVENT); GetTaskID(&my_task_ID); GetEvent(my_task_ID, &my_task_event); if (my_task_event & AN_EVENT) ClearEvent(ALARM_EVENT); /* ... */ } } </pre>

GetTaskState

Prototype	StatusType GetTaskState (TaskType TaskID, TaskStateRefType State)
Description	Return the state of the requested task.
Parameters [in] [out]	TaskID: ID of the requested task. State: state of the requested task.
Return code	E_OK if the ID of the requested task exists. E_OS_ID otherwise.
Comments	<p>GetTaskState returns the exact value of the state of the task requested at the time of the call.</p> <p>If the call is made within a task, the returned state is RUNNING if the task seeks to inquire its own state (only the task in progress is in state RUNNING).</p> <p>This function can be called from an ISR or Hook Routine. If called while in the function main() it returns the state SUSPENDED (no task is yet been activated).</p>
File	pro_man.c
Example	<pre> TASK(MY_TASK) { TaskStateType the_task_state; while (1) { WaitEvent(ALARM_EVENT); ClearEvent(ALARM_EVENT); GetTaskState(TARGET_TASK_ID, &the_task_state); if (the_task_state == SUSPENDED) { ActivateTask(TARGET_TASK_ID); } /* ... */ } } </pre> <p>It is possible to find out the current state of any task of the application using the GetTaskState function. Applied to itself a task obtains 0x6x (READY + RUNNING).</p> <p><i>SUSPENDED = 0x00</i> <i>READY = 0x20</i> <i>RUNNING = 0x40</i> <i>WAITING = 0x80</i></p>

4. Event Management

Events in OSEK/VDX™ are equivalent to the semaphores of standard POSIX: a means of inter-process communication. Events have an essential role, because they make it possible for a task to put itself on standby and wait for an event to occur. It is also possible for a task or an interrupt to post events in order to awake tasks or to synchronize actions between several tasks.

Types used for the management of the events :

EventMaskType	Indicates all events awaited or posted by a task. Value is 8 bits signed (char) ranging between 0 and 128 inclusive. This value is a power of 2 (0, 1, 2, 4, 8, 16, 32, 64 and 128).
EventMaskRefType	Reference to the events posted by a task. The Reference is 8 bits signed (char *) ranging between 0 and 128 inclusive. This value is a power of 2 (0, 1, 2, 4, 8, 16, 32, 64 and 128).

DeclareEvent

Prototype	<code>DeclareEvent (EventIdentifier)</code>
Description	Used to declare the name of an event during compilation. Used by the linker at the time of the linkage editing.
Parameters [in]	EventIdentifier: name of the event.
Return code	/
Comments	This function of the OSEK-VDX™ API is not used in PICos18. The name of the event is not treated by the linker MPLINK of Microchip. This function is not implemented in PICos18.
File	/
Example	/

SetEvent

Prototype	<code>StatusType SetEvent (TaskType TaskID, EventMaskType Mask)</code>
Description	Posts events to the task whose identifier is ID, using the events in bit mask Mask.
Parameters [in] [in]	TaskID : ID of the task concerned. Mask : mask posted event.
Return code	E_OS_STATE if the task concerned is in state SUSPENDED. E_OK if the task concerned does not await the posted event. None if the task concerned awaits the posted event.
Comments	This function is used to post an event to another task or an ISR. It should not be used in a Hook Routine. If the task targeted is in a state of WAITING for the posted event, a schedule is forced: if the task targeted has priority, it takes over from the task in progress if it is in a state of READY (case of a preemption on event). The posted event must always be a power of 2. It is thus possible to post only 8 different types of events to a task.
File	even_man.c
Example	<p>When you write a driver, function ISR in charge of the IT flag detection posts an event to the task "driver" using the SetEvent function if the flag of interruption is positioned.</p> <pre> /* CAN driver ISR */ Void CAN_INT(void) { if (PIR3bits.RX0IF) { PIR3bits.RX0IF = 0; SetEvent(CAN_DRV_ID, CAN_RX_EVENT); }; /* ... */ } </pre>

ClearEvent

Prototype	<code>StatusType ClearEvent (EventMaskType Mask)</code>
Description	Used to clear an event received by the task in progress, as defined in the bit mask of Mask.
Parameters [in]	Mask: mask clear event.
Return code	E_OK in all cases.
Comments	<p>This function is used to remove an event received by a task. It is the responsibility of the task which receives events to remove these events once received.</p> <p>If the awaked task does not remove the received event, it is likely to loop ad infinitum, awaked by the event always being present.</p> <p>It should not be used in a Hook Routine, an ISR or the main() function.</p> <p>The removed event must always be a power of 2. Thus it is thus only possible to remove 8 different types of events.</p>
File	even_man.c
Example	<p>After having received an event, the task must erase it by the call with the ClearEvent function.</p> <pre> TASK(MY_TASK) { My_Task_Init(); while (1) { WaitEvent(ALARM_EVENT AN_EVENT); GetEvent(MY_TASK_ID, &my_task_event); if (my_task_event & ALARM_EVENT) { ClearEvent(ALARM_EVENT); LATEbits.LATE2 = ~LATEbits.LATE2; } /* ... */ } } </pre>

GetEvent

Prototype	StatusType GetEvent (TaskType TaskID, EventMaskRefType Event)
Description	Used to get the events received by the task whose identifier is ID.
Parameters [in] [out]	TaskID: ID of the task concerned. Event: mask received events.
Return code	E_OS_STATE if the task concerned is in state SUSPENDED. E_OS_ID if the ID does not correspond to a task of the application. E_OK normal.
Comments	This function makes it possible to find out the events received by a task, and not just the events available. It can be used in a Hook Routine, an ISR, or the function main(). The Event value (Bit Field) is composed of several events, each one being a power of 2. When the awakened task is waiting for several events, it is necessary to filter the Event value in order to determine the event which awoke the task.
File	even_man.c
Example	<p>When a task is waiting for evants and it is awaked by one of these events, it can find out which events have been posted using the GetEvent function.</p> <pre> TASK(MY_TASK) { My_Task_Init(); while (1) { WaitEvent(ALARM_EVENT AN_EVENT); GetEvent(MY_TASK_ID, &my_task_event); if (my_task_event & ALARM_EVENT) { ClearEvent(ALARM_EVENT); LATEbits.LATE2 = ~LATEbits.LATE2; } /* ... */ } } </pre>

WaitEvent

Prototype	<code>StatusType WaitEvent (EventMaskType Mask)</code>
Description	Changes the state of the task from RUNNING to WAITING. The Mask value indicates the OR-ed sum of the events awaited by the task.
Parameters [in]	Mask: bit mask of awaited events.
Return code	E_OK if none the awaited events were already posted. E_OS_ID if the ID does not correspond to a task of the application. None if one of the awaited event is present.
Comments	<p>If one of the awaited events beforehand were already posted then a schedule is forced: if the task in progress has the highest priority it continue the execution. If not the task with the highest priority takes the control.</p> <p>This function cannot be called since Hook Routine, a ISR or the function main().</p> <p>The Mask value is made up of several events, each one being a power of 2. To suspend the current task for several events at the same time, it is necessary to compose the Mask value using an OR-ed combination of events.</p> <p>Any one of the awaited events is sufficient to awaken the task from suspension.</p>
File	even_man.c
Example	<p>In order to place a task in a waiting state it is recommended to use the system call WaitEvent inside a while loop.</p> <pre> TASK(MY_TASK) { My_Task_Init(); while (1) { WaitEvent(ALARM_EVENT AN_EVENT); GetEvent(MY_TASK_ID, &my_task_event); if (my_task_event & ALARM_EVENT) { ClearEvent(ALARM_EVENT); LATEbits.LATE2 = ~LATEbits.LATE2; } /* ... */ } } </pre>

5. Alarm Management

The synchronization of the tasks is an essential, but not the only element of multi-tasking systems. Specific tasks may require a timed synchronization, for example software alarms. A task using an alarm could then be awakened at fixed intervals for a fixed or infinite number of cycles. With alarms you could easily create a task which would, for example, toggle a LED every 300 ms.

Types used for the management of alarms :

TickType	Unit of time of counting of alarm. Value 16 bits signed (int *) ranging between 0 and 65535 included. Equal to 1ms on a PIC18 at 40 MHz (10 MHz with PLLx4).
TickRefType	Used to get a value of ticks from an alarm. Reference on a value 16 bits signed (int *) ranging between 0 and 65535 inclusive.
AlarmBaseType	Internal Parameters of an alarm. This type is not used in PICos18.
AlarmBaseRefType	Reference to the internal parameters of an alarm. This type is not used in PICos18.
AlarmType	Identifier of alarm: contains ID of alarm and the ID of the task which created this alarm. Value 8 bits signed (task) ranging between 1 and 255 inclusive.
AlarmObject	Alarm object which has a counter with the internal TickType type. Once the counter reaches a set point, the alarm starts either by posting an event or by activating a task. An alarm is not necessarily based on a Tick of 1ms. It can be managed as a simple counter which can be incremented or decremented in a task.
AlarmRefObject	Reference to an alarm.

In order to get the tick in PICos18 to correspond to 1ms, it is necessary to specify the frequency at which you will run your PIC18.

To do that, open the file main.c and modify the Init() function to specify the value of the internal frequency of the PIC18 :

```
Tmr0.It = _32MHZ;
```

You will find the different values possible in the file device.h. Never use PICos18 with an internal frequency less than 8MHz (quartz 4MHZ without PLL, for example).

DeclareAlarm

Prototype	<code>StatusType DeclareAlarm (AlarmIdentifier)</code>
Prototype	<code>DeclareAlarm (EventIdentifier)</code>
Description	Used to declare the name of an alarm during compilation. Used by the linker during linkage.
Parameters [in]	AlarmIdentifier : name of the event.
Return code	/
Comments	This function of the API OSEK-VDX™ is not used in PICos18. The name of the event is not used by the Microchip linker MPLINK. This function is not implemented in PICos18.
File	/

GetAlarmBase

Prototype	<code>StatusType GetAlarmBase (AlarmType AlarmID, AlarmBaseRefType Info)</code>
Description	Returns the ID of the task in progress. If called in a task, this function returns the ID of the task. If called in function main(), it returns INVALID_TASK.
Parameters [in] [out]	AlarmID : ID of the Alarm. Index of Alarm_list table in the file tascdesc.c Info : Reference on the structure of information.
Return code	E_OS_ID if the ID of alarm does not exist. E_OK otherwise.
Comments	This function informs about the internal parameters of alarm. Does the structure of the internal parameters of an alarm exist in PICos18 for reasons of compatibilities with standard OSEK-VDX™ but is not used by the kernel.
File	alarm.c
Example	/

GetAlarm

Prototype	StatusType GetAlarm (AlarmType AlarmID, TickRefType Tick)
Description	Returns the number of ticks remaining before the alarm goes off.
Parameters [in] [out]	TaskID : ID of requested alarm. Tick : ticks remaining before the alarm goes off.
Return code	E_OS_NOFUNC if alarm is not in the course of operation. E_OS_ID if AlarmID does not correspond to any alarm. E_OK otherwise.
Comments	This function makes it possible, for example, to cancel an alarm by a call to CancelAlarm, if that proves to be necessary.
File	alarm.c
Example	<pre> AlarmObject Alarm_list[] = { OFF, /* State */ 0, /* AlarmValue */ 0, /* Cycle */ &Counter_kernel, /* ptrCounter */ MY_TASK, /* TaskID2Activate */ ALARM_EVENT, /* EventToPost */ 0 /* CallBack */ }; ... #define ALARM_TSK0 0 TASK(MY_TASK) { SetRelAlarm(ALARM_TSK0, 1000, 0); while (1) { WaitEvent(ALARM_EVENT); ClearEvent(ALARM_EVENT); LATEbits.LATE2 = ~LATEbits.LATE2; /* ... */ GetAlarm(ALARM_TSK0, &Tick); if (Tick > 400) { CancelAlarm(GetAlarmID(TheTimer)); SetRelAlarm(GetAlarmID(TheTimer), 500, 0); } /* ... */ } } </pre>

When an alarm starts, it does not stop therefore: it continues to run during the application progress.

Sometimes it can be interesting to know the progress report of an alarm and to thus detect if it is likely to start quickly. In this case it is possible to invalidate the alarm to reprogram it.

SetRelAlarm

Prototype	<code>StatusType SetRelAlarm (AlarmType AlarmID, TickType increment, TickType cycle)</code>						
Description	Program an alarm by specifying time of release. The programming is relative, in that it is done compared to the current value of the number of ticks of the alarm.						
Parameters	<table border="0"> <tr> <td style="vertical-align: top;">[in]</td> <td>AlarmID : ID of programmed alarm. Index of Alarm_list table in the file tascdesc.c.</td> </tr> <tr> <td style="vertical-align: top;">[in]</td> <td>Increment : offset of the alarm (first time the alarm occurs).</td> </tr> <tr> <td style="vertical-align: top;">[in]</td> <td>Cycle : period of the alarm (next time the alarm occurs).</td> </tr> </table>	[in]	AlarmID : ID of programmed alarm. Index of Alarm_list table in the file tascdesc.c.	[in]	Increment : offset of the alarm (first time the alarm occurs).	[in]	Cycle : period of the alarm (next time the alarm occurs).
[in]	AlarmID : ID of programmed alarm. Index of Alarm_list table in the file tascdesc.c.						
[in]	Increment : offset of the alarm (first time the alarm occurs).						
[in]	Cycle : period of the alarm (next time the alarm occurs).						
Return code	E_OS_STATE if alarm is in the course of operation. E_OS_ID if AlarmID does not correspond to any alarm. E_OK otherwise.						
Comments	<p>This function makes it possible to awake a cyclic task of way, to light a LED several times for example.</p> <p>If you wish the alarm to be activated just few time, use a local variable to count the number of time the alarm has been activated and use also the CancelAlarm to stop it.</p> <p>Alarm thus programmed should not be in the course of operation. In the contrary case call the CancelAlarm function before reprogramming it.</p>						
File	alarm.c						
Example	<p>AlarmObject Alarm_list[] = { ... }; ...</p> <pre>#define ALARM_TSK0 0 TASK(MY_TASK) { SetRelAlarm(ALARM_TSK0, 10, 500); while (1) { WaitEvent(ALARM_EVENT); ClearEvent(ALARM_EVENT); LATEbits.LATE2 = ~LATEbits.LATE2; /* ... */ } }</pre> <p>SetRelAlarm makes it possible to program the times of release of an alarm.</p> <p>In this example an alarm is programmed to start after 10ms and been activated every 500 ms.</p> <p>The programming relates to the current value of the alarm.</p>						

SetAbsAlarm

Prototype	<code>StatusType SetAbsAlarm (AlarmType AlarmID, TickType start, TickType cycle)</code>						
Description	Program an alarm by specifying time of release. The programming is absolute, which means that it is done compared to the value 0, the original value of the alarm at the time of its creation.						
Parameters	<table border="0"> <tr> <td style="vertical-align: top;">[in]</td> <td>AlarmID : ID of programmed alarm. Index of Alarm_list table in the file tascdesc.c.</td> </tr> <tr> <td style="vertical-align: top;">[in]</td> <td>Increment : offset of the alarm (first time the alarm occurs).</td> </tr> <tr> <td style="vertical-align: top;">[in]</td> <td>Cycle : period of the alarm (next time the alarm occurs).</td> </tr> </table>	[in]	AlarmID : ID of programmed alarm. Index of Alarm_list table in the file tascdesc.c.	[in]	Increment : offset of the alarm (first time the alarm occurs).	[in]	Cycle : period of the alarm (next time the alarm occurs).
[in]	AlarmID : ID of programmed alarm. Index of Alarm_list table in the file tascdesc.c.						
[in]	Increment : offset of the alarm (first time the alarm occurs).						
[in]	Cycle : period of the alarm (next time the alarm occurs).						
Return code	E_OS_STATE if alarm is in the course of operation. E_OS_ID if AlarmID does not correspond to any alarm. E_OK otherwise						
Comments	<p>This function makes it possible to awake a task absolutely, in a very precise way compared to time of the system. After release alarm continues to increment its number of ticks (from 0 to 65535). To use one smaller period, prefer the SetRelAlarm function.</p> <p>If you wish the alarm to be activated just few time, use a local variable to count the number of time the alarm has been activated and use also the CancelAlarm to stop it.</p> <p>Alarm thus programmed should not be in the course of operation. In the contrary case call the CancelAlarm function before reprogramming it.</p>						
File	alarm.c						
Example	<pre>AlarmObject Alarm_list[] = { ... }; ... #define ALARM_TSK0 0 TASK(MY_TASK) { SetAbsAlarm(ALARM_TSK0, 60000, 0); while (1) { WaitEvent(ALARM_EVENT); ClearEvent(ALARM_EVENT); LATEbits.LATE2 = ~LATEbits.LATE2; /* ... */ } }</pre> <p>SetAbsAlarm makes it possible to program the times of releases of an alarm.</p> <p>In this example, an alarm is programmed to be activated when the system counter is 60000 ms then is stopped.</p> <p>The programming relates to the absolute value of the clock system.</p>						

CancelAlarm

Prototype	StatusType CancelAlarm (AlarmType AlarmID)
Description	Used to deactivate an alarm. This means that the alarm not only no longer continues to count, but also that there will be no alarm when the number of ticks reaches the programmed value.
Parameters [in]	AlarmID : ID of programmed alarm. Index of Alarm_list table in the file tascdesc.c.
Return code	E_OS_NOFUNC if alarm is already disabled. E_OS_ID if AlarmID does not correspond to any alarm. E_OK otherwise
Comments	This function stops the capacity of release of an alarm. It does not give to zero the value of ticks running of alarm. Must preferably be called before the use of the other functions of alarm management (SetAbsAlarm, SetRelAlarm...) in order to be sure that the alarm is not under operation during a handling.
File	alarm.c
Example	<pre> AlarmObject Alarm_list[] = { ... }; ... #define ALARM_TSK0 0 TASK(MY_TASK) { SetRelAlarm(ALARM_TSK0, 500, 500); while (1) { WaitEvent(ALARM_EVENT); ClearEvent(ALARM_EVENT); LATEbits.LATE2 = ~LATEbits.LATE2; /* ... */ GetAlarm(ALARM_TSK0, &Tick); if (Tick > 400) { CancelAlarm(ALARM_TSK0); SetRelAlarm(ALARM_TSK0, 500, 500); } /* ... */ } } </pre>

6. Kernel Services

The kernel does not start automatically with the start of the application, according to standard OSEK/VDX™. Rather, it is the responsibility of the main() function of the application to first initialize the system. After that the kernel is started and it takes over the management of the tasks in your application.

Types used for the management of the kernel :

AppModeType	Inform about the mode in which the application functions. The direction of this value is specific to the application. Value 8 bits signed (char) ranging between 0 and 255 inclusive.
--------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

GetActiveApplicationMode

Prototype	<code>AppModeType GetActiveApplicationMode (void)</code>
Description	The function returns the mode the application is running in. This value is stored in the global variable <code>appMode</code> . File: <code>pro_man.c</code> .
Parameters [in]	/
Return code	Value of <code>appMode</code> .
Comments	This function makes it possible to write an application sensitive to different operating modes, like a mode R&D, VALIDATION, DELIVERY, ... The tasks would be able to react differently, according to the value of <code>appMode</code> .
File	<code>pro_man.c</code>
Example	/

StartOS

Prototype	<code>void StartOS (AppModeType Mode)</code>
Description	Save the return address in the function main(). Call the function to initialize the PICos18 kernel by assigning the value Mode to the global variable appMode. .
Parameters [in]	Mode : operation mode of the application.
Return code	/
Comments	This function is necessary to the starting of the PICos18 kernel. The Mode value is used to preserve an indicator on the operation mode of the application. This makes it possible to write an application sensitive to different operating modes like a mode R&D, VALIDATION, DELIVERY, ... The tasks would be then able to react differently according to the value in appMode.
File	pro_man.c
Example	/

ShutdownOS

Prototype	<code>void ShutdownOS (StatusType Error)</code>
Description	Used to return to the function main() in the event of an emergency stop.
Parameters [in]	Error : error at the origin of instability.
Return code	/
Comments	<p>In the event of detection of a major error requiring a restarting of the system, a task can request ShutdownOS in order to stop the kernel and return to the function main().</p> <p>The remainder of the operations is specific to the application but in a general way, it is preferable that the function main() restarts the complete application.</p>
File	pro_man.c
Example	/



7. Interrupt Management

Why use a microcontroller if you can't use its peripherals? PICos18 is conceived to enable you to use the interrupts, which can be interfaced to your tasks through the posting of events. The interrupt routines, called ISRs, can also call the kernel services and thus act as a pre-processor for the associated tasks.

ResumeAllInterrupts

Prototype	<code>StatusType ResumeAllInterrupts (void)</code>
Description	Used to enable the general interrupts of the PIC18. This service functions with the DisableAllInterrupts service.
Parameters [in]	/
Return code	/
Comments	<p>The PIC18 has a bit for general enabling of interrupts (GIEL for Global Interrupt Enable for Low Interrupts and GIEH for High Interrupts).</p> <p>When GIE_x is set, all the interrupts of this category are enabled. When GIE_x is cleared, all the interrupts of this category are disabled.</p> <p>ResumeAllInterrupts makes it possible to set both GIEL and GIEH, so that the all interrupts are enabled (i.e. no longer masked).</p>
File	int_man.c
Example	<pre> TASK(MY_TASK) { unsigned char table[10]; Init_Table(&table[0], 10); while (1) { /* ... */ /* Enter critical region */ DisableAllInterrupts(); For (i = 0 ; i < 10 ; i++) { SendEEPROMAddr(i); WriteData(table[i]); Schedule(); DisableAllInterrupts(); } ResumeAllInterrupts(); /* Leave critical region */ /* ... */ } } </pre> <p>Certain accesses to internal or external peripherals necessitate the use of a critical region, where all interrupts are disabled. In this case one can call upon the services ResumeAllInterrupts and SuspendAllInterrupts.</p> <p>Warning: Schedule resets the interrupts, so it is necessary to disable them in order to remain in a critical area.</p>

SuspendAllInterrupts

Prototype	<code>StatusType SuspendAllInterrupts (void)</code>
Description	Used to disable the general interrupts of the PIC18. This service functions with the EnableAllInterrupts service.
Parameters [in]	/
Return code	/
Comments	<p>The PIC18 has a bit for general enabling of interrupts (GIEL for Global Interrupt Enable for Low Interrupts and GIEH for High Interrupts).</p> <p>When GIE_x is set, all the interrupts of this category are enabled. When GIE_x is cleared, all the interrupts of this category are disabled.</p> <p>SuspendAllInterrupts clears GIEL and GIEH, so that all interrupts will be disabled.</p>
File	int_man.c
Example	<pre> TASK(MY_TASK) { unsigned char table[10]; Init_Table(&table[0], 10); while (1) { /* ... */ /* Enter critical region */ SuspendAllInterrupts (); For (i = 0 ; i < 10 ; i++) { SendEEPROMAddr(i); WriteData(table[i]); Schedule(); SuspendAllInterrupts (); } EnableAllInterrupts(); /* Leave critical region */ /* ... */ } } </pre> <p>Certain accesses to internal or external peripherals necessitate the use of a critical region, where all interrupts are disabled. In this case one can call upon the services ResumeAllInterrupts and SuspendAllInterrupts.</p> <p>Warning: Schedule resets the interrupts, so it is necessary to disable them in order to remain in a critical area.</p>

EnableAllInterrupts

Prototype	<code>void EnableAllInterrupts (void)</code>
Description	Used to enable the general interrupts of the PIC18. This service functions with the DisableAllInterrupts service.
Parameters [in]	/
Return code	/
Comments	<p>The PIC18 has a bit to enable interrupts (GIEL for Global Interrupt Enable for Low Interrupts and GIEH for High Interrupts).</p> <p>When GIE_x is set, all the interrupts of this category are enabled. When GIE_x is cleared, all the interrupts of this category are disabled.</p> <p>EnableAllInterrupts makes it possible to enable GIEL and GIEH, so that all of the interrupts are enabled.</p>
File	int_man.c
Example	<pre> TASK(MY_TASK) { unsigned char table[10]; Init_Table(&table[0], 10); while (1) { /* ... */ /* Enter critical region */ DisableAllInterrupts(); For (i = 0 ; i < 10 ; i++) { SendEEPROMAddr(i); WriteData(table[i]); Schedule(); DisableAllInterrupts(); } EnableAllInterrupts(); /* Leave critical region */ /* ... */ } } </pre> <p>Certain accesses to internal or external peripherals necessitate the use of a critical region, where all interrupts are disabled. In this case one can call upon the services ResumeAllInterrupts and SuspendAllInterrupts.</p> <p>Warning: Schedule resets the interrupts, so it is necessary to disable them in order to remain in a critical area.</p>

DisableAllInterrupts

Prototype	<code>void DisableAllInterrupts (void)</code>
Description	Used to disable the general interrupts of the PIC18. This service functions with the EnableAllInterrupts service.
Parameters [in]	/
Return code	/
Comments	<p>The PIC18 has a bit to enable interrupts (GIEL for Global Interrupt Enable for Low Interrupts and GIEH for High Interrupts).</p> <p>When GIE_x is set, all the interrupts of this category are enabled. When GIE_x is cleared, all the interrupts of this category are disabled.</p> <p>DisableAllInterrupts makes it possible to disable GIEL and GIEH, so that all of the interrupts are disabled (masked).</p>
File	int_man.c
<p>Example</p> <p>Certain accesses to internal or external peripherals necessitate the use of a critical region, where all interrupts are disabled. In this case one can call upon the services ResumeAllInterrupts and SuspendAllInterrupts.</p> <p>Warning: Schedule resets the interrupts, so it is necessary to disable them in order to remain in a critical area.</p>	<pre> TASK(MY_TASK) { unsigned char table[10]; Init_Table(&table[0], 10); while (1) { /* ... */ /* Enter critical region */ DisableAllInterrupts(); For (i = 0 ; i < 10 ; i++) { SendEEPROMAddr(i); WriteData(table[i]); Schedule(); DisableAllInterrupts(); } EnableAllInterrupts(); /* Leave critical region */ /* ... */ } } </pre>

ResumeOSInterrupts

Prototype	<code>void ResumeOSInterrupts (void)</code>
Description	Used to enable the low level interrupts of the PIC18. This service functions with the SuspendOSInterrupts service.
Parameters [in]	/
Return code	/
Comments	<p>The PIC18 has a bit to enable interrupts (GIEL for Global Interrupt Enable for Low Interrupts and GIEH for High Interrupts).</p> <p>When GIE_x is set, all the interrupts of this category are enabled. When GIE_x is cleared, all the interrupts of this category are disabled.</p> <p>ResumeOSInterrupts makes it possible to enable the low level interrupts of the PIC18 after they were disabled.</p>
File	int_man.c
Example	<p>The low level interrupts cannot stop the PICos18 kernel, but they can interrupt any task.</p> <p>Sometimes one wishes a task be able to be stopped, except when one carries out delicate operations in the kernel, like the passage from one task to another. One can then call the kernel with the call "Schedule" and to be sure not to be interrupted by a high level IT (known as fast interrupt).</p> <pre> TASK(MY_TASK) { unsigned char table[10]; Init_Table(&table[0], 10); while (1) { /* ... */ For (i = 0 ; i < 10 ; i++) { SendEEPROMAddr(i); WriteData(table[i]); /* Disable fast interrupts only */ SuspendOSInterrupts(); Schedule(); ResumeOSInterrupts(); } /* Leave critical region */ /* ... */ } } </pre>

SuspendOSInterrupts

Prototype	<code>void SuspendOSInterrupts (void)</code>
Description	Used to disable the fast interrupts of the PIC18. This service functions with the ResumeOSInterrupts service.
Parameters [in]	/
Return code	/
Comments	<p>The PIC18 has a bit to enable interrupts (GIEL for Global Interrupt Enable for Low Interrupts and GIEH for High Interrupts).</p> <p>When GIE_x is set, all the interrupts of this category are enabled. When GIE_x is cleared, all the interrupts of this category are disabled.</p> <p>SuspendOSInterrupts allows disabling the high level interrupts (fast) of the PIC18.</p>
File	int_man.c
Example	<p>The low level interrupts cannot stop the PICos18 kernel, but they can interrupt any task.</p> <p>Sometimes one wishes a task be able to be stopped, except when one carries out delicate operations in the kernel, like the passage from one task to another. One can then call the kernel with the call "Schedule" and to be sure not to be interrupted by a high level IT (known as fast interrupt).</p> <pre> TASK(MY_TASK) { unsigned char table[10]; Init_Table(&table[0], 10); while (1) { /* ... */ For (i = 0 ; i < 10 ; i++) { SendEEPROMAddr(i); WriteData(table[i]); /* Disable fast interrupts only */ SuspendOSInterrupts(); Schedule(); ResumeOSInterrupts(); } /* Leave critical region */ /* ... */ } } </pre>

8. Resource Management

PICos18 allows free access to all of the resources of a PIC18 for the tasks in your application. This makes it possible for you to simplify the access to the peripherals, but this can cause some problems concerning the right of access when this same peripheral is used by several tasks at the same time. This problem is solved by the resource management in the "ceiling protocol" mode of standard OSEK/VDX™. However, this operating mode is not yet supported by PICos18. It is therefore necessary, that you try to limit the competition for material resources between the tasks in your application.

Types used for Resource Management :

ResourceType	Used to get information about the parameters of a resource : <ul style="list-style-type: none">- priority : priority of the resource concerned- Taskprio : priority of the task requesting the resource- lock : semaphore for mutual exclusion
---------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

The "ceiling protocol" algorithm is simple :

A resource has a certain priority. Suppose that two tasks try to access the internal EEPROM of the PIC18. To avoid conflicting access owing to the fact that the application is multi-tasking, we give the resource a priority that is higher than either of the two tasks.

When one of the tasks requests the resource, that task's priority is modified to take on the value of the priority of the resource. The priority of that task is thus ensured to be the higher of the two tasks. Due to the way the kernel handles the scheduling of priorities, the resource cannot be locked by the task of lesser priority.

However the task which has locked the resource could be suspended, for example by calling the WaitEvent service. To prevent the task of lower priority from hanging while waiting for a resource to become available, a "lock" field was added so that a task can find out whether a resource is locked or not before it tries to request that resource.

Consult the tasks of the tutorial delivered with the release of PICos18. You will find good examples of resource management with tasks 0 and 1.

DeclareResource

Prototype	<code>DeclareResource (ResourceIdentifier)</code>
Description	Used to declare the name of a resource used in the compilation.
Parameters [in]	ResourceIdentifier: name of the resource.
Return code	/
Comments	This function is not implemented in PICos18. Resources are instead declared statically in the file tascdesc.c
File	/
Example	/

GetResource

Prototype	StatusType GetResource (ResourceType ResID)
Description	Used to reserve a resource by CeilingProtocol (OSEK-VDX™).
Parameters [in]	ResID: ID of the resource. Index of the resource in Resource_list table of tascdesc.c.
Return code	E_OS_ACCESS: if the resource is already reserved. E_OS_ID: if the resource does not exist. E_OK: if successful.
Comments	This function makes it possible to lock a resource.
File	pro_man.c
Example	<pre>Resource Resource_list[] = { { 10, /* priority */ 0, /* Task prio */ 0, /* lock */ } }; ... #define ALARM_TSK0 0 TASK(TASK0) { unsigned char value; SetRelAlarm(ALARM_TSK0, 20, 20); while(1) { WaitEvent(ALARM_EVENT); ClearEvent(ALARM_EVENT); if (GetResource(0) == E_OK) { value++; ReleaseResource(0); } } }</pre>
<p>Task TASK0 locks the resource whose index is 0. At that point, the task assumes the priority of the resource, 10, so that the other tasks cannot interrupt.</p> <p>However another task that has already locked the resource could be suspended. It is thus necessary to check if the resource is locked or not before really requesting it.</p>	

ReleaseResource

Prototype	StatusType ReleaseResource (ResourceType ResID)
Description	Used to release a resource by CeilingProtocol (OSEK-VDX™).
Parameters [in]	ResID: ID of the resource. Index of the resource in Resource_list table of tascdesc.c.
Return code	E_OS_ACCESS: if the resource is already reserved. E_OS_ID: if the resource does not exist. E_OK: if successful.
Comments	This function makes it possible to release a resource.
File	pro_man.c
Example	<pre> Resource Resource_list[] = { { 10, /* priority */ 0, /* Task prio */ 0, /* lock */ } }; ... #define ALARM_TSK0 0 TASK(TASK0) { unsigned char value; SetRelAlarm(ALARM_TSK0, 20, 20); while(1) { WaitEvent(ALARM_EVENT); ClearEvent(ALARM_EVENT); if (GetResource(0) == E_OK) { value++; ReleaseResource(0); } } } </pre> <p>Task TASK0 locks the resource whose index is 0. At that point, the task assumes the priority of the resource, 10, so that the other tasks cannot interrupt it. However another task that has already locked the resource could be suspended. It is thus necessary to check if the resource is locked or not before really requesting it. Finally, TASK0 releases the resource when it is finished.</p>

9. Hook Routines

During the execution of your application it always is possible to check for correct execution thanks to the « software probes », the hook routines, which are true sensors, linked to your tasks.

Types used for the management of Hook routines :

OSServiceIdType	Used to inquire the last service called in the kernel. This type is not implemented in PICos18.
------------------------	----------------------------------------------------------------------------------------------------

ErrorHook

Prototype	<code>void ErrorHook (StatusType Error)</code>
Description	This Hook routine is called systematically by the kernel when a service returns an error code (value of return other than E_OK).
Parameters [in]	Error returned by the service at fault.
Return code	/
Comments	This function is not implemented in PICos18.
File	main.c
Example	/

PreTaskHook

Prototype	<code>void PreTaskHook (void)</code>
Description	This Hook routine is called just before jumping into the next active task.
Parameters [in]	/
Return code	/
Comments	<p>This function allows the execution of several instructions just before starting the next active task.</p> <p>You can thus detect the entry into a particular task or save certain parameters before they are modified by the task.</p> <p>The PreTaskHook function should be used only during the debugging run of your code. It is not intended to remain in the final version of your application.</p> <p>To activate the PreTaskHook function, compile your project with the option "PRETASKHOOK". To do that, add "- D PRETASKHOOK " to the command line options in the panel for compiler options. Consult the documentation of c18 to obtain more information on the command line options of the compiler.</p>
File	main.c
Example	<pre> /***** /* From the main.c file /***** ... /***** * Hook routine called just before entering in a task. * * @return void *****/ void PreTaskHook(void) { unsigned char task_ID; task_ID = GetTaskID(); if (task_ID == TASK_COUNTER) { LATEbits.RE1 = ~LATEbits.RE1 ; } return ; } </pre>

PostTaskHook

Prototype	<code>void PostTaskHook (void)</code>
Description	This Hook routine is called just after leaving an active task.
Parameters [in]	/
Return code	/
Comments	<p>This function allows the execution of several instructions before returning to the kernel.</p> <p>You can thus detect the exit of a particular task or save the values of certain variables at the moment you leave the task.</p> <p>The PostTaskHook function should be used only during the debugging run of your code. It is not intended to remain in the final version of your application.</p> <p>To activate the PostTaskHook function, compile your project with the option "POSTTASKHOOK". To do that, add "- D POSTTASKHOOK " to the command line options in the panel of the compiler options. Consult the documentation of c18 to obtain more information on the command line options of the compiler.</p>
File	main.c
Example	<pre> /***** /* From the main.c file /***** ... void PreTaskHook(void) { unsigned char task_ID; task_ID = GetTaskID(); if (task_ID == TASK_COUNTER) LATEbits.RE1 = 1; } /***** * Hook routine called just after leaving a task. * * @return void *****/ void PostTaskHook(void) { unsigned char task_ID; task_ID = GetTaskID(); if (task_ID == TASK_COUNTER) LATEbits.RE1 = 0; } </pre>

StartupHook

Prototype	<code>void StartupHook (void)</code>
Description	This Hook routine is called just after the initialization of the kernel has completed. The function will be activated only once during the initialization of the kernel
Parameters [in]	/
Return code	/
Comments	<p>This function allows the execution of several instructions before returning to the kernel.</p> <p>You could, for example, initialize variables of your application to a certain value.</p> <p>The StartUpHook function should be used only during the debugging run of your code. It is not intended to remain in the final version of your application.</p> <p>To activate the StartupHook function, compile your project with the option "STARTUPHOOK". To do that, add "- DSTARTUPHOOK" to the command line options in the panel of the compiler options. Consult the documentation of c18 to obtain more information on the command line options of the compiler.</p>
File	main.c
Example In this StartUpHook example a table of even numbers is initialized. It is possible to call kernel services from the StartUpHook function. However, as the tasks have not yet been activated by the core, calls to the majority of the services will not have any effect.	<pre> /***** /* From the main.c file /***** ... extern unsigned char even_tab[8]; ... /***** * Hook routine called before entering into the kernel * * @return void *****/ void StartupHook(void) { unsigned char index; for (index = 0; index < 8; index++) { even_tab[index] = index * 2; } } </pre>

ShutdownHook

Prototype	<code>void ShutdownHook (StatusType Error)</code>
Description	This Hook routine is called from the ShutDownOS function. This makes it possible to find the origin of the breakdown which caused the system to stop
Parameters [in]	Error (the same Error passed in as parameter to the ShutDownOS function)
Value of Return	/
Comments	<p>This function is called from ShutDownOS and allows the execution of several instructions before returning to the ShutDownOS function.</p> <p>The ShutDownOS function receives an error as parameter and retransmits this error to the ShutdownHook function, where it can be used to identify the origin of the breakdown.</p> <p>To activate the ShutdownHook function, compile your project with the option "SHUTDOWNHOOK". To do that, add "- DSHUTDOWNHOOK" to the command line options in the panel of the compiler options. Consult the documentation of c18 to obtain more information on the command line options of the compiler.</p>
File	main.c
Example Here the ShutdownHook function is used for recording the last error received in a table of errors, then saving the table in EEPROM. The ShutdownHook routine should be considered as the last function of the application to be executed before the system is restarted. All of the variables will be re-initialized afterwards.	<pre> /***** /* From the main.c file /***** ... extern unsigned char error_tab[10]; extern unsigned char error_index; ... /***** * Hook routine called just after leaving the kernel. * * @return void *****/ void ShutdownHook(StatusType error) { even_tab[error_index] = error; WriteInEEPROM(error_tab, error_index); } </pre>