P-RTOS

REAL TIME OPERATING SYSTEM
FOR
PROTON DEVELOPMENT SYSTEM

TABLE OF CONTENTS

INTRODUCTION

RTOS is a Real Time Operating System designed and written specifically in PDS Basic. The system uses co-operative as opposed to pre-emptive scheduling which means that the application code you write has to voluntarily release back to the operating system at appropriate times.

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.

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 time.

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

RTOS FUNDAMENTALS

This section describes the fundamentals of the RTOS citing simple examples written using the PDS RTOS syntax.

A typical program written in PDS Basic 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.

SOME BASIC DEFINITIONS

The fundamental building block of RTOS are ***Tasks***. Tasks are a discrete set of instructions that will perform a recognised 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*** 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.

A ***Context Switch*** occurs when one task is ***Suspended*** and another task is ***Started*** or ***Resumed***. This is core functionality to a RTOS. In the PDS RTOS the action of suspending is co-operative. This means that your tasks must be written in a way that it will ***Yield*** back to RTOS in a timely manner. If the task fails to Yield back the system will fail as the non-yielding task will run to the exclusion of all the others.

Tasks can call for a ***Delay*** 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 DelaymS or DelayuS functions in PDS 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 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 (***Signalled***) 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. PDS RTOS supports ***Semaphores***, ***Messages*** and ***Event Flags***. (Currently only Semaphores and Messages are implemented). 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 is "not signalled". ***Messages*** require both sender (***Signaller***) and receiver (***Waiting Task***) to have knowledge of the size and type of data to be shared. The signalled event itself only contains a pointer to the message being passed.

There are a number of other features which are part of PDS RTOS but these will be covered later. However, there is one important aspect that it is important to appreciate before we get into more detail. In a multi tasking environment such as RTOS it is quite conceivable that two tasks could make a call to the same function. This requires that the function can be used simultaneously by more than one task without corrupting its data. PDS does not naturally generate re-entrant code and you will have to write any functions which require re-entrancy with great care or protect the situation from occurring. However with PDS RTOS's co-operative scheduling or through the use of events this problem can be circumvented.

STRUCTURE OF A TASK

Typically a task is a piece of code which will perform an operation within the program repeatedly. A task in PDS RTOS would look like this:

```
UsefulTask:
      Repeat
            'Do something useful
            OS_Yield          'Context Switch
      Forever
```

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. (Note the expression Forever is a macro for "Until 1=0") . In a co-operative RTOS every task **<u>must</u>** make a call back to the operating at least once in its loop. OS_Yield is one of a number of mechanisms for relinquishing control back to the operating system.

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 import 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.

TASK STATES

A Task can assume a number of states:

Dormant	*Task not created*
Pending	*Task created but not started*
Delayed	*Task has been started but is suspended for a period*
Waiting	*Task has been started and is waiting an event*
Ready	*Task has been started and is ready or eligible to run*
Running	*Task is the current active task*

Tasks have to be registered or ***Created*** in RTOS before they can be used. Details including the state of each task are held by RTOS in Task control blocks (***TCBs***). Before a task is created the TCB state will be ***Dormant***. When a task is first created its state will be ***Pending.*** This means the task has been registered but has not yet been started. Once started the task can have 4 states; ***Delayed*** meaning it is waiting for a certain number of operating ticks, ***Waiting*** means it is waiting for an event to occur, ***Ready*** means its waiting to be run by the scheduler. When a task is finally called by the scheduler its state will be ***Running***.

REAL LIFE EXAMPLE

Let's look at a very basic example of a real program written for RTOS. This code has been written simply to test the RTOS and does nothing useful except demonstrate some of the features of RTOS.

```
Device 18F452
Optimiser_Level = 3
Xtal = 20
Bootloader = Off
All_Digital  = True
Create_Coff = On

Include "RTOS Defines.inc"

$define OSTASKS_COUNT 6                 ' Maximum Task count is 256
$define OSPRIO_COUNT 8                  ' Number of priority levels

$define OSENABLE_TIMER True             ' Enables timer service
$define OSENABLE_TIMEOUTS True          ' allow timeouts for events and counters
$define OSTICK_SOURCE T1                ' T0, T1, EXT
$define OSTIMER_PRESCALE Off            ' Pre-scale value or off
$define OSTIMER_PRELOAD $3CB0           ' Preload value    $D8E0
$define OSTICK_CTR_SIZE 2               ' Size of OS Tick Counter (bytes) (must be 1, 2 or 4 max)
$define OSENABLE_CYCLIC_TIMERS True     ' allow cyclic timers to be created

$define OSENABLE_EVENTS True            ' Enables Events
$define OSEVENTS_COUNT 2                ' Max number of events
$define OSENABLE_MESSAGES False         ' Event Messages enabled
$define OSENABLE_SEMAPHORES True        ' Event Semaphores enabled
$define OSENABLE_EVENT_FLAGS False      ' Event Flags enabled
$define OSEVENT_FLAGS 1                 ' Max Event flags supported
```

The code above sets up RTOS and defines which RTOS features are to be turned on. These features are described in the chapter on CONFIURATION.

```
GoTo Start

Include "RTOS Vars.inc"
Include "RTOS Macros.Inc"
Include "RTOS Main.bas"
```

These includes are the RTOS operating code.

```
Dim Ctr As Word

Symbol T_Count   = OSTCBP(1)
Symbol T_LEDOut = OSTCBP(2)
Symbol T_OSCOut = OSTCBP(4)
Symbol T_Delayed = OSTCBP(5)
Symbol T_BinSem = OSTCBP(6)

Symbol E_LedCtrl = OSECBP(1)
```

We are going to need to register 5 tasks and one event for this application. Each task will be assigned a unique ID which we have aliased to meaningful names derived from the tasks to which they refer. Below are the actual tasks which make up the application.

```
CountTsk:
Repeat
    Inc Ctr
    If Ctr = $1FF Then OSSignalBinSem E_LedCtrl
    if Ctr = $2FF then Ctr = 0
    OS_Yield
Forever

LEDOut:
```

```
Repeat
    PORTD = Ctr & $3F
    OS_Yield
Forever

OSCOut:
Repeat
    PORTC = Ctr & $0F
    OS_Yield
Forever

DelayedTask:
Repeat
    Toggle PORTA.5
    OSStartTask T_OSCOut
    OS_Delay 2
    Toggle PORTA.5
    OSStopTask T_OSCOut
    OS_Delay 10
    OS_Replace DelayedTask2, 3
Forever

DelayedTask2:
Repeat
    Toggle PORTA.5
    OSStartTask T_OSCOut
    OS_Delay 1
    Toggle PORTA.5
    OSStopTask T_OSCOut
    OS_Delay 20
    OS_Replace DelayedTask, 2
Forever

BinSemTask:
Repeat
    OS_WaitBinSem E_LedCtrl,OSNO_TIMEOUT
    OS_Delay 1
    OSStartTask T_LEDOut
    OS_Delay 1
    OSStopTask T_LEDOut
Forever
```

All the tasks are very basic and are there just to demonstrate some activity in each task.

CountTsk increments a counter, CTR, and when it's reaches $1FF it signals an event. It then continues counting until it reaches $2FF when it resets to 0. Every iteration of CountTsk will Yield back to RTOS.

LEDOut and **OSCOut** simply output the bottom 4 bits of the counter, CTR to PortD and PortC respectively.

DelayedTask and **DelayedTask2** both do the same thing but with different timings and are written to demonstrate the operation of the TaskReplace function. They toggle a bit on PORTA and start the OSCOut task, delay and then stop OSCOut task Delay again then swap with each other and change priorities at the same time.

BinSemTask waits on a binary semaphore. When that semaphore is signalled (by CountTsk) it delays 1 tick then starts the LEDOut Task for one OS Tick then stops it. As the CountTsk is operating completely asynchronously from the OS Tick source BinSemTask could be signalled at any time relative to the next OS Tick. By implementing a Delay before starting the LEDOut task we can be sure that the LEDOut task will run for a full OS Tick.

Now we get to the main program.

```
Start:
TRISA = %000000         ' All Port A Outputs
TRISB = %00000000       '
TRISD = %00000000       ' All port D pins output
TRISC = %11000000       ' Set port C to output
Ctr   = $0000           ' reset ctr

OSInit                  ' Initialise RTOS
OSCreateTask T_Count, CountTsk, 3
OSCreateTask T_LEDOut, LEDOut, 3
OSCreateTask T_OSCOut, OSCOut, 3
OSCreateTask T_Delayed, DelayedTask, 3
OSCreateTask T_BinSem, BinSemTask, 3
OSCreateBinSem E_LedCtrl, 0
```

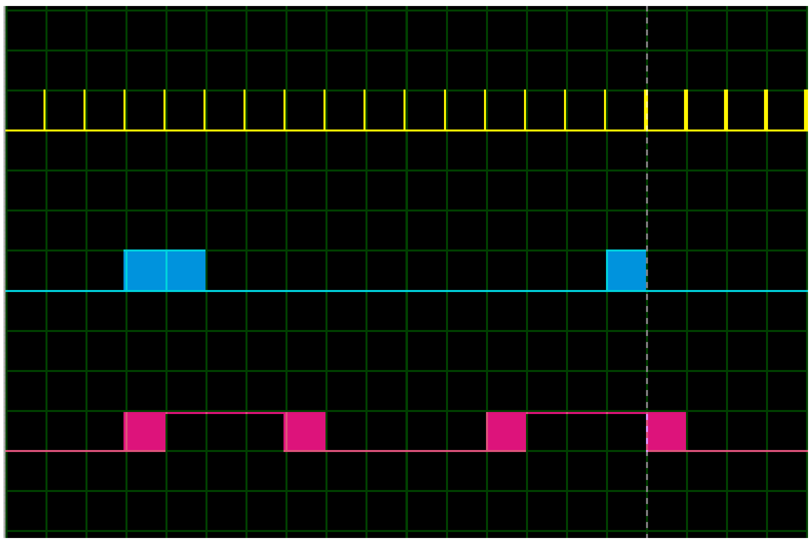
```
        OSStartTask T_LEDOut    '
        OSStartTask T_Count     '
        OSStartTask T_Delayed   '
        OSStartTask T_BinSem    '

        Repeat
            OSSched             ' run scheduler continuously
        Forever
```

After setting up the ports, we have to initialise the RTOS before we can do anything else. Next we register the tasks we want to run with RTOS by calling OSCreateTask with the TaskName and the priority at which we want it to run. (Note we haven't registered DelayedTask2 as this will occupy the same task control block as DelayedTask.) Next we register the Event we want to use with OSCreateEvent and set its initial value.

We then start each of the tasks in turn and finally and perhaps the most important bit we start and repeatedly call the scheduler.

The image below shows the program in operation.



The yellow trace shows the 10mS ticks from the operating system

The cyan trace shows OSCOut task being controlled by DelayedTask and DelayedTask2. The longer pulse is DelayedTask while the shorter is DelayedTask2

The magenta trace is the BinSem task starting and stopping the LEDOut task for 1 tick each time the counter hits $1FF.

You can see the BinSem Task running apparently at the same time as the Delayed Task showing the multitasking nature of the RTOS. To achieve this the tasks which are running concurrently must be on the same priority where they will alternate in a round robin sequence.

Note - there are periods when no tasks are running and the RTOS is idling waiting for another task to run. In a conventional polled system this idle time would not be available but with RTOS you can use this time for other tasks when required.

IMPORTANT:

When a task is started or resumed by the operating system the only register you can assume is correct is the program counter. Therefore, when a task relinquishes control it should ensure that any working data is saved before relinquishing or that it only relinquishes control at a point in the routine where it is safe to do so.

INSTALLATION & FILE LOCATIONS

P-RTOS comprises a set of include files which are distributed as a plug-in installation. They can be downloaded from the PDS Forum or from the Amicus18 site.

INSTALLATION

Download and save **P-RTOS Installation Proton.zip** (PDS) or **P-RTOS Installation.zip** (Amicus).

Run the installation executable.

FILE LOCATIONS

Once the install program has run the P-RTOS files will be installed as shown below:

Amicus File Locations:

Directory:

Win 7 installations	c:\Program Files (x86)\AmicusIDE\Includes\P-RTOS
XP Installations	c:\Program Files\AmicusIDE\Includes\P-RTOS

Files: Amicus18_Timers.inc
P-RTOS Defines.inc
P-RTOS HUSART.bas
P-RTOS Macros.inc
P-RTOS Main.bas
P-RTOS Vars.inc

PDS File Locations:

Directory:

Win7 installations	c:\Program Files (x86)\ProtonIDE\PDS\Includes\P-RTOS
XP Installations	c: \Program Files\ProtonIDE\PDS\Includes\P-RTOS

Files: P-RTOS Defines.inc
P-RTOS HUSART.bas
P-RTOS Macros.inc
P-RTOS Main.bas
P-RTOS TimerDefs.inc
P-RTOS Timers.inc
P-RTOS Vars.inc

UNINSTALLING P-RTOS

To un-install P-RTOS from the **Proton** or **Amicus IDE** choose **View**, **Plugin**, **Uninstall** and select the **P-RTOS** menu item. This will uninstall the above files and remove the subdirectory providing there are no other files in that directory.

REFERENCE

PDS RTOS uses a co-operative scheduler which requires that certain rules must be obeyed when writing applications to run under RTOS. Ignoring these rules will stop RTOS working.

EVERY TASK MUST HAVE A CONTEXT SWITCH

PDS RTOS tasks must have at least one context switch. RTOS calls which will execute a context switch are identified from other calls by the prefix "OS_". Non-context switching calls are prefixed just with "OS" i.e. there is no underscore. Here is an example of a correctly constructed task.

```
MyTask:
Repeat
      Do something...
      OS_Delay 10
Forever
```

Here MyTask uses a context switch which will switch back to the OS through OS_Delay. The OS will then run MyTask again after 10 OS ticks. Note the Repeat - Forever construct. All tasks should be written as an infinite loop. The Forever keyword is an RTOS macro which equates to 'Until 1 = 0'.

Here are some examples of Task constructs which will fail under RTOS.

```
UncontrolledTask:
      Toggle PORTD.0
```

This task, once started, will not pass control back to RTOS and the application will continue to execute whatever instructions follow.

```
GreedyTask:
Repeat
      Toggle PORTD.0
Forever
```

This task, once started, will continually loop but, as it never calls a context switch, control will never be returned to the OS and no other tasks will run.

CONTEXT SWITCHES CAN ONLY OCCUR IN TASKS

The only state that is saved when Context switching in RTOS is the program counter. It is not good practice to context switch from a subroutine called from a task because of the issues of possible re-entrancy and context saving. Always wait until the function has returned back to the task before context switching.

MANAGE YOUR OWN VARIABLES

You should design your task so that it specifically saves any working variables that it needs when it resumes. Alternatively write your task so that it context switches at a point where there is no need for any working variables to be saved.

RTOS SERVICES

The following details all the user calls which can be made to RTOS. All services are accessed via a macro to maintain a consistent calling interface.

CONTEXT SWITCHING SERVICES

All context switching services are prefixed with OS_.   These calls should only ever be made from within a task and will return control to the scheduler.

OS_DELAY

Syntax:	`OS_Delay DelayTicks`
Description:	Stops the current Task and returns to scheduler which will resume the task after DelayTicks of the OS. A DelayTicks of 0 will have the same effect as calling OS_STOP although this is not the most efficient method of stopping a task.
Parameters:	DelayTicks Word size variable
Requires:	OSENABLE_TIMER services to be set true.

OS_DESTROY

Syntax:	`OS_Destroy`
Description:	Destroys the current task and returns to the scheduler. Removes the record of the task in RTOS leaving the Task Control block to which it was assigned free to be used by another task. You will have to call OSCreateTask before this task can be used again.
Parameters:	None

OS_REPLACE

Syntax:	`OS_Replace TaskPtr, Priority`
Description:	Replaces the current task with the task specified at the priority specified and returns to the scheduler. The new task will occupy the same Task Control Block as the existing task and so will have the same TaskID.
Parameters:	TaskPtr: Pointer to the New task to replace current task. (The Label of the new Task). Priority: The priority to be assigned to the new task.

OS_SETPRIO

Syntax:	`OS_SetPrio Priority`
Description:	Changes the priority of the current task to the Priority level defined and returns to the scheduler. If more than one task exists at the new priority level this task will added into the list of tasks at the new priority.

Parameters: Priority: Byte variable defining the priority. Ranging from 0 (OSHIGHEST_PRIO) through to OSPRIORITY_COUNT -1 (OSLOWEST_PRIO)

OS_STOP

Syntax: `OS_Stop`

Description: Stops the current task and returns to the scheduler. The task can only be restarted from OSStartTask when the task will resume from its last program counter position.

Parameters: none

OS_WAITBINSEM

Syntax: `OS_WaitBinSem EventID, TimeOut`

Description: Suspends task until the binary semaphore referenced in EventID has been signalled or the Timeout has elapsed. If the Event is already signalled when the wait is called the Task will continue without context switching. If the wait times out the Task will be resumed with the timeout flag set. If the Event is signalled, the Task will be resumed with the timeout flag cleared.

This function can only be called after the referenced event has been created.

Parameters: EventID: Pointer to the associated event control block
Timeout: a byte variable specifying the number of OS Ticks before timing out. Set to OSNO_TIMEOUT to wait indefinitely.

Requires: OSENABLE_EVENTS and OSENABLE_SEMAPHORES to be set to true.

OS_WAITEFLAG

Syntax:

Description: Not implemented yet.

Parameters:

OS_WAITMSG

Syntax: `OS_WaitMsg EventID, Timeout, (PMessage)`

Description: Suspends the current task until the message is signalled by another task or the timeout has elapsed. A message could be any variable type so it is necessary that both sender and receiver tasks expect the same data type and size. When signalled will return with a pointer to the start of the message in PMessage. If the timeout has elapsed it will return with the Timeout flag set. The value returned in PMessage will be undefined. Set Timeout to OSNO_TIMEOUT to wait indefinitely.

As PDS doesn't recognise pointers as a data type you would typically use the SFR registers to indirectly address the message.

This function can only be called after the referenced event has been created.

Parameters: EventID: Pointer to the associated event control block.
Timeout: a byte variable specifying the number of OS Ticks before timing out. Set to OSNO_TIMEOUT to wait indefinitely.
Message Pointer Placeholder for the returned pointer.

Requires: OSENABLE_EVENTS and OSENABLE_MESSAGES to be set to true.

OS_WAITSEM

Syntax: `OS_WaitSem EventID, Timeout`

Description: Suspends the current task until the counting semaphore referenced in EventID has been signalled or the timeout has been elapsed. If the semaphore value is 0 it remains waiting and returns to the scheduler . If the semaphore is non-zero it will decrement the semaphore value and continue without context switching. If the timeout expires before the semaphore value has reached zero continue execution with the timeout flag set. Set timeout to OSNO_TIMEOUT to wait indefinitely.

This function can only be called after the referenced event has been created.

Parameters: EventID: Pointer to the associated event control block.
Timeout: a byte variable specifying the number of OS Ticks before timing out. Set to OSNO_TIMEOUT to wait indefinitely.

Requires: OSENABLE_EVENTS and OSENABLE_SEMAPHORES to be set to true.

OS_YIELD

Syntax: `OS_Yield`

Description: Unconditionally Yields to the scheduler. If no other task is waiting to run will resume at next instruction after OS_Yield.

Parameters: None

NON-CONTEXT SWITCHING SERVICES

The following calls to RTOS do not initiate a context switch. In general these can be called from anywhere in your application.

OSCREATEBINSEM

Syntax: `OSCreateBinSem EventID, BinSem`

Description: Assign an Event Control Block to a binary semaphore and set its initial value. (True or False)

Parameters: EventID: Pointer to the associated event control block.
BinSem: Initial values assigned to the binary semaphore (True or False)

OSCREATECYCTMR

Syntax:	`OSCreateCycTmr TmrTaskPtr, TaskID, Delay, Period, Mode`
Description:	Assign a Task Control Block to a Cyclic timer. Cyclic Timers are structured like conventional subroutines, starting with a start address and finishing with a Return.
Parameters:	TmrTaskPtr: Start Address of the Cyclic Timer code. TaskID: Pointer to the associated Task Control Block Delay: Initial delay in OS Ticks before calling the task for the first time. Period: The time in OS Ticks between successive calls of the Cyclic timer Mode: The timer can have one of 2 modes operating mode, OSCT_ONE_SHOT and OSCT_CONTINUOUS. If you don't want the Timer to start when you have created it And OSCT_DONT_START_CYCTMR with your chosen mode.

OSCREATEEFLAG

Syntax:	
Description:	Not Implemented.
Parameters:	

OSCREATEMSG

Syntax:	`OSCreateMsg EventID, PMessage`
Description:	Assigns an Event Control Block to a Message Event and allocates the Message Pointer. If the message is to be signalled immediately the PMessage should point to an existing message otherwise set the PMessage to 0.
Parameters:	EventID: Pointer to the associated event control block. PMessage: Pointer to the message

OSCREATESEM

Syntax:	`OSCreateSem EventID, Sem`
Description:	Assign an Event Control Block to a counting semaphore and set its initial value.
Parameters:	EventID: Pointer to the associated event control block. Sem: Byte - Initial value assigned to the semaphore count.
Requirements:	OSENABLE_EVENTS and OSENABLE_SEMAPHORES

OSCREATETASK

Syntax:	`OSCreateTask TaskPtr, Priority`
Description:	Assign a task control block to a the task defined in TaskPtr.

Parameters: TaskPtr: Address of the task you wish to assign. This would normally be the Label at the start of the task.
Priority: Byte Variable defining the priority you wish the task to run at. The value must lie between OSHIGHEST_PRIO and OSLOWEST_PRIO.

OSDESTROYCYCTMR

Syntax: `OSDestroyCycTmr  TaskID`

Description: Destroys the Cyclic timer task identified by TaskID. Removes the reference to the cyclic timer leaving the Task Control block to which it was assigned free to be used by another task. You will have to call OSCreateCycTmr before this Cyclic Timer can be used again.

Parameters: TaskID: Pointer to the associated Task Control Block for the timer.

OSDESTROYTASK

Syntax: `OSDestroyTask TaskID`

Description: Destroys the task identified by TaskID. Removes the notification of the task in RTOS leaving the Task Control block to which it was assigned free to be used by another task. You will have to call OSCreateTask before this task can be used again.

Parameters: TaskID: Pointer to the associated Task Control Block for the Task.

OSGETPRIO

Syntax: `OSGetPrio`

Description: Returns the priority of the active task.

Parameters: None

OSGETPRIOTASK

Syntax: `OSGetPrioTask TaskID`

Description: Returns the priority of the task defined in TaskID.

Parameters: TaskID: Pointer to task control block of the referenced task

OSGETSTATE

Syntax: `OSGetState`

Description: Returns the state of the current task, always OSTCB_TASK_RUNNING. Included for completeness only

Parameters: None

OSGETSTATETASK

Syntax: `OSGetStateTask TaskID`

Description: Returns the state of the task identified by TaskID. Possible values are:

OSTCB_DESTROYED	Destroyed or uninitialised
OSTCB_TASK_STOPPED	Task Stopped
OSTCB_TASK_DELAYED	Delayed n OSTicks
OSTCB_TASK_WAITING	Waiting on an event
OSTCB_TASK_WAITING_TO	Waiting and event with a timeout
OSTCB_TASK_ELIGABLE	Ready to run
OSTCB_TASK_RUNNING	Running

Parameters: TaskID: Pointer to task control block of the referenced task

OSGETTICKS

Syntax: `OSGetTicks`

Description: Returns the current system timer in ticks.
The size of the return value will be determined by OSTICK_CTR_SIZE

Parameters: None

OSINIT

Syntax: `OSInit`

Description: This function must be called before calling any other RTOS functions. It initialises the RTOS setting up the task and event control blocks and starting the timer and events if necessary. OSInit relies on a number of configuration settings which you must define prior to calling OSInit. These are described more fully in the Configuration chapter.

Parameters: None

OSQUEUE_EVENT

Syntax: `OSQueue_Event EventID&Type, {AddrLow, AddrHigh}`

Description: This function will add a Signal event to a queue for processing when the scheduler next runs. It is designed to be used in interrupt routines where you need to keep the time spent in the interrupt to a minimum.   For more information see "Using Events with Interrupts".

Parameters: EventID&Type: a byte parameter which contains the Event ID in the lower 5 bits of the byte and the event type in the upper 2 bits.

The event type can be: OSCNTSEMEVNT, OSBINSEMEVNT, OSMSGEVNT, OSEFLAGEVNT and should be OR'ed with the Event ID.

AddrHigh, AddrLow   If a Message event or EFlag event two additional bytes are used to carry the address of the message or EFlag.

OSREADMSG

Syntax: `OSReadMsg EventID`

Description: Returns a pointer to the current message. Returns 0 if there is no message This function has no effect on the Message Events.

Parameters: EventID: Pointer to the associated event control block.

OSREADBINSEM

Syntax: `OSReadBinSem EventID`

Description: Returns the value (True or False) of the BinSem identified by EventID. This function has no effect on the binary semaphore.

Parameters: EventID: Pointer to the associated event control block.

OSREADSEM

Syntax: `OSReadSem EventID`

Description: Returns the value $0 ..$FF of the counting semaphore specified in EventID. This function has no effect on the binary semaphores

Parameters: EventID: Pointer to the associated event control block.

OSRESETCYCTMR

Syntax: `OSResetCycTmr TaskID`

Description: Resets the Cyclic timer specified in TaskID to its initial conditions after OSCreateCycTmr. This means that the timer will start with the defined initial delay.

Parameters: TaskID: Pointer to task control block of the referenced task:

OSSCHED

Syntax: `OSched`

Description: Runs the highest priority eligible task. This function must be called continuously from your main program to continue multitasking. It must be called after OSInit.

Typically your main program would call OSSched like this:

```
Repeat
      OSSched
Forever
```

Every time a task yields it will return to the main program which should call OSSched. If the main program stops calling OSSched then multitasking will cease.

Parameters: None

OSSETPRIO

Syntax: `OSSetPrio Priority`

Description: Changes the priority of the current task.

Parameters: Priority: Byte variable defining the new priority (0 is highest priority)

OSSETPRIOTASK

Syntax: `OSSetPrioTask TaskID, Priority`

Description: Changes the priority assigned to the task identified in TaskID.

Parameters: TaskID: Pointer to task control block of the referenced task
Priority: Byte variable defining the new priority.

OSSETTICKS

Syntax: `OSSetTicks TickValue`

Description: Initialises the value of the OS Tick Counter to TickValue

Parameters: TickValue: Byte, Word or DWord depending on OS_TICK_SIZE

OSSIGNALBINSEM

Syntax: `OSSignalBinSem EventID`

Description: Signals a binary semaphore. If one or more tasks are waiting this semaphore the highest priority t ask waiting will be made eligible to run. The task will run when it becomes the highest priority eligible task.

Parameters: EventID: Pointer to the associated event control block.

OSSIGNALMSG

Syntax: `OSSignalMsg EventID, PMessage`

Description: Signals a message is ready to be read by any waiting task. The message address passed in PMessage.

Parameters: EventID: Pointer to the associated event control block.
PMessage: Pointer to the Message.

OSSIGNALSEM

Syntax: `OSSignalSem EventID`

Description: Increments a counting semaphore. If one or more tasks are waiting this semaphore the highest priority t ask waiting will be made eligible to run. The task will run when it becomes the highest priority eligible task.

Parameters: EventID: Pointer to the associated event control block.

OSSTARTCYCTMR

Syntax: `OSStartCycTmr TaskID`

Description: Starts a cyclic timer. If the timer has never been run since it was created or reset then the it will start with the initial delay. If the timer had previously been run it will start with the period value.

Parameters: TaskID: Pointer to task control block of the referenced task

OSSTARTTASK

Syntax: `OSStartTask TaskID`

Description: Starts a dormant or stopped task identified by TaskID

Parameters: TaskID: Pointer to task control block of the referenced task

OSSTOPCYCTMR

Syntax: `OSStopCycTmr TaskID`

Description: Stops a Cyclic Timer identified by TaskID

Parameters: TaskID: Pointer to task control block of the referenced task

OSSTOPTASK

Syntax: `OSStopTask TaskID`

Description: Makes a task identified by TaskID ineligible.

Parameters: TaskID: Pointer to task control block of the referenced task

OSTRYBINSEM

Syntax: `OSTryBinSem EventID`

Description: Behaves like OS_WaitBinSem but does not context switch from the current task.
As it doesn't context switch it can be used outside a task. Typically this would be used in a ISR to handle an external event.

Parameters: EventID: Pointer to the associated event control block.

OSTRYMSG

Syntax: `OSTryMsg EventID`

Description: Behaves like OS_WaitMsg but does not context switch from the current task.
As it doesn't context switch it can be used outside a task.

Parameters: EventID: Pointer to the associated event control block.

OSTRYSEM

Syntax: `OSTrySem EventID`

Description: Behaves like OS_WaitSem but does not context switch from the current task.
As it doesn't context switch it can be used outside a task. Typically this would be used in an ISR to handle outgoing data.

Parameters: EventID: Pointer to the associated event control block.

ADDITIONAL USER SERVICES

The following services will be available if enabled by the relevant Option.

CLEAR_SERIAL_BUFFER

Syntax: `Clear_Serial_Buffer`

Description: Disables USART interrupts, clears the serial receive and transmitter buffers to zero and re-enables USART Interrupts.

Requires: Buffered serial in must be enabled (`$define OSENABLE_BUFFERED_SERIN`) and optionally buffered serial out (`$define OSENABLE_BUFFERED_SEROUT)`

OS_HRSOUT

Syntax: `OS_HRSOut Item`

Description: This is a context switching function which combines the PDS HRSOut command with OS_WaitTxBuffer to make a single inline command. Use is as if you were using the standard PDS HRSOut command but limited to only one Item. If the output buffer contains data when this function is called it will return to the OS until such time that the buffer is empty.

Parameters: A constant, variable or string list. All modifiers are supported.

Requires: Buffered serial out must be enabled (`$define OSENABLE_BUFFERED_SEROUT)`

OS_WAITTXBUFFER

Syntax: `OS_WaitTxBuffer`

Description: Waits current task until the Tx Buffer is empty. This would normally be used before calling HRSOut when there is a possibility that the USART is still transmitting data. If you can be confident that there is room in the Tx buffer for the data you wish to send this can be skipped.

Requires: Buffered serial out must be enabled (`$define OSENABLE_BUFFERED_SEROUT)`

OS_HRSIN

Syntax: `OS_HRSIn (Timeout, Item)`

Description: This is a context switching function which combines the PDS HRSIn command with OS_WaitRxBuffer to make a single in line command. Use it in a similar way you would with HRSIn with the following exceptions:

Only the second format of the PDS command is supported
Timeout label and Parity label are not supported.
Timeout is not an optional parameter, use OSNO_TIMEOUT if not required.
Only one variable is supported. All modifiers are supported.
Wait for sequence and Skip are not supported.

This function will return to the OS until a character has been received in the input buffer.

Requires: Buffered serial in must be enabled (`$define OSENABLE_BUFFERED_SERIN)`

OS_WAITRXBUFFER

Syntax: `OS_WaitRxBuffer (Timeout)`

Description: Waits current task until the Rx Buffer contains at least one byte of data or the timeout period has elapsed. This would normally be used before calling HRSIn so that you only continue when data is there to be processed. If a timeout has been specified test the OSTIMEDOUT flag for timeout.

Parameter: Timeout value specified in OS Ticks or OSNO_TIMEOUT.

Requires: Buffered serial in must be enabled (`$define OSENABLE_BUFFERED_SERIN`)

INIT_USART_INTERRUPT

Syntax: `Init_Usart_Interrupt`

Description: This will set up the interrupts for the USART. It will set interrupt to high Priority, create the RxCharEv event and RxErrEv event if required, and clear the receive buffer. This will be automatically called by the RTOS Init function if buffered serial in is enabled. If buffered serial out is enabled the routine will additionally clear the Tx ready interrupt and clear the transmit Buffer.

Note. The user must set the USART baud rate and I/O register settings in the application's initialisation routines.

Requires: Buffered serial in must be enabled (`$define OSENABLE_BUFFERED_SERIN`)
and optionally buffered serial out (`$define OSENABLE_BUFFERED_SEROUT`)

OSCYCTMRRUNNING

Syntax: `OSCycTmrRunning TaskID`

Description: Returns True is Cyclic Timer referenced in TaskID is running.

Requires: OSENABLE_CYCLIC_TIMERS to be enabled. (**`$define OSENABLE_CYCLIC_TIMERS True`**)

OSTASKSTOPPED

Syntax: `OSTaskStopped TaskID`

Description: Returns true if Task defined by TaskID is running, False if task is stopped

OSTIMEDOUT

syntax: `OSTimedOut`

Description: True if current task has timed out. Valid after calling OS_WaitSem, OS_WaitBinSem or OS_WaitMsg.

Requires: OSENABLE_EVENTS to be enabled (**`$define OSENABLE_EVENTS True`**)

USER MACROS

This section describes some additional macros which are provided to simplify usage.

OSTCBP(X)	Returns a pointer value to a specific Task Control Block (TCB) within the TCB array. Use this to create an alias to a TCB. E.g. Symbol MyTaskPtr = OSTCBP(3) OSCreateTask MyTaskPtr, MyTask
OSECBP(X)	Returns a pointer value to a specific Event Control Block (ECB) within the ECB array.
OSEFCBP(X)	Returns a pointer value to a specific Event Flag Control Block (EFCB) Note - In all the above macros ensure that X does not exceed the number of Tasks, Events or Event Flags set in the Configuration settings
Forever	Returns 1 = 0 - use this for endless Repeat loops.

OS HOOKS

The following hooks and return labels should be used when writing user ISR routines:

OSTICK_ISR_ISR	This is the label the OS will jump to if the OSTICK_SOURCE is set to ext.
OSTICK_ISR_RTN	This label should be used at the end of your TICK ISR. E.g. Goto OSTICK_ISR_RTN See also configuration setting OSTICK_SOURCE
OSUSER_ISR _RTN	This return label should be used at the end of your USER ISR. E.g. GoTo OSUSER_ISR_RTN See also configuration setting OSISR_USER_HOOK

ERROR CODES

The following error codes are returned in RTOS. The error code will be returned in OSRESULT after calling any OS function.

OSNOERR	Function successful
OSCREATEERR	An error occurred while attempting to create a Task, Event or Cyclic Timer.
OSEVENTERR	An error occurred while attempting to Wait or signal and event.
OSPRIOERR	An error occurred while attempting to change the priority of a task
OSTASKERR	An error occurred while attempting to start, stop replace or destroy a task.
OSCYCTMRERR	An error occurred while attempting using a Cyclic timer function.

CONFIGURATION

PDS RTOS provides a number of configuration options which you can use to tailor the RTOS features to suit your requirements and minimise the size of your program.

These settings use the PDS pre-processor commands and should be placed at the beginning of your main program.

OSTASKS_COUNT

Syntax: `$define OSTASKS_COUNT N` *(where N is an integer between 0 and 32)*

Description: Sets the maximum number of tasks supported. RTOS will allocate 8 bytes of RAM per task up to a maximum of 32 tasks (256 bytes). If OSTASKS_COUNT is not defined it will default to 4 tasks.

OSPRIO_COUNT

Syntax: `$define OSPRIO_COUNT N` *(where N is an integer between 0 and 15)*

Description: Sets the number of priority levels supported. RTOS will allocate 3 bytes of RAM for each priority level up to a maximum of 16 levels (48 bytes). If OSPRIO_COUNT is not defined it will default to 4 priority levels.

OSENABLE_INTERRUPTS

Syntax `$define OSENABLE_INTERRUPTS True/False`

Description: OSENABLE_INTERRUPTS must be set true if you are using Timer services OSENABLE_TIMER, OSENABLE_TIMEOUTS), RTOS USART service (OSENABLE_BUFFER_SERIN) or a User define ISR (OSISR_USER_HOOK).

OSENABLE_TIMER

Syntax: `$define OSENABLE_TIMER True/False`

Description: Enables the RTOS timer services. Timer services are required to use Delays, Timeouts or Cyclic Timers. If not defined OSENABLE_TIMER will default to False.

This option must be set true to use any of the following options: OSENABLE_TIMEOUTS, OSTICK_SOURCE, OSTIMER_PRESCALE, OSTIMER_PRLOAD, OSTICK_CTR_SIZE, OSENABLE_CYCLIC_TIMERS.

OSENABLE_TIMEOUTS

Syntax: `$define OSENABLE_TIMEOUTS True/False`

Description: Enables timeouts to be used on OS_Wait... calls. If not defined will OSENABLE_TIOMEOUTS will default to False.

OSISR_USER_HOOK

Syntax: `$define OSISR_USER_HOOK ISR Label`

Description: Used to define a label to a user defined Interrupt Subroutine to handle additional sources of interrupt. E.g. Keyboard entry or switch contact etc. When an interrupt is received RTOS will perform a context save and Jump to the label defined in OSISR_USER_HOOK. The ISR must return to RTOS by jumping to OSISR_RTN. Typically the user ISR should clear the interrupt save any necessary data and signal an event which RTOS will use to trigger the appropriate task to processes the event.

OSISR_USER_HOOK_INIT

Syntax: `OSISR_USER_HOOK_INIT` *`ISRInit Label`*

Description: Used to define a user defined initialisation subroutine for the User interrupt service routine see OSISR_USR_HOOK.

OSTICK_SOURCE

Syntax: `$define OSTICK_SOURCE` *`T0/T1/EXT`*

Description: Defines the tick source for RTOS. OSTICK_SOURCE values of T0 or T1 will define the tick source as Timer0 or Timer1. To configure the timers use OSTIMER_PRESCALE and OSTIMER_PRELOAD.

Setting the OSTICK_SOURCE value to EXT enables an external interrupt source to be the RTOS Tick Source. This will bypass the RTOS tick timer initialisation and interrupt handling and use instead user defined initialisation and interrupt service routine. During RTOS initialisation RTOS will call a user defined a initialisation routine OSTICK_EXT_INIT. On interrupt RTOS will perform a context save and jump to OSTICK_EXT_ISR. This ISR should jump back to OSTICK_ISR_RTN.

OSTIMER_PRESCALE

Syntax: `$define OSTIMER_PRESCALE` *`Off/0..7`*

Description: This parameter allows you to choose a Timer Pre-scale value. For Timer0 the value can range from 0 to 7 and for Timer1 the value can range from 0 to 3. If undefined OSTIMER_PRESCALE will default to Off.

OSTIMER_PRELOAD

Syntax: `$define OSTIMER_PRELOAD` *`$NNNN`*

Description: This parameter is the value loaded into Timer 0 or Timer 1 when the OSTICK_SOURCE is T0 or T1. If this define is omitted and T0 or T1 is selected a compile error will be reported.

OSTICK_CTR_SIZE

Syntax: `$define OSTICK_CTR_SIZE` *`1/2/4`*

Description: The tick counter increments for each RTOS tick and rolls over back to 0 on overflow. The tick counter can be a byte(1), word(2) or double word (4). If not defined OSTICK_CTR_SIZE will default to byte size.

OSENABLE_CYCLIC_TIMERS

Syntax: `$define OSENABLE_CYCLIC_TIMERS True/False`

Description: Enables cyclic timers to be used. If not defined OSENABLE_CYCLIC_TIMERS will default to False.

OSENABLE_EVENTS

Syntax: `$define OSENABLE_EVENTS` *`True/False`*

Description: Enables the RTOS Events services. Event services are required to support semaphores, event flags and messages. If not defined OSENABLE_EVENTS will default to False.

This option must be set true to use any of the following services: OSENABLE_MESSAGES, OSENABLE_SEMAPHORES and OSENABLE_EVENT_FLAGS.

OSEVENTS_COUNT

Syntax: `$define OSEVENTS_COUNT` *`N`*

Description: Sets the maximum number of events supported. RTOS will allocate 2/3 bytes of RAM per event depending on the event types are enabled up to a maximum of 32 events (64/96 bytes). If OSEVENTS_COUNT is not defined it will default to 4 events.

OSENABLE_MESSAGES

Syntax: `$define OSENABLE_MESSAGES` *`True/False`*

Description: Enables Message services to be supported. If not defined OSENABLE_MESSAGES will default to False.

OSENABLE_SEMAPHORES

Syntax: `$define OSENABLE_SEMAPHORES` *`True/False`*

Description: Enables binary and counting Semaphore services to be supported. If not defined OSENABLE_SEMAPHORES will default to False.

OSENABLE_EVENT_FLAGS

Syntax: `$define OSENABLE_EVENT_FLAGS` *`True/False`*

Description: Enables Event flag services to be supported. If not defined OSENABLE_EVENT_FLAGS will default to False.

OSEVENT_FLAGS

Syntax: `$define OSEVENT_FLAGS` *`N`*

Description: Defines the number of event flags supported. Each event flag requires one byte of RAM. If not defined and OSENABLE_EVENTS is True OSEVENT_FLAGS will default to 2.

OSENABLE_BUFFERED_SERIN

Syntax: `$define OSENABLE_BUFFERED_SERIN`

Description: Introduces a buffered serial input function to RTOS using the hardware USART. Received characters are placed in a buffer and a RxCharEv counting semaphore event is raised. The value of the semaphore contains the number of received bytes in the buffer.

You can access these characters using the PDS **HRSIn** command. There are 2 ways you can use this command. To read the whole buffer without context switching put HRSIn in a loop reading the buffer until OSTRYSEM RxCharEv returns 0. Alternatively to context switch back on each read use OS_WaitTRxBuffer. If no other task has a higher priority RTOS will return to this task until the receive buffer is empty. See also OS_HRSIn.

When enabled, buffered serial in will create a new Event (RxCharEv) which will occupy the last slot in the Event Control block. If OSSER_ENABLE_ERROR is enabled an additional event will be created (RxErrEv). If buffered serial out is not enabled, the error event will occupy the last but one slot in the event control block. If buffered serial out is enabled, the error event will occupy the second from last slot in the event control block. When determining the size of the event control block you must remember to include these additional events.

Important: when used in this configuration **HRSIn** does not support the PDS HRSIn timeout function, instead call OS_WAITSEM RxCharEv with a timeout value. The OSTimedOut flag will return true if the event has timed out.

OSENABLE_BUFFERED_SEROUT

Syntax: `$Define OSENABLE_BUFFERED_SEROUT`

Description: Introduces a buffered serial out function to RTOS using the Hardware USART. You can use the PDS HRSOut function to load the buffer which will be emptied automatically by RTOS at the rate dictated by the USART TX Baud rate.

Be careful not to place more characters into the buffer than can be accommodated by the buffer. If you need to send strings of data which are longer that the buffer size break up the string and use repeated HRSOut commands, each command being preceded by a OS_WaitTxBuffer. See also OS_HRSOut.

When enabled, buffered serial out will create a new Event (TxEmptyEv) which will occupy the penultimate slot in the Event control block.

OSSERIN_BUFFER_SIZE

Syntax: `$define OSSERIN_BUFFER_SIZE` *N*

Description: Defines the buffer size in bytes for the incoming serial data. Must not exceed 256 bytes. Defaults to 64 bytes. OSSERIN_BUFFER_SIZE will reserve N bytes at the top of RAM for the buffer.

OSSEROUT_BUFFER_SIZE

Syntax: `$define OSSEROUT_BUFFER_SIZE` *N*

Description: Defines the buffer size in bytes for the outgoing serial data. Must not exceed 256 bytes. Defaults to 64 bytes. OSSEROUT_BUFFER_SIZE will reserve N bytes at the top of RAM for the buffer.

OSSER_ERROR_EVENT

Syntax: `$define OSSER_ERROR_EVENT`

Description: When serial receive errors occur if this Option is **not** defined the error flags will be cleared automatically, no receive event will be generated and the incoming data will be effectively ignored. If this option is defined the error flags will not be cleared and a `RxCharErr` event will be signalled.

To use this function you will need to add a task which waits on this event.

WHAT'S GOING ON INSIDE

This section is written to give you an outline of what the RTOS is doing internally.

CONTROL BLOCKS

Core to the operation of the RTOS are control blocks. These hold the data associated with tasks and events.

TASK CONTROL BLOCK

The task control block (TCB) is a block of 8 bytes. TCBs are held in a byte array. Each TCB looks like this...

	7	6	5	4	3	2	1	0	
TASK ID	STATE				PRIORITY				*Task State and Priority*
+ 1	DELAY (OSTicks) or EVENTID								*LS of a Delay or a Waiting EVENT id*
+ 2	DELAY (OSTicks) or TIMEOUT								*MS of a Delay or Timeout in OS Ticks*
+ 3	EVENT FLAG MASK								*Not implement at present*
+ 4	PROGRAM COUNTER LSB								*LS address of next instruction*
+ 5	PROGRAM COUNTER MSB								*MS address of next instruction*
+ 6	PREV TASK ID								*Pointer to Previous task at this priority*
+ 7	NEXT TASK ID								*Pointer to Next task at this priority*

STATE can have the following values:

Ready	$00	*Task is ready to run*
Delay	$10	*Waiting 'N' OS Ticks*
Event	$20	*Waiting an Event forever*
Event w/to	$30	*Waiting an Event with a timeout*
CycTmrOff	$40	*Cyclic timer stopped*
CycTmrOn	$50	*Cyclic Timer running*
Pending	$C0	*Task stopped*
Dormant	$FX	*Task not present - deleted*

PRIORITY can have any value between 0 (Highest) to OSPRIO_COUNT (Lowest)

Program Counter When a task Yields to the OS, the Program Counter in the TCB holds the address of the next instruction to be executed when the task is next run.

Tasks which share the same priority level will execute on a round robin basis. The TCBs for these tasks form a linked list using the PREV and NEXT task IDs.

EVENT CONTROL BLOCK

The event control block(ECB) is a block of 2 or 3 bytes depending on the which specific event options have been enabled.

	7	6	5	4	3	2	1	0	
EVENTID	SIGNALLED				TYPE				*Signalled state and Event Type*
+ 1	SEM VAL, MSG PTR or EFLG PTR								*Semaphore value or pointers to Msg or EFlag*
+ 2	MSG PTR MSB								*MS of a Delay or Timeout in OS Ticks*

The event type field can carry the following codes:

Message	$01	The event is a Message event
Counting	$02	The event is a counting semaphore
Binary	$03	The event is a Binary semaphore
Event Flag	$04	The event is a EFlags event (Currently not implemented)

The event Signalled field has the following states:

Not signalled	$00
Signalled	$FF

DISPATCH LIST

The dispatch list is an array of bytes, OS_PRIO_COUNT deep. The array is ordered in priority representing priority level 0 through OSPRIO_COUNT. Each location holds the TaskID of the next task to be run or $FF if no task is to run. The scheduler runs through each location starting at 0 until it finds a TaskID. It makes this task the Active Task and updates the dispatch list at the current priority with the next task to run if any. This is done by scanning the Task Control blocks for a Ready state task at the current priority.

FIRST TASK IN and LAST TASK IN are 2 additional arrays which hold the TaskIDs of the first task and most recent task respectively added at that priority. It is used to maintain the order of tasks sharing the same priority so that an equitable Round Robin scheme operates.

EVENTS

An event has to be created before it can be used. When a task calls a Wait on a semaphore or message it can do one of 2 things: If the event has already been signalled the task will continue without context switching. If it hasn't been signalled the task will context switch. For Events to operate correctly when a waiting task is signalled it is necessary to call the Wait again in order that it can process the event as a signalled Event. This means the task will resume at the Wait instruction as it is now signalled whereas in all other cases of Yielding the task will resume at the instruction following the yield.

USING EVENTS WITH INTERRUPTS

Any part of the system can signal an event which will normally be registered immediately unless called from an interrupt service routine (ISR). In order to keep the ISR as short as possible and to prevent the corruption of data, events should be placed in an events queue which will be processed next time the scheduler runs.

The RTOS Macro provided for this is `OSQueue_Event`.

However, it may be preferable to build this function into your own ISR so below are the details necessary to update the Event queue directly:

Variables used:

EVNTQ Array of byte	The event queue
EVNTQ_IN Byte	Index into the queue of the most recent event added
EVNTQ_OUT	Index into the queue of the most recent event extracted

Counting and Binary semaphores use a single byte, Message events and event flags use 2 additional bytes to carry the address of the message.

Event Queue First Byte

B7	B6	B5	B4	B3	B2	B1	B0
Event Type		X	Event ID				

B6-Bt Event Type

00	Counting Semaphore	Symbol OSCNTSEMEVNT
01	Binary Semaphore	Symbol OSBINSEMEVNT
10	Message Semaphore	Symbol OSMSGEVNT
11	Event Flags	Symbol OSEFLAGEVNT

B4-B0 Event ID

Event Queue second and third bytes (Message and EFlag Events only)

Second byte	LS byte of address
Third Byte	MS byte of address.

Example code:

```
Dim EVNT_FSR0 As FSR0L.Word

' Save FSR0 before entering this code...
EVNT_FSR0 = VarPtr EVNTQ              ' Put address of Start of EVNTQ in FSR0
Inc EVNTQ_IN                          ' Increment In pointer to next slot
If EVNTQ_IN >= SizeOf (EVNTQ) then EVNTQ_IN = 0      ' reset to 0 if overrun
EVNT_FSR0 = EVNT_FSR0 + EVNTQ_IN      ' Add pointer in to EVNTQ start address
INDF0 = EventID | EventType           ' Load EVNTQ with signalled EventID and type
' If message or eflag event type repeat for the 2 additional bytes.
' Restore FSR0 after this code...
```

RESERVED WORDS

Below is a list of protected words that P-RTOS uses internally. Be sure not to use any of these words as variable or label names, otherwise errors will be produced.

In general all variables or names starting with OS and _OS

Forever, TCBs, ECBs, EFCBs, labels beginning with t_, e_ (incomplete)