PRTOS

REAL TIME OPERATING SYSTEM

FOR

PROTON DEVELOPMENT SYSTEM

INTRODUCTION

RTOS is a Real Time Operating System designed 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 which 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 have 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 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 will carry a value typically it could be used to indicate the number of bytes in an input buffer.

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=1") . In a co-operative RTOS every task **must** 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.

```
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            ' Prescale 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

GoTo Start

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

Dim Ctr As Byte

Symbol T_Count   = OSTCBP(1)
Symbol T_LEDOut = OSTCBP(2)
Symbol T_Delayed2 = OSTCBP(3)
Symbol T_OSCOut = OSTCBP(4)
Symbol T_Delayed = OSTCBP(5)
Symbol T_BinSem = OSTCBP(6)

Symbol E_LedCtrl = OSECBP(1)

CountTsk:
Repeat
    Inc Ctr
    If Ctr = $FF Then OSSignalBinSem E_LedCtrl
    OS_Yield
Forever

LEDOut:
Repeat
    PORTD = Ctr & $3F
    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_LEDOut
    OS_Delay 1
    Toggle PORTA.5
    OSStopTask T_LEDOut
    OS_Delay 20
    OS_Replace DelayedTask, 2
Forever

BinSemTask:
Repeat
    OS_WaitBinSem E_LedCtrl,0
    OSStartTask T_LEDOut
    OS_Delay 1
    OSStopTask T_LEDOut
    OS_Delay 5
Forever

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

'-----Start-----------------------------------------------------------------------------------
Start:
TRISA = %000000         ' All Port A Outputs
TRISB = %00000000       '
TRISD = %00000000       ' All port D pins output
TRISC = %11000000       ' Set port C to output
Ctr   = $00             ' reset ctr

OSInit                  ' Initialise RTOS
OSCreateTask T_Count, CountTsk, 4
OSCreateTask T_LEDOut, LEDOut, 4
OSCreateTask T_OSCOut, OSCOut, 4
OSCreateTask T_Delayed, DelayedTask, 3
OSCreateTask T_BinSem, BinSemTask, 3
OSCreateBinSem E_LedCtrl, 0

OSStartTask T_LEDOut    '
OSStartTask T_Count     '
OSStartTask T_Delayed   ' delayed will start and stop OSCOut
OSStartTask T_BinSem    '

Repeat
    OSSched             ' run scheduler continuously
Forever
```

REFERENCE

PDS RTOS uses a co-operative scheduler which requires that certain rules must be obeyed when writting 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 = 1'.

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

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

This task 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 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 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 (top priority) through to OSPRIORTY_COUNT -1 (Lowest Priority)

OS_STOP

Syntax: `OS_Stop`

Description: Stops the current task and returns to the scheduler. The task can only be restarted from OSStartTask and will the task will resume from its last 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 be resumed if there is no other higher priority task waiting to run. 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.

OS_WAITEFLAG

Syntax:

Description: Not implemented yet.

Parameters:

OS_WAITMSG

Syntax:

Description: Not implemented yet.

Parameters:

OS_WAITSEM

Syntax: OS_WaitSem EventID

Description: Suspends the current task on a counting semaphore. If the semaphore value is 0 it returns to the scheduler. If the semaphore is non-zero it will decrement the semaphore value and continue execution. If the timeout expires before the semaphore value has reached zero continue execution with the timeout flag set.

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

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: Register 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 Or OSCT_DONT_START_CYCTMR with your chosen mode.

OSCYCTMRRUNNING

Syntax: `OSCYCTMRRUNNING TaskID`

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

OSCREATEEFLAG

Syntax:

Description: Not Implemented.

Parameters:

OSCREATEMSG

Syntax:

Description: Not Implemented.

Parameters:

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. OS_Init 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

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

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.

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: `OSSStartCycTmr 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.

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 a ISR to handle outgoing data.

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

OTHER 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)

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_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.

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 allows you to choose an external interrupt source for the RTOS Tick Source. This will bypass the RTOS tick initialisation and interrupt handling and use instead user defined initialisation and interrupt service routine. During RTOS initialisation RTOS will call OSTICK_EXT_INIT. The On_Hardware_Interrupt will jump to a user define ISR called OSTICK_EXT_HDLR. This ISR will be responsible for context saving, detecting the interrupt, flagging a Tick to RTOS and any other interrupt processing required.

To flag an RTOS tick set OSTick_Flag true. This will be cleared by RTOS when it has processed the Tick.

OSTIMER_PRESCALE

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

Description: This parameter allows you to choose a Timer Prescale 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: OSEVENTS_COUNT N

Description: Sets the maximum number of events supported. RTOS will allocate 3 bytes of RAM per event up to a maximum of 32 events (64 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.