Novo RTOS

for use with the BoostC, BoostC++ and BoostBasic Compilers

Reference Manual

V1.08

Index

This page is intentionally left blank.

Novo RTOS

Introduction

The word Novo is Latin for *change*. Novo was given this name because its changes the way you construct you program – once you understanding how to use it, it makes things much easier. It you want to code a system with more than one job (or task) to do, without an RTOS you have little choice but to pursue the classical super loop programming methodology.

Novo was specifically written for use with the SourceBoost Technologies BoostC, BoostC++ and BoostBasic compilers for PICmicro, although it would be quite possible to port it to other compilers and platforms.

Super Loop How it works

With a super loop the code whizzes round in a big loop, calling different pieces of code. Each piece of code will have its own state tracking variable so it knows whats its already done, and what its needs to do next as part of its task. The mechanism of keeping track and controlling transitions between one state and the next is called a state machine.

Super Loop The Waste

All in all the super loop programming methodology can be pretty wasteful:

i) Unnecessary processor clock cycles - With a super loop some tasks lay dormant, but still consume many processor clock cycles just checking that they have nothing to do.

ii) Longer coding time – The state machine code doesn't look after itself, the programmer has to keep track of the state variables and state machine.

iii) Long testing and debug time – In our experience its so easy to make a mistake if changes are made to state machine the chances of code changes working "first time" is substantially reduce.

Benefits of RTOS

Using an RTOS removes the wasteful aspects of super loop programming. Heres how its helps:

i) Better utilization of processor clock cycles – by allowing tasks to "sleep" or stopping a task altogether, their processor usage falls to next to nothing allowing other tasks to have more processor time available.

ii) Shortened coding time – By reducing the code required to keep track of the state of a task, code becomes much easier. The RTOS is keeping track of the tasks state for us. The code also becomes much clearer.

iii) Testing and debug time reduced – By relieving much of the coding of the state machine and state management, the state machine code has a much higher chance of working "first time".

RTOS and a single task

If you have only a single task and never want to add another, then there is no benefit to using an RTOS. In fact it is detrimental because it uses both RAM and ROM that then cannot be used by the application. This is normally the exception. In real applications, normally you want to scan a keypad, flash a light and update a display in a way that appears as though they are all happening at the same time.

Novo composition

Novo Queues

It has four queue types run, sleep and event, pending event. A task is added to the run queue when it can be run. A task is moved from the run queue to the sleep queue when the task is sleeping or when the task is waiting for an event with a timeout. A task is added to the event queue when a task is waiting for an event to occur. When an event occurs the task is moved from the event queue to the pending event queue, and then finally to the run queue.

Task Priorities

Each task has a priority associated with it. The lower the task priority value, the higher the task priority. A task with a priority value of 0 (zero) has the highest priority. Only the highest priority tasks in the run queue are executed when another task yields.

Task and Event Handles

In order to refer to a task or event Novo uses the abstract concept of handles. The reason for using this concept is that allows future changes of the implementation of Novo without greatly affecting existing code.

Novo Events

Novo handles are in the form of semaphores. A semaphore allows once task to signal to another task that something has happened. The benefit of a semaphore over a simple flag that one task sets and another one monitors, is that a task waiting for a semaphore consumes very few processor cycle. The task that is waiting is not continuously polling, the task will be awoken when the semaphore is signaled.

Counting Semaphore

Novo has counting semaphores. Every time a semaphore is signaled, the semaphore count value is increased. The semaphore value is decreased every time a wait for semaphore function is called, before execution returns to the waiting task.

Maximum Number of Priorities

In order to economize on usage of memory, Novo allows a maximum of 15 different priority levels (0 to 14).

Maximum Number of Tasks

In order to economize on usage of memory, Novo allows a maximum of 15 tasks (Task handles in the range 0 to 14).

Hardware and Software Stacks

Novo uses a composite stack, that is it uses a combination of both hardware and software stacks. The use of a software call stack allows Novo to be able to resume a task at from the point at which the task yields. The yielding points can be at any point in the call tree, but not in any function called by more than one task.

Each task has its software stack which is independent of all other tasks.

In order for the software call stack to be enabled, the **-swcs** command line option needs to be used with the **boostlink** linker.

Important notes on usage

Placement of yielding functions

- The placement of yielding functions affect the amount of RAM used. The deeper they appear down the call tree the greater the amount of RAM used. Each level of call stack deeper a yielding function is placed uses an extra byte of RAM.
- Yielding functions must not be placed in functions called by more than one task, unless extreme care is taken to ensure that both tasks cannot yield from within this function at the same time.
  BoostLinker will generate a 'serious warning' if it detects this occurring.

Quantity of yielding functions

Each yielding function call generates a new return point. Each return point is added to a table, consuming more ROM. This table is limited to 255 instructions in length (including any code page switching that maybe required) and is not allowed to cross a page boundary. If the table becomes full no more yielding points can be added. If the table overflows a boundary crossing error will be reported.

So it is recommended that you use a minimum number of yielding function calls.

Task Functions

Tasks start at a task root function. This function is where execution will start when a task is run.

The name of the root function of any task must be prefixed with “Task”.

Novo Task names in BoostC and BoostC++

```
void Task_FlashLight()
void Task0()
void TaskBlinkLed()
```

Novo Task names in BoostBasic

```
sub Task_FlashLight()
sub Task0()
subTaskBlinkLed()
```

The tasks functions need to have this special name prefix because it affects how the tasks are treated in BoostLink linker.

Novo API

All Novo API functions begin with Sys or Sys_. The function that begin with Sys_ (ie have the additional underscore) yield to the task scheduler, thereby causing a task switch.

Specific Note for BoostC/C++ users:

Some of the Novo routines are implemented as true functions and some as macros. In the following description all Novo API functions are treated as though they are implemented as functions to maintain clarity in the arguments and return values.

Specific Note for BoostBasic users:

Some of the routines are implemented as pure macros and some call functions and subroutines with a thin macro wrapper. This simply means that all of the Novo api functions are are invoked with using the ***call*** keyword. In the following description all Novo API functions are treated as though they are implemented as functions to maintain clarity in the arguments and return values.

Novo Base Functions

SysInit

BoostC/C++ Function: void SysInit();
BoostBasic Function: sub SysInit()
Arguments:
Return:
Callable from: main execution thread only.
Description: This function initializes the Novo RTOS system by setting the values of the data structure that hold non-task specific data.
Notes: This function should only be called once in an application and should be called before any other Novo API function is called.

SysCreateTask

BoostC/C++ Function: void SysCreateTask(TASK_HANDLE hTask, BYTE priority, void (*taskFuncPtr)());
BoostBasic Function: sub SysCreateTask(hTask as TASK_HANDLE , priority as BYTE , taskFuncPtr as NovoTask)
Arguments: hTask – handle to the task to start.
priority – the priority of the task (lower value means higher priority).
taskFuncPtr – a pointer to the task function.
Return:
Callable from: main and task execution threads
Description: This function is used to create a task. The creation process initializes the data structure that holds information about the task.
Notes: This function must be called for each task before any other task related function is called.

SysStartTask

BoostC/C++ Function: void SysStartTask(TASK_HANDLE hTask);
BoostBasic Function: sub SysStartTask(hTask as TASK_HANDLE)
Arguments: hTask - handle to a task.
Return:
Callable from: main and task execution threads
Description: SysStartTask causes the a task with the handle specified to join the running queue. Execution of the task may or may not occur at the next scheduler yield, it depends on the relative priorities of other tasks in the queue.
Notes: The scheduler only executes the task(s) in the run queue with the highest priority.

Sys_StopTask

BoostC/C++ Function: void Sys_StopTask(TASK_HANDLE hTask);
BoostBasic Function: sub Sys_StopTask(hTask as TASK_HANDLE)
Arguments: hTask - handle to a task.
Return:
Callable from: main and task execution threads
Description: Sys_StopTask causes a task to be removed from any queues it is in and its status to be set to stopped. Execution of a stopped task can be resumed by calling the SysStartTask function.
Notes:

Sys_Yield

BoostC/C++ Function: void Sys_Yield();
BoostBasic Function: sub Sys_Yield()
Arguments:
Return:
Callable from: main and task execution threads
Description: This function causes a task to yield to the scheduler. Yielding to the scheduler allows the scheduler to perform a number of background operations and allows the scheduler to decide which task to run next.
Notes:

SysGetTaskStatus

BoostC/C++ Function: BYTE SysGetTaskStatus(TASK_HANDLE hTask);
BoostBasic Function: function SysGetTaskStatus(hTask as TASK_HANDLE) as byte
Arguments: hTask - handle to a task.
Return: A byte of data representing the task status.
Callable from: main and task threads
Description: This macro gets the current status of a task. The status consists of a number of bits, which are NOT mutually exclusive, so the task state (TS) values need to be bitwise ANDed with the status value to determined the status.

- TS_IN_RUN_Q – Task is in the run queue.
- TS_IN_WAIT_Q – Task is in the wait queue (waiting for a timeout).
- TS_IN_EVENT_Q – Task is in the event queue (waiting for an event)
- TS_EVENT_TIMEOUT – Waiting for an event terminated because of a timeout.
- TS_NOT_STOPPED – Task is not stopped, ie it is currently active in some way (in one of the queues).

Notes:

Task Priority Functions

SysSetPriority

BoostC/C++ Function: void SysSetPriority(TASK_HANDLE hTask, BYTE priority);
BoostBasic Function: sub SysSetPriority(hTask as TASK_HANDLE, priority as BYTE)
Arguments: hTask – handle to a task.
priority – the required task priority.
Return:
Callable from: main and task execution threads
Description: Changes the priority of a task. This may cause a task to stop/start being executed depending on the relative priority of tasks in the run queue. If the task is in the event in the event queue its waiting priority will be changed
Notes: Only tasks the are in the run queue with the highest priority actually get executed. If multiple task have the highest priority, then they are executed in a round robin fashion.

SysGetPriority

BoostC/C++ Function: BYTE SysGetPriority(TASK_HANDLE hTask);
BoostBasic Function: function SysGetPriority(hTask as TASK_HANDLE) as byte
Arguments:
Return:
Callable from: main and task threads
Description: This function returns the current priority of the specified task.
Notes:

Event Functions

SysSignalSemaphore

BoostC/C++ Function: void SysSignalSemaphore(EVENT_HANDLE hSemaphore);
BoostBasic Function: sub SysSignalSemaphore(hSemaphore as EVENT_HANDLE)
Arguments: hSemaphore – handle to a semaphore.
Return:
Callable from: main and task threads
Description: This function signals a semaphore, that is it increases the semaphore value by 1. As a consequence any task that is waiting on the semaphore will be moved from the wait and event queues to the run queue.
Notes:

SysSignalSemaphoreIsr

BoostC/C++ Function: void SysSignalSemaphoreIsr(EVENT_HANDLE hSemaphore);
BoostBasic Function: sub SysSignalSemaphoreIsr(hSemaphore as EVENT_HANDLE)
Arguments: hSemaphore – handle to a semaphore.
Return:
Callable from: ISR
Description: This function signals a semaphore from within an interrupt service routine (ISR), that is it increases the semaphore value by 1. Because this function is called within an ISR, the event wakeup action iwill not be completed until the next task scheduler yield occurs. This is performed by the semaphore signaling being put in a an ISR command queue. The command queue only has a length of 1, so if more than one event object is signaled before the next scheduler yield occurs, then task wakeup will be missed, leaving the task in the event queue.
Notes:

Sys_WaitSemaphore

BoostC/C++ Function: void Sys_WaitSemaphore(EVENT_HANDLE hSemaphore, TICK_COUNT timeOut);
BoostBasic Function: sub Sys_WaitSemaphore(hSemaphore as EVENT_HANDLE , timeOut as TICK_COUNT)
Arguments: hSemaphore – handle to a semaphore.
timeOut – timeout time in system ticks.
Return:
Callable from: task threads
Description: This macro causes the current value of a counting semaphore to be examined. If the counting semaphore is non-zero then the function decrements the semaphore value and returns immediately. If the counting semaphores value is zero, then the execution does not return to the task until the semaphore is signaled or the timeout time has elapsed.
Notes: If the wait times out before the semaphore is signaled, the task status TS_EVENT_TIMEOUT is set. Test the task status using SysGetTaskStatus() or use the SysWaitTimedOut() function to check for timeouts.

SysTrySemaphore

BoostC/C++ Function: FLAG SysTrySemaphore(EVENT_HANDLE hSemaphore);
BoostBasic Function: function SysTrySemaphore(hSemaphore as EVENT_HANDLE) as FLAG
Arguments: hSemaphore – handle to a semaphore.
Return: Returns true if semaphore value is non-zero.
Callable from: main and task threads
Description: Tests a counting semaphore for a non-zero value. If found to be non-zero it decrements the semaphore and returns a true value. If found to be zero it returns a false value. This allows a semaphore to be tested without a task having to wait for the semaphore.
Notes:

SysReadSemaphore

BoostC/C++ Function: BYTE SysReadSemaphore(EVENT_HANDLE hSemaphore);
BoostBasic Function: function SysReadSemaphore(hSemaphore as EVENT_HANDLE) as byte
Arguments: hSemaphore – handle to a semaphore.
Return: main and task threads
Callable from: Critical Section or ISR if SysSignalSemaphoreIsr is used.
Description: Retrieves the current value of a counting semaphore.
Notes:

SysWaitTimedOut

BoostC/C++ Function: FLAG SysWaitTimedOut(TASK_HANDLE hTask);
BoostBasic Function: function SysWaitTimedOut(hTask as TASK_HANDLE) as FLAG
Arguments: hTask – handle of the task who's timeout bit is to be checked.
Return: flag with a true value if the last task Wait timed out.
Callable from: main and task threads
Description: This function returns a value that indicates whether the last event wait (eg Sys_WaitSemaphore) for a task completed because of a timeout instead of the semaphore being signaled.

This function returns the same as (SysGetTaskStatus(hTask) & TS_EVENT_TIMEOUT) != 0
Notes:

Timer Functions

SysTimerUpdate

BoostC/C++ Function: void SysTimerUpdate();
BoostBasic Function: sub SysTimerUpdate()
Arguments:
Return:
Callable from: main thread or interrupt service routine
Description: This function needs to be called periodically to update the system time value that is used by sleeping and waiting tasks. The rate at which this function is called set the timebase of the system timing functions.

For example if you want Sys_Sleep(100) to result in 100ms sleep, then *SysTimerUpdate* should be called every 1ms. This is normally achieved by by calling the *SysTimerUpdate* function from a period interrupt routine.
Notes:

Sys_Sleep

BoostC/C++ Function: void Sys_Sleep(TICK_COUNT sleepTime);
BoostBasic Function: sub Sys_Sleep(sleepTime as TICK_COUNT)
Arguments: sleepTime – the time to sleep the current task for in system ticks.
Return:
Callable from: task threads
Description: This macro allows a task to be put to sleep for the specified amount of time. The time value is in system ticks. The system tick time is determined by the rate at which *SysTimerUpdate* is called. While a task is sleeping, it usage of processor clock cycles is minimized.
Notes:

SysGetTime

BoostC/C++ Function: TICK_COUNT SysGetTime();
BoostBasic Function: function SysGetTime() as TICK_COUNT
Arguments:
Return:
Callable from: main and task threads
Description: Retrieves the current system tick count.
Notes:

SysGetElapsedTime

BoostC/C++ Function: TICK_COUNT SysGetElapsedTime(TICK_COUNT startTime);
BoostBasic Function: function SysGetElapsedTime(startTime as TICK_COUNT) as TICK_COUNT
Arguments: startTime – start time value,
Return: Returns the elapsed time
Callable from: main and task threads
Description: Computes the time elapsed since the startTime.
Notes: The startTime value is normally generated from a call to SysGetTime().

The result is invalid if the time elapsed is > the maximum value that TICK_COUNT can hold.

Examples:
if the TICK_COUNT data is 2 bytes long (16 bits), the maximum valid elapsed time that can be measure is 65535 ticks.
if the TICK_COUNT data is 1 bytes long (8 bits), the maximum valid elapsed time that can be measure is = 255 ticks.

Critical Sections

SysCriticalSectionBegin

BoostC/C++ Function: void SysCriticalSectionBegin();
BoostBasic Function: sub SysCriticalSectionBegin()
Arguments:
Return:
Callable from: main and task threads
Description:

This macro is used when a critical section of code is entered. By calling this function the section of code can be access Novo data and be sure that nothing outside will be changing the data. This is acheived by disabling interrupts. The critical section is exited by using the complementary macro SysCriticalSectionEnd().

Notes: Typical use would be to examine two semaphore's that are changed by an interrupt service routine, knowing that they would not change part way through execution of this section of code. Some Novo functions internally use critical sections to prevent data corruption by interrupts.

Other Important points:

- Critical sections can be nested.
- Critical sections may increase interrupt latency.
- Critical section macros can be re-written for applications that require minimal interrupt latency, and don't call any Novo functions from within the interrupt service routine. In this case the Novo library would need to be re-built from the source code.

SysCriticalSectionEnd

BoostC/C++ Function: void SysCriticalSectionEnd();
BoostBasic Function: sub SysCriticalSectionEnd()
Arguments:
Return:
Callable from: main and task threads
Description: This macro is used to end a critical code section, it is the complementary function to SysCriticalSectionBegin(). See SysCriticalSectionBegin() for more details.
Notes:

Critical Sections General Notes

These notes are more easily understood with access to Novo source code.

1. **Critical Section Macros can be nested:**
   The critical section macros use a counter so that nesting is possible. This allows user code to call a Novo API function that has critical sections from within a critical section.
2. **Don't use Critical Section Macros in ISR:**
   Code that uses the critical section macros should NOT be called from within an interrupt service routine (ISR), this includes calling any Novo API functions that use critical sections (most). Doing so will cause corruption of the critical section nesting count, therefore resulting in interrupts being potentially enabled in critical sections.

3. **Create New functions or Macro if necessary:**
   To avoid critical section macros in an ISR, it may be necessary to add new functions or macros, or access the scheduler data directly. For example to gain access to the scheduler tick value normally the SysGetTime(); function is called. This could be replaced in an ISR by scheduler.os_tickCnt;
4. **Minimize Interrupt Latency:**
   Novo RTOS has been written to keep the affects on interrupt latency low, with trade offs against code size. Critical section macros by default disable interrupts, so they potentially causing a delay in response (increase latency) to an interrupt. Keep critical sections short to minimize increase interrupt latency further. If no Novo API functions or scheduler data is accessed in an ISR and no user code critical sections are used, then no critical sections are actually necessary. The critical section macros can be rewritten to do nothing, thus interrupt latency will be minimized.

Critical Sections with Prioritized Interrupts Notes

Changes require access to Novo source code to allow custom library builds.

It is common practice to use the high priority interrupt for ultra time critical events, and so use the low priority interrupt to call Novo functions, in which case the notes below are important.

1. **Don't call from Both High and Low Priority interrupt routines:**
   If you are using High and Low priority interrupt routines, then only call Novo API functions (those that are callable from an ISR) in one of the interrupt routines or corruption may result.
2. **Modify Macros when using PIC18 prioritized interrupts:**
   Ensure the routine ISR that calls Novo API routines (or modifies Novo scheduler data) is disabled/enabled in the critical section macro. The supplied macro disables the high priority ISR, so corruption would result if Novo API functions or scheduler data is modified in the low priority ISR. This can be resolved by changing GIE to GIEL in the critical section macros when calling Novo API functions from within a low priority interrupt service routine.

Changing Critical Section Macros

Changing the critical section macros will require a complete rebuild of the Novo Library. It is recommended that you take a complete copy of the Novo source code and modify that when creating a custom build.

Building Custom Novo Libraries

To minimize the use of RAM you may decide to create a Novo library that meets your exact needs. In order to do this you will need to have the Novo source code (which you will have access to if you have an appropriate license – please visit www.sourceboost.com). The source code is made available by running goodies.exe.

Novo has been written using the BoostC compiler so Novo libraries can only be built using the BoostC or BoostC++ compilers.

Important Note

Just changing the header file without re-building a library will potentially cause all sorts of bizarre behavior to occur as the header file will no longer match the library - **so don't do it!**

Files to Create for a Custom Build

1) Project file (.__c) to build the library.

2) Custom configuration header file (.h) that contains the custom options.

3) Source file to build (.c).

The easiest way to create the files is to copy and rename the .c and .h files from an existing library build (that can be found in the novo folder), and create a new project file. Be sure to follow the recommended N*aming convention.*

File recommended file locations:

Custom Novo Build Files	Location
Configuration header file	SourceBoost Installation folder\include
Project	SourceBoost Installation folder\novo
source file	SourceBoost Installation folder\novo

Typical Novo Library project file

File name: novolib_pic18t6e4ts2.c

```
// build novo library with appropriate configuration

#include <novocfg_pic18t6e4ts2.h>
#include "novo.c"
```

Typical Novo Configuration file Contents

File name: novocfg_pic18t6e4ts2.h

```
#define _NOVOCFG_H_

// Maximum number of tasks and events
#define MAX_TASKS 3
#define MAX_EVENTS 5

// Tick count data size
typedef unsigned int TICK_COUNT;

// future - the ability to disable features to save RAM and ROM
// #define _NOVO_EVENTS
// #define _NOVO_TIMERS
// #define _NOVO_PRIORITIES
```

Naming convention

Its is important to stick to a consistent naming convention or the configuration of a build will be unknown, and will then no doubt cause usage issues.

Typical configuration file name: novocfg_pic18t4e4ts1p.h
novocfg - novo config file
pic18 - target
t4 - tasks maximum of 4
e4 - events maximum of 4 (also means events are supported).
ts2 - timer size 2 bytes (also means timers are supported).
p - means priorities are support

Typical library build source file name: novolib_pic18t4e4ts1p.c
novolib - novo lib file
pic18 - target
t4 - tasks maximum of 4
e4 - events maximum of 4 (also means events are supported).
ts2 - timer size 2 bytes (also means timers are supported).
p - means priorities are support

The built library file name will be:
novolib_pic18t4e4ts1p.lib

Build Options

The build options for a build are contained in a header file. This header file is used during the process of building of the library and is included into files that call novo functions.

Novo can be built with different levels of functionality. As more functionality is added there is an increase in the amount of RAM and ROM that is required.

Build options	***Desc***
MAX_TASKS	Numerical that sets the maximum number of tasks
MAX_EVENTS	Numerical that sets the maximum number of events
TICK_COUNT	Data type that determines the size of the data used for Novo sleep and wait timing.

Build options	*Desc*
_NOVO_MINIMAL_RAM	By defining this symbol Novo combines some of the task specific data to reduce the amount of RAM used. Not defining this symbol can be useful for debugging purposes.
_NOVO_EVENTS	By defining this symbol Novo supports Events (ie semaphores).
_NOVO_TIMERS	By defining this symbol Novo supports timer Dependant functions such as *Sys_Sleep()* and *Sys_WaitSemaphore()*.
_NOVO_PRIORITIES	By defining this symbol Novo supports tasks with different priorities.

Building the custom library

Things to check:

1) Check that your .c file includes the appropriate configuration header file.

2) Check that your .c has been added to the project

3) Check that your project is setup to build a library (SourceBoost IDE menu Settings->Options->Compiler Options).

Your library is ready to be built. Initiate a build (press the 'C' button, followed by the 'L' button in SourceBoost IDE). The project should compile and link successfully.

How to use the custom library

In order to use the custom library in a project:

1) Add the custom Novo library to the project (so that it is linked in during a project build). see the SourceBoost IDE and BoostC/BoostC++ manuals on how todo this.

2) Include the custom Novo build configuration header file and novo.h in the source files that call novo functions.

The start of a typical source file would look like this:

```
// BoostC/BoostC++ Code that uses Novo RTOS
#include <system.h>
#include <novocfg_pic16t3e5ts1.h>
#include <novo.h>

#pragma DATA _CONFIG, _CP_OFF & _CCP1_RB0 & _DEBUG_OFF & _WRT_PROTECT_OFF &
_CPD_OFF & _LVP_OFF & _BODEN_OFF & _MCLR_ON & _PWRTE_OFF & _WDT_OFF &
_HS_OSC

void main()
{
....
....
```

```
rem BoostBasic Code that uses Novo RTOS

#include <basic\novocfg_pic16t3e5ts1_h.bas>
#include <novo_h.bas>

#pragma DATA _CONFIG, _CP_OFF & _CCP1_RB0 & _DEBUG_OFF & _WRT_PROTECT_OFF &
_CPD_OFF & _LVP_OFF & _BODEN_OFF & _MCLR_ON & _PWRTE_OFF & _WDT_OFF &
_HS_OSC

sub main()
....
....
```

Semaphores and Priorities Under the hood

Semaphores Queues

Each semaphore has its own queue of tasks waiting for the semaphore. They are ordered according to their priorities. Higher priority semaphores are tasks are at the front of the queue, low priority tasks at the back of the queue.

If tasks have the same priority they appear in the semaphore queue in the order in which they are added, the first one to be added will be ahead of others in the queue.

What happens when a Semaphore is signaled

When a semaphore is signaled the task that is at the front of the semaphore's queue is moved from the semaphores queue (and from the sleeping queue if it has a wait timeout) into the run queue ready for execution. Only one task is moved from the semaphore's queue to the run queue each time a semaphore is signaled. If the semaphore's queue is empty, then no tasks are moved to the run queue when the semaphore is signaled, instead the semaphores value is increase.

Pending Event Queue

In reality when a semaphore is signaled, the task at the front of the semaphore's queue is not directly moved into the run queue. It is first moved into the pending event queue. This queue is used to minimize the amount of time that it takes to signal a semaphore. This is beneficial for semaphores that are signaled in an interrupt service routine. The pending events queue is handled the next time the task yield occurs.

Sharing a resource between threads

Overview

There are many occasions in which you want to share a common resource between two separate tasks. If any task has completely finished with a resource before it yields, no special action is required because naturally the resource is available before another thread can do anything. But if a task is part way through an operation with a shared resource then care has to be taken or another task could start trying to work with that resource.

The answer to this sharing problem is to use a semaphore. A semaphore can be set to signal that a resource is in use, so that another task has a means of checking the availability of the resource.

Example 1 - For BoostC Compiler

This example shares Port B between two tasks which both generate and output sequence on the port when they have control over the port. When the semaphore is signaled it means that Port B is available.

```
//////////////////////////////////////////
// Portb sharing using a semaphore
//////////////////////////////////////////
// Uses Novo RTOS
//
// This has two tasks. They both share PORTB for
// output. A semaphore is used in binary mode to
// ensure that only one task is using PORTB at a
// time.
//
// Target Device: PIC16F88 20MHz
//
// Author: David Hobday
//
// Version History:
// V1.0 - 19/09/2006
// Initial release.

#include <system.h>
#include <novocfg_pic16t3e5ts1.h>
#include <novo.h>

#pragma DATA _CONFIG, _CP_OFF & _CCP1_RB0 & _DEBUG_OFF & _WRT_PROTECT_OFF & _CPD_OFF &
_LVP_OFF & _BODEN_OFF & _MCLR_ON & _PWRTE_OFF & _WDT_OFF & _HS_OSC

#define hTask0 0
#define hTask1 1

#define hSemaPortbAvailable 0

void interrupt( void )
{
      if( intcon.TMR0IF )
      {
            // update system time every 1ms (actually is 204.8us x 5 = 1.024ms)
            static BYTE intDivider;
            if( ++intDivider == 5 )
            {
                  intDivider = 0;
                  SysTimerUpdate();
            }

            intcon.TMR0IF = 0; //clear TMR0 overflow flag
      }
}

void InitTimer()
{
      // configure Timer0
      option_reg.T0CS = 0; // use internal clock
      option_reg.PSA = 0; // use prescaler form timer 0


      // so we get an interrupt around every 204.8us with 20MHz Clock
      // set prescaller to divide by 4


      option_reg.PS0 = 1;
      option_reg.PS1 = 0;
      option_reg.PS2 = 0;

      // enable interrupts
      intcon.TMR0IE = 1; //enable TMR0 overflow bit
      intcon.GIE = 1;
}

void Task0()
{
      while( 1 )
      {
            Sys_WaitSemaphore( hSemaPortbAvailable, EVENT_NO_TIMEOUT );

            portb = 0b00000000;
            Sys_Sleep( 255 );
            portb = 0b10000001;
            Sys_Sleep( 255 );
            portb = 0b01000010;
            Sys_Sleep( 255 );
```

```
            portb = 0b00100100;
            Sys_Sleep( 255 );
            portb = 0b00011000;
            Sys_Sleep( 255 );

            SysSignalSemaphore( hSemaPortbAvailable );
            Sys_Yield();
      }
}

void Task1()
{
      while( 1 )
      {
            Sys_WaitSemaphore( hSemaPortbAvailable, EVENT_NO_TIMEOUT );

            BYTE i;
            for( i = 0; i < 5; i++ )
            {
                  portb = 0b01010101;
                  Sys_Sleep( 255 );
                  portb = 0b10101010;
                  Sys_Sleep( 255 );
            }

            SysSignalSemaphore( hSemaPortbAvailable );
            Sys_Yield();
      }
}

void main()
{
      ansel = 0;
      trisb = 0x00;

      InitTimer();

      SysInit();

      SysCreateTask( hTask0, 2, Task0 );
      SysCreateTask( hTask1, 2, Task1 );

      SysSignalSemaphore( hSemaPortbAvailable ); // allow initial use of portb

      // Task0 will be the first to run as it was started first and
      // both tasks have equal priority. Neither task will run however until
      // we executed the first yielding instruction.
      SysStartTask( hTask0 );
      SysStartTask( hTask1 );

      while( 1 )
      {
            Sys_Yield();
      }
}
```

Example 2 - For BoostC Compiler

The following example has a shared resource (which in this case is the function that uses the software stack) used in three tasks. This occurs because a function that yields (so requiring use of the software stack) is called from more than one task. Two of the tasks wait (effectively go to sleep) by calling the Novo RTOS Sys_WaitSemaphore function, while the other task keeps running and tries the semaphore by calling the Novo RTOS SysTrySemaphore function.

```
// Example of resource sharing between tasks using a common function that yields.
//
// When using the Novo RTOS it is dangerous to call a function that yields in more
// than one task because the software call stack can be corrupted.
//
// This code shows the how a function that yields can be safely called from more
// than one task. It does this by treating the function as a shared resource, using
// a semaphore to signal its availability.
//
// Author: David Hobday
// Date: 12/05/2005
// Target: PIC16
//

#include <novocfg_pic16t3e5ts1.h>
#include <novo.h>

#define hTask0 0
#define hTask1 1
#define hTask2 2

#define hSemPutsResource 0

void puts( unsigned char c )
{
    // This routine doesn't actually do anything other than yield once before
    // loading the txchar variable.
    //
    // The code in this function could be accessing a hardware uart.
    bool busy = 1;
    while( busy )
    {
        Sys_Yield();
        busy = 0;
    }

    unsigned char txchar = c;
}

void Task0()
{
    while( 1 )
    {
        // see if we can get the resource, might never get it as we are not in the queue
        if( SysTrySemaphore( hSemPutsResource ) )
        {
            puts( '0' );
            puts( '0' );
            puts( '0' );
            SysSignalSemaphore( hSemPutsResource ); // release the resource
        }
        // do other continous stuff
        Sys_Yield();
    }
}

void Task1()
{
    while( 1 )
    {
        // wait for then lock the resource
        Sys_WaitSemaphore( hSemPutsResource, EVENT_NO_TIMEOUT );
        puts( '1' );
        puts( '1' );
        puts( '1' );
        SysSignalSemaphore( hSemPutsResource ); // release the resource
        Sys_Sleep( 10 );
    }
}

void Task2()
{
    while( 1 )
    {
        // wait for then lock the resource
        Sys_WaitSemaphore( hSemPutsResource, EVENT_NO_TIMEOUT );
        puts( '2' );
```

```
        puts( '2' );
        puts( '2' );
        SysSignalSemaphore( hSemPutsResource ); // release the resource
        Sys_Sleep( 10 );
    }
}

void main()
{
    SysInit();
    SysCreateTask( hTask0, 2, Task0 );
    SysCreateTask( hTask1, 2, Task1 );
    SysCreateTask( hTask2, 2, Task2 );

    SysStartTask( hTask0 );
    SysStartTask( hTask1 );
    SysStartTask( hTask2 );

    SysSignalSemaphore( hSemPutsResource );

    while( 1 )
    {
        Sys_Yield();
        SysTimerUpdate(); // generate a tick
    }
}
```

General Support

For general support issues, please mail support@sourceboost.com

Always Pleased To Hear From You!

We are always pleased to hear your comments, this helps us to satisfy your needs. Send mail to support@sourceboost.com or post your comments on the SourceBoost Forum.