Task Synchronization

In a multitasking environment, you sometimes need to synchronize the order of operations between two or more tasks. For example, some tasks may need the results of another task before executing.

Semaphores can be used to synchronize tasks. If a semaphore is created in an unavailable state, every task that waits on the semaphore will block until it is signaled. In this case, the dependent tasks must wait on the semaphore, and the data-generating task will signal it when it has finished producing data. That way, the dependent tasks will execute only when the results are ready, thereby correctly synchronizing the order of operations. Assuming the Ssync semaphore is initialized in the unavailable state, the listings below show the pseudocode for synchronizing two tasks together:

```plaintext
Begin Loop:
    wait(Ssync);
    Use results generated by Master Task;
End Loop.
```

Listing - Slave Task

```plaintext
Begin Loop:
    Generate the needed results for the dependent tasks;
    signal(Ssync);
End Loop.
```

Listing - Master Task

A semaphore can also be used to optimize the processing of interrupts. Generally speaking, the code that runs directly in response to an interrupt – the Interrupt Service Routine (ISR) – is executed while all other interrupts are blocked. To reduce the amount of time the interrupts are disabled, and increase the responsiveness of the application, it is often better to process the interrupt in a task rather than in the ISR. One method for doing this is to have a task that waits on an unavailable semaphore. Once the interrupt occurs, the ISR needs only to clear it and then call signal() to notify the task. The task is then made ready to run and will execute as soon as the kernel schedules it. This way, the ISR is kept short, and the application will be responsive.