

Inter-Task Communication

INTRODUCTION

All but the simplest imaginable operating systems have to manage multiple Tasks. Tasks must be able to communicate with each other and with Interrupt Service Routines (ISR). One Task may need to call another, or wait for another to complete and return a result, while an ISR may need to call or suspend a Task. The operating system must manage all of this without errors.

For this purpose, Real Time Operating Systems (RTOS) use a number of possible mechanisms for communicating signals or data between Tasks. It is usually necessary to buffer data so that it can be exchanged between Tasks that are not synchronised. This technical note is to describe these mechanisms and how they may be used.

QUEUES

Queues are needed to communicate complex data or multiple items of data between Tasks.

A Queue is the simplest method of buffering communication between Tasks, or between ISRs and Tasks. It is a list of data items that communicates information between two systems or processes, or between two Tasks or threads in the same program.

For example, If Task 1 and Task 2 needed to communicate, Task 1 may place a data item (represented as Message "A" in Figure 1) into a Queue, from there it may be read and/or removed by Task 2.



Figure 1 A Data item in a Queue

This data item in the Queue may be a copy of a variable, or a pointer to where the data is stored. Task 2 can be blocked waiting for data from the Queue. Optionally, a timeout can be specified to ensure that Task 2 becomes ready again if no data is provided in time. Similarly, blocking can occur on writing to the Queue. Queues may have multiple writers, and (less commonly) multiple readers.

SAFERTOS® AND OPENRTOS® QUEUES

SAFERTOS® and OPENRTOS® use 'Queue by Copy' by default, as this provides a number of advantages. If the programmer wants to "Queue by Reference", then pointers can be queued rather than copies of the data, but care must be taken to ensure that all memory issues are handled.

For safety, SAFERTOS® requires the application to allocate memory for Queues and Semaphores statically.

MUTEXES

A Mutex is a software mechanism for protecting a resource that is available to multiple Tasks.

A Mutex ensures that a resource can only be used by one Task at a time. This is analogous to having one key available to a room, so that only the person in current possession of the key can have access to the room.

A Mutex on a resource is taken by a single Task and must be released before another Task can take the Mutex on the same resource. If several resources need to be protected, then each should have its own Mutex. It is possible for a Mutex to be recursive, that is, one Task may take the Mutex more than once, and has to give it back as many times as it has taken it before the resource becomes available to another Task.

A Mutex, unless carefully programmed, can give rise to a side effect called 'priority inversion' in which a low-priority Task can block a higher one as the result of holding the Mutex. A Mutex should never be called from an ISR, because an interrupt cannot block to wait for a Mutex to become available.

An alternative to using a Mutex is to have a Gatekeeper Task that has sole control of a particular resource. Tasks requiring

the resource must get it from the gatekeeper, which allows access to only one Task at a time.

TASK NOTIFICATIONS

Whenever Task Notifications are possible, they have the least processing and RAM overheads, including from ISRs to Tasks.

Certain RTOSes enable a simpler, direct communication between Tasks that does not involve a communication object such as a Semaphore or Queue.

An ISR or Task can send an Event or data to another Task by setting a value in the receiving Task itself. This is typically faster and uses less RAM than using a communication object. However, it is restricted to relatively simple cases.

In those simple cases, Task Notifications can be used to replace a Binary Semaphore or to implement light-weight alternatives to Counting Semaphores, Event Groups and Mailboxes.

Task Notifications cannot be used to send an Event or data to an ISR, to communicate with multiple Tasks or buffer multiple data items, and in such cases solutions must be found using the other mechanisms.

SAFERTOS® AND OPENRTOS® TASK NOTIFICATIONS

SAFERTOS® and OPENRTOS® implement the Task Notification mechanism. They do this by setting a flag in the Task Control Block of the consumer Task.

ISRs must never call an API function that could cause a Task to become blocked, as the ISR cannot then be guaranteed to become unblocked. For this reason, SAFERTOS® and OPENRTOS® implement special API functions safe to be used from within ISRs.

EVENT GROUPS

Event Groups are an efficient way to communicate groups of single bits, and to respond to combinations of events. If many Tasks are blocked waiting for a single Semaphore, then an Event Flag should provide the most efficient solution.

An Event is an input from the external environment, upon which a Task may need to take some action. An Event Group is a group of bits, each of which can be set to signal that some specific Event has happened.

Employing an Event Group simplifies the process by allowing the programming of a Task to seamlessly respond to various Event combinations through the utilization of a bit mask. It is also possible to use Event Groups to send an Event to more than one Task at the same time.

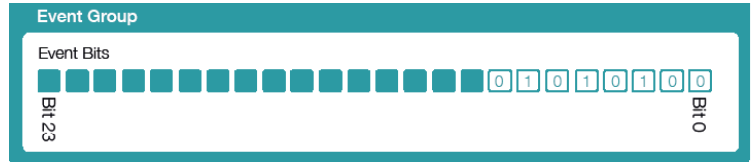


Figure 2 An Event Group with 8 bits

SEMAPHORES

Semaphores are generally more efficient when the interaction is between ISR and Tasks. When a Semaphore is used as a trigger, multiple tasks can unblock on a Semaphore, unlike on a Task Notification.

A Semaphore is a signalling mechanism between Tasks. It is a simple way for one Task to signal to another when the receiving Task does not have to be synchronized with the sender.

A Binary Semaphore is a variable that is set by one Task to indicate some condition to another Task. As the Semaphore has persistence, the Task that reads it does not have to be synchronised with the producer of the Semaphore. It is the responsibility of the consumer Task to reset the Semaphore.

An extension of a Semaphore is the Counting Semaphore, which is a variable that can be set to an integer value. This can be used, for example, to control access to a pool of resources, where the maximum value of the integer is the total number of available resources. As each resource is taken by a Task, it decrements the Semaphore by 1. The Task must increment the Semaphore when it yields the resource. Binary and Counting Semaphores are used in all multitasking systems.

SAFERTOS® implements Semaphores as Queues of minimum size, so that the same API functions can be used, saving on RAM.

WITTENSTEIN high integrity systems

Worldwide Sales and Support
Americas: +1 408 625 4712
ROTW: +44 1275 395 600
Email: sales@highintegritysystems.com
Web: www.highintegritysystems.com



WITTENSTEIN