Message Queues, Mailboxes, and Pipes

Tasks must be able to communicate with one another to coordinate their activities or to share data. For example, in the underground tank monitoring system the task that calculates the amount of gas in the tanks must let other parts of the system know how much gasoline there is. In Telegraph, the system we discussed earlier that connects a serial-port printer to a network, the tasks that receive data on the network must hand that data off to other tasks that pass the data on to the printer or that determine responses to send on the network.

We also discussed using shared data and semaphores to allow tasks to communicate with one another. In this section we will discuss several other methods that most RTOSs other: queues, mailboxes, and pipes.

Here's a very simple example. Suppose that we have two tasks, Task1 and Task2, each of which has a number of high-priority, urgent things to do. Suppose also that from time to time these two tasks discover error conditions that must be reported on a network, a time consuming process. In order not to delay Task1 and Task2, it makes sense to have a separate task, Errors Task that is responsible for reporting the error conditions on the network. Whenever Task1 or Task2 discovers an error, it reports that error to ErrorsTask and then goes on about its own business. The error reporting process undertaken by ErrorsTask does not delay the other tasks. An RTOS queue is the way to implement this design.

Some Ugly Details.

As you've no doubt guessed, queues are not quite simple. Here are some of the complications that you will have to deal with in most RTOSs:

Most RTOSs require that you initialize your queues before you use them, by calling a function provided for this purpose. On some systems, it is also up to you to allocate the memory that the RTOS will manage as a queue. As with semaphores, it makes most sense to initialize queues in some code that is guaranteed to run before any task tries to use them.

Since most RTOSs allow you to have as many queues as you want, you pass an additional parameter to every queue function: the identity of the queue to which you want to write or from which you want to read. Various systems do this in various ways.

If your code tries to write to a queue when the queue is full, the RTOS must either return an error to let you know that the write operation failed (a more common RTOS behavior) or it must block the task until some other task reads data from the queue and thereby creates some space (a less common RTOS behavior). Your code must deal with whichever of these behaviors your RTOS exhibits.

Many RTOSs include a function that will read from a queue if there is any data and will return an error code if not. This function is in addition to the one that will block your task if the queue is empty.

The amount of data that the RTOS lets you write to the queue in one call may not be exactly the amount that you want to write. Many RTOSs are inflexible about this. One common RTOS characteristic is to allow you to write onto a queue in one call the number of bytes taken up by a void pointer.

Mailboxes

In general, mailboxes are much like queues. The typical RTOS has functions to create, to write to, and to read from mailboxes, and perhaps functions to check whether the mailbox contains any messages and to destroy the mailbox if it is no longer needed. The details of mailboxes, however, are different in different RTOSs.

Here are some of the variations that you might see:

Although some RTOSs allow a certain number of messages in each mailbox, a number that you can usually choose when you create the mailbox, others allow only one message in a mailbox at a time. Once one message is written to a mailbox under these systems, the mailbox is full; no other message can be written to the mailbox until the first one is read.

In some RTOSs, the number of messages in each mailbox is unlimited. There is a limit to the total number of messages that can be in all of the mailboxes in the system, but these messages will be distributed into the individual mailboxes as they are needed.

In some RTOSs, you can prioritize mailbox messages. Higher-priority messages will be read before lower-priority messages, regardless of the order in which they are written into the mailbox.

Pipes

Pipes are also much like queues. The RTOS can create them, write to them, read from them, and so on. The details of pipes, however, like the details of mailboxes and queues, vary from RTOS to RTOS. Some variations you might see include the following:

Some RTOSs allow you to write messages of varying lengths onto pipes (unlike mailboxes and queues, in which the message length is typically fixed).

Pipes in some RTOSs are entirely byte-oriented: if Task A writes 11 bytes to the pipe and then Task B writes 19 bytes to the pipe, then if Task C reads 14 bytes from the pipe, it will get the 11 that Task A wrote plus the first 3 that Task B wrote. The other 16 that task B wrote remain in the pipe for whatever task reads from it next.

Some RTOSs use the standard C library functions fread and fwrite to read from and write to pipes.

Which Should I Use?

Since queues, mailboxes, and pipes vary so much from one RTOS to another, it is hard to give much universal guidance about which to use in any given situation. When RTOS vendors design these features, they must make the usual programming trade-off's among flexibility, speed, memory space, the length of time that interrupts must be disabled within the RTOS functions, and so on. Most RTOS vendors describe these characteristics in their documentation; read it to determine which of the communications mechanisms best meets your requirements.

Pitfalls

Although queues, mailboxes, and pipes can make it quite easy to share data among tasks, they can also make it quite easy to insert bugs into your system. Here are a few tried-and-true methods for making yourself some trouble:

Most RTOSs do not restrict which tasks can read from or write to any given queue, mailbox, or pipe. Therefore, you must ensure that tasks use the correct one each time. If some task writes temperature data onto a queue read by a task expecting error codes, your system will not work very well. This is obvious, but it is easy to mess up.

The RTOS cannot ensure that data written onto a queue, mailbox, or pipe will be properly interpreted by the task that reads it. If one task writes an integer onto the queue and another task reads it and then treats it as a pointer, your product will not ship until the problem is found and fixed.

Running out of space in queues, mailboxes, or pipes is usually a disaster for embedded software. When one task needs to pass data to another, it is usually not optional. Good solutions to this problem are scarce. Often, the only workable one is to make your queues, mailboxes, and pipes large enough in the first place.

Passing pointers from one task to another through a queue, mailbox, or pipe is one of several ways to create shared data inadvertently.

Link to lesson 7 -Timer Functions
Link to this part Self Test
Link to this part Assignment
Link to this part Chat

Subpages (1): Week 6

Comments

You do not have permission to add comments.