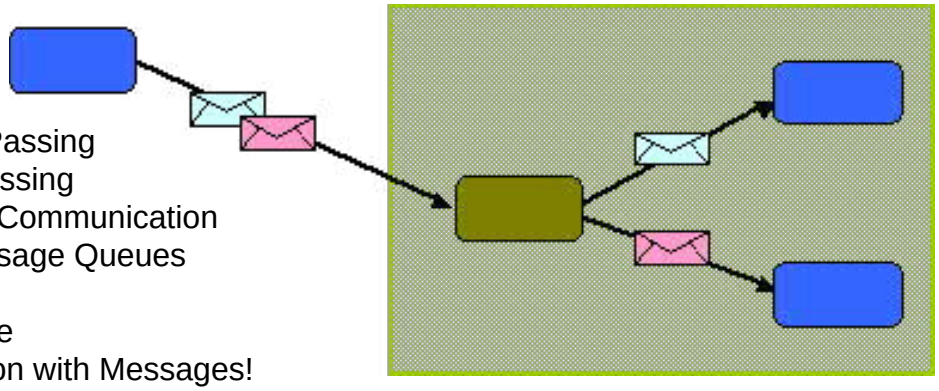
D. Kalinsky Associates

Online Training Session: Messages for Intertask Communication

Messages for Intertask Communication



CONTENTS :

NOTES:

This online training lesson contains material taken from our past classroom-based introductory courses. This particular section presents the basic approaches to message passing for communication between concurrent tasks. In an actual training course, this section would be preceded by introductory sessions covering the fundamental definitions of embedded and real-time systems, as well as the fundamentals of multi-tasking (priority-based pre-emptive task scheduling). This section would be followed by a presentation of the detailed syntax of real-time operating system ("RTOS") service calls for message sending and receiving, then examples of how to write programs that do this in a multi-tasking environment, and then student exercises.

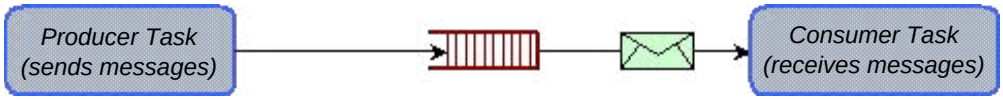
This page lists the issues that will be discussed in this lesson. "Direct" and "Indirect" message passing will be compared and contrasted, as will be "Synchronous" and "Asynchronous" message passing. Message queueing issues will be covered. This will be followed by step-by-step "cookbooks" to guide the programmer in setting up message communications in his/her software.

Message passing is the most popular technique for transferring data between software tasks in a multi-tasking software environment. This lesson will end with a page discussing how message passing can be used even in situations where semaphores have traditionally been used.

This page Updated March 25, 2016

"Indirect" Message Passing

- **Messages are Sent from Task to Task via Queues**
- **Not** Sent Directly from Task to Task
- **Message Queues are Separate RTOS Objects**
- Need to be Created by RTOS Service Calls



- **Caution: Messages are Copied by the RTOS**
  … as they Enter/Exit the Queue
  --> Usually Copied Twice
- **Example: RTOS "X": `msg_send()` copies Message into "Hidden" RTOS Memory**
  `msg_receive()` copies Message out of "Hidden" RTOS Memory

NOTES:

Most real-time operating systems use a technique we'll call "Indirect" Message Passing. In this approach, messages are NOT sent straight from task to task, but rather through an intermediary 'entity' called a "**Message Queue**". *(I'm using the word 'entity' because I don't want to use the word 'object'.)* The idea is that one task will send messages into the queue; and then, perhaps later on, another task will fetch the messages from the queue.

In the diagram on this page, the message queue is shown in the middle, with the message sender task shown on the left and the message receiver task on the right. The envelope in the diagram represents a message, and is shown as it is being fetched from the queue for the receiver task.

Before a task can send a message to another task, the Message Queue needs to be created. This is done by asking that your RTOS create the queue, using an RTOS service call named something like "`queue_create()`". Any task can ask to create the queue; it doesn't have to be the message sender task or the message receiver task. But, both the message sender and message receiver tasks need to be informed of the identity of the queue (this is usually a 32-bit number, called a "Queue-Identifier"), in order for them to communicate through the queue.

Typically, an embedded or real-time application using "Indirect" Message Passing will contain lots of message queues. It is simplest to design software so that there's a separate queue for every pair of tasks that need to transfer data from one to the other. If the data transfer is bi-directional, that pair of tasks will typically need two queues -- one for each direction of data transfer. If different kinds of data are being transferred, separate queues will usually be used for the separate kinds of data.

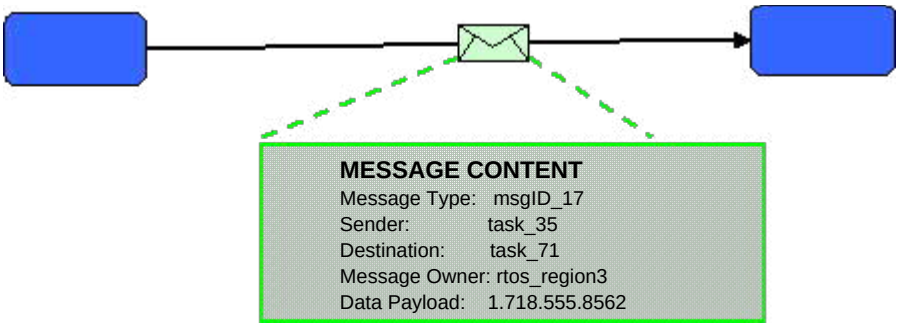
Many real-time operating systems that use "Indirect" Message Passing work by copying the message data into a "hidden" area of RAM memory, when asked by a task to "send" a message. You can think of this "hidden" area of RAM as the actual "queue" -- even though it isn't organized in the way we'd expect a queue to be organized. When a task wants to "receive" a message, the RTOS copies the message data out of the hidden area of RAM and delivers it to the requesting task. This "hiding" of the message while it's in the queue, is done so that no task can change or read the message data while it's enqueued -- thus preventing nasty access collisions that might cause data corruptions ("mutual exclusion violations"). But there's also a 'down-side': Copying the message data into and out of this "hidden" area can be a lot of RTOS performance overhead, especially for long messages.

How can a programmer make message passing happen? A task that has prepared a message for sending, can make an RTOS service call named something like "`msg_send()`". A task that wants to receive a message, can make an RTOS service call named something like "`msg_receive()`". The precise names of the service calls and their parameters, are slightly different for each specific RTOS.

This page Updated January 1, 2010

"Direct" Message Passing

- **Messages themselves are High Performance, Separate RTOS Objects**
- **Sent Directly from Task to Task**
- **No Intermediary Queue Object !**
- **Synchronous vs. Asynchronous Direct Message Passing**



NOTES:

A growing number of real-time operating systems use a different data-passing technique we'll call "Direct" Message Passing. In this approach, messages are sent **directly** from task to task, without any intermediary like a message queue.

In the diagram on this page, the message sender task shown on the left communicates directly with the message receiver task on the right. There is no "independent" message queue between the two tasks. The envelope in the diagram represents a message, and is shown on its way directly from the sender to the receiver task.

There is no need to create a Message Queue when using direct message passing. However, the message sender task needs to know the identity of the receiver task (this is usually a 32-bit number, called a "Task-Identifier"), in order to send a message to that task.

Some real-time operating systems that use "Direct" Message Passing avoid the copying of message data (via a "hidden" area of RAM memory), that can cause performance problems in many "indirect" message passing RTOSs. They do this by delivering to the receiver task a pointer to the buffer containing the message. For long messages, this is much more efficient than delivering a copy of the message itself. In order to prevent "mutual exclusion violations" (*see previous page*), the RTOS will obliterate the buffer pointer that is being stored in the sender task, before delivering a copy of the pointer to the receiver task.

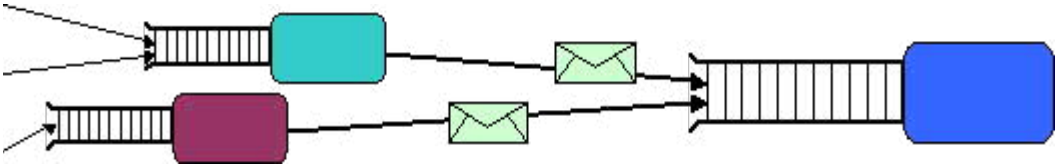
This technique of message passing via pointer passing is highly efficient. And it is equally efficient for both short and

long messages. Hence, messages can be lengthened without slowing down their delivery. This idea is used by some direct message passing RTOSs in order to add some extra bytes of 'administrative data' to the messages they carry. These RTOSs can then use these extra data for quality control and reliability management. Some typical administrative data are shown in the 'blow-up' of the message in the diagram on this page.

Some RTOSs offer **synchronous** direct message passing, while others offer **asynchronous** direct message passing. In synchronous message passing, the message sender task must wait for the receiver task to actually receive the message before the sender task will continue to run. This could delay a message sender task if, for example, the receiver task is busy doing something else and is not attempting to fetch the new message. In asynchronous message passing, the message sender task need not wait for the receiver task to do anything. After sending a message to the receiver task, the sender task can continue to run irrespective of the situation of the receiver task. Both of these synchronization approaches have their advantages and disadvantages, depending on your specific software design.

How can a programmer make **direct** message passing happen? It's quite similar to the programming of indirect message passing. The main difference is that a message sender task will provide the receiver's Task-Identifier (instead of a Queue-Identifier), when making a "`message_send()`"-style service call.

Asynchronous Communication



- **Loosely-coupled operation: Sender "Shoot and Forget"**
- **Task delivers Messages to RTOS, then Continues Execution**
- Never Waits "Non-Blocking"
- Message normally deposited at the Tail of the Receiver's Queue
- "Hopes" that Receiver Task will (eventually) Fetch the Message

- **Content/Structure of Message Payload are irrelevant to RTOS**
- Variable Data, Pointers, Structures, Unions, …

NOTES:

The majority of off-the-shelf RTOSs, whether using direct or indirect message passing, offer message passing which is asynchronous in its behavior: The message sender task does not wait (or "block") for the receiver task to receive the message, or do anything else. After sending its message, the sender task can continue to run irrespective of the situation of the receiver task. [In the Western USA, this is often called "Shoot and Forget".]

In **in**direct message passing RTOSs, the message sender task is allowed to continue to run as soon as its message has been deposited into the **queue** from which the message will eventually (hopefully) be fetched by the receiver task. It may be some time (if ever) before the message will be fetched, but the sender task is not waiting for this to occur.

In **direct** message passing RTOSs, the message sender task is also allowed to continue to run as soon as it gives its message to the RTOS. But what would happen if the intended receiver task is not ready to immediately receive the message? Answer: The RTOS will automatically create a message queue for the intended receiver task, and will hold messages in this queue until the task asks for them. The queue is created without any programmer involvement. In fact, these RTOSs don't even need a "`create_queue()`" service call in their repertoire of services. This kind of automatic creation of message queues only happens in asynchronous direct message passing RTOSs.

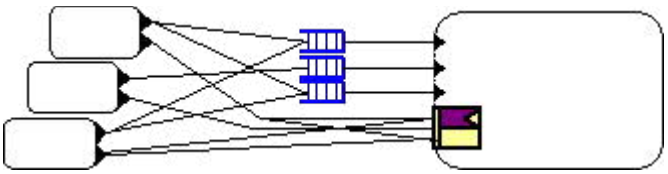
Please note that when a message sender task is allowed to continue execution by an asynchronous message passing RTOS, this does not necessarily mean that the message has been delivered to the intended receiver task (or any other task, for that matter). In many instances, it only means that the message has been stored by the RTOS for

later delivery. The actual time of delivery depends on the behavior of the task that fetches the message from RTOS storage. And if no task ever asks to fetch the message, the message may never be delivered. So there's no guarantee of delivery in asynchronous message passing RTOSs. That's what's meant by the phrase "Hopes that Receiver Task will (eventually) Fetch the Message". [In fact, in **in**direct message passing RTOSs, the wrong task may eventually come around and fetch the message you intended for a totally different task.]

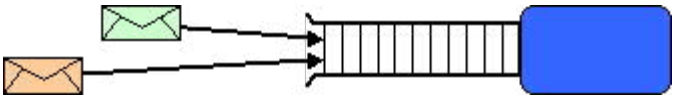
Final point on this page: RTOSs do not check or manipulate the application "payload" content of the messages your tasks send and receive. So you can in fact structure the content of your messages in any way you please. You can even do dangerous things like sending pointers to pointers to pointers, or esoteric things like sending unions of unions of unions.

This page Updated January 1, 2010

Tasks and their Message Queues



- **Using "Indirect" Message Passing RTOSs, there can be a Many-to-Many Relationship between Tasks and Queues**
  - Complex Application Software
    - Task can Wait for Messages on Only 1 Queue at a time
- **Most Applications Restrict Design to one Queue per Task**
- Conceptually like "Direct" Message Passing



NOTES:

As we've seen, Message Queues are used as the conduits through which messages can be passed between software tasks. In some cases, the RTOS will create the queues automatically (asynchronous direct message passing). In other cases, you will need to make service calls to the RTOS to create your queues (indirect message passing). [Only in the case of synchronous direct message passing can the need for queues be avoided.]

Applications using **in**direct message passing will contain lots of message queues. You may want to set up a separate queue for every pair of tasks that need to transfer data from one to the other. If data transfer is bi-directional, that pair of tasks may well need two queues -- one for each direction of data transfer. If different kinds of data are being transferred, separate queues are often used for the separate kinds of data. In this way, you'll end up with lots and lots of queues -- probably many more queues than you've got tasks in your software system. And the result may well be quite a complex many-to-many relationship between your tasks and your queues.

This situation may be made even more complicated by restrictions in your particular RTOS, as seen in the diagram at the top of this page. It shows a portion of a software design using an RTOS which allows a task to wait for messages on only one queue at a time. But the task shown on the right has a requirement to fetch a message from any of 3 queues, from whichever queue has a message first. This sounds impossible, based on the restriction in the RTOS. And it can best be solved using a complex work-around (involving event flags) which makes the design diagram look like a bowl of spaghetti.

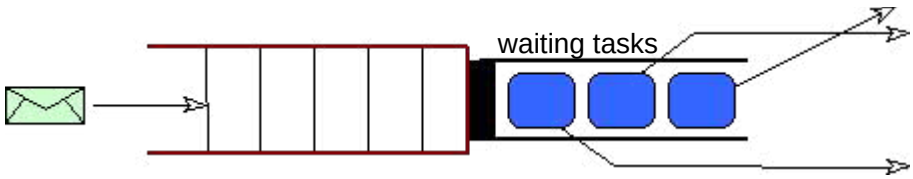
How can we avoid the complexities of designing inter-task message communications when using an **in**direct

message passing RTOS? Answer: Use the same "Design Rule" that's built-in to asynchronous direct message passing RTOSs: Set up no more than **one message queue for each task** that will be receiving messages. [If different messages to this task will contain different kinds of payload content, have the first field of the message data contain a payload type identification number.]

This page Updated January 1, 2010

How Can Tasks Wait for Messages?

- **"Direct" Message Passing RTOSs:**
- **Task Waits in the "Blocked" ("Waiting") State until Message Arrives**
  - Message is Addressed Specifically to this Task
- **"Indirect" Message Passing RTOSs:**
- **Task Waits in the "Blocked" State AND ALSO in a Task Waiting Queue**
  - FIFO or Priority, Time-Limited or Unlimited
- **Each Message Queue has an associated Task Waiting Queue**
  - At least one of these 2 interlinked Queues is empty at any time



NOTES:

We usually think about message queues in the situation where there are a number of messages waiting to be fetched, but no task has yet asked to fetch them. But sometimes, just the reverse happens: There may be no messages waiting to be fetched, but one or more tasks is asking to fetch them. How will a real-time operating system handle this situation? The answer is different, for different categories of real-time operating systems:

Direct message passing RTOSs answer this situation in a very simple way. Every task has its own (automatically created) message queue. So if there are no messages waiting in that queue, the task that asked to fetch its message will be made to wait until its message arrives. The task is then said to be in the "Waiting" or "Blocked" or "Pended" state. [The term used depends on the specific RTOS.] When a message arrives, the task will be given the message and allowed to run with it, .

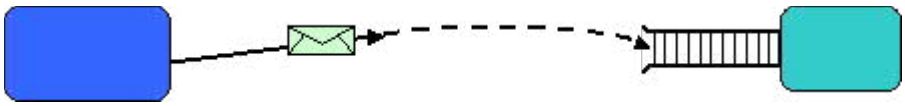
Indirect message passing RTOSs handle this question in a more complex way. This is because many different tasks may ask to fetch messages from the same message queue. For these RTOSs, each queue is actually constructed of two queues, as shown in the diagram on this page: When we think of a queue, we normally think of the queue for messages that's shown on the left side of this diagram. Messages wait in this queue when no task has asked to fetch them yet. But these RTOSs automatically create a second queue for tasks that's shown on the right side of this diagram. Both queues are created together, every time the "`create_queue()`" RTOS service is called. The queue at the right is used to queue up tasks that are waiting for messages that haven't yet arrived. It's convenient to think of these two interrelated queues as "butting head-to-head", or "conjoined".

The tasks in the queue at the right are in a waiting state, and at the same time are queued up in preparation for getting a message in the future. They queue up in the order that they'll get their next message --- when future messages arrive on the left. The order of the queueing of tasks can be either FIFO ordering or Task-Priority ordering. Whenever a task is about to enter this queue (as part of the "`msg_receive()`" RTOS service), it can specify one of 3 choices:
(a) I'm OK to wait in this queue for as long as it takes, until I get a message. ["WAIT_FOREVER"];
(b) I'm OK to wait in this queue for a limited time only. At the end of this time, let me continue to run -- even if I didn't get a message ["WAIT_LIMIT"]; or
(c) I don't want to wait in this queue at all. If there's no message ready for me right now, then let me continue to run immediately. I'm OK to continue running right now without getting any message. ["NOWAIT"]

If you think about it, you'll realize that at least one of these 2 "conjioned" queues will be empty at any given time. Or maybe both will be empty. If there are messages waiting in the message queue at the left, then tasks will fetch them for processing and won't need to wait in the task-waiting queue on the right. On the other hand, if there are tasks waiting in the queue at the right, then messages arriving on the left will be taken for processing immediately by tasks -- and they won't need to wait in the message queue on the left. So both queues in a "conjoined" pair will never be populated at the same time.

This page Updated January 1, 2010

How does a Task Send a Message?



1. **Define Content and Structure of the Message**
   - Make Sure Sender and Receiver Tasks have Same Definition
2. **Allocate a Buffer for Message of appropriate size**
3. **Fill in the Data Payload of the Message**
4. **Send the Message**
   - "Indirect" Message Passing RTOSs: Send to a predefined Queue
   - "Direct" Message Passing RTOSs: Send to Receiver Task
5. **Make Sure the Buffer is eventually Freed**
   - "Indirect" Message Passing RTOSs: Sender Task frees buffer
   - "Direct" Message Passing RTOSs: Receiver Task frees buffer

NOTES:

This is a 5-step "cookbook" for what to do to program a task to send a message. The details are slightly different for different off-the-shelf RTOSs and RTOS categories.

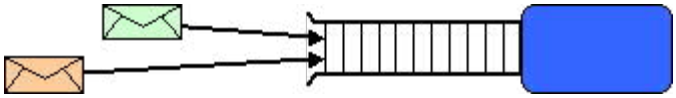
FOR INDIRECT MESSAGE PASSING RTOSs

1. Define Content and Structure of the Message
   - Make Sure Sender and Receiver Tasks have Same Definition
2. Allocate a Buffer for Message of appropriate size
3. Fill in the Data Payload of the Message
4. Send the Message to a predefined Queue
5. Make Sure the Buffer is eventually Freed by the Sender Task

FOR DIRECT MESSAGE PASSING RTOSs

1. Define Content and Structure of the Message
   - Make Sure Sender and Receiver Tasks have Same Definition
2. Allocate a Buffer for Message of appropriate size
3. Fill in the Data Payload of the Message
4. Send the Message to the Receiver Task
5. Make Sure the Buffer is eventually Freed by the **Receiver** Task
   - Sender Task does **not** free buffer (when message is passed by pointer)

How does a Task Receive a Message?



1. Define Content and Structure of the Message
- Make Sure Sender and Receiver Tasks have Same Definition

2. Ask to Receive a Message
- Task may need to Wait (in “Blocked” state) for Message to Arrive

3. If / When Message is Received, Process Appropriately

4. Free the Message Buffer
- … if the Message was in RTOS-allocated Memory

5. (Optionally) Loop back and Ask for Another Message

NOTES:

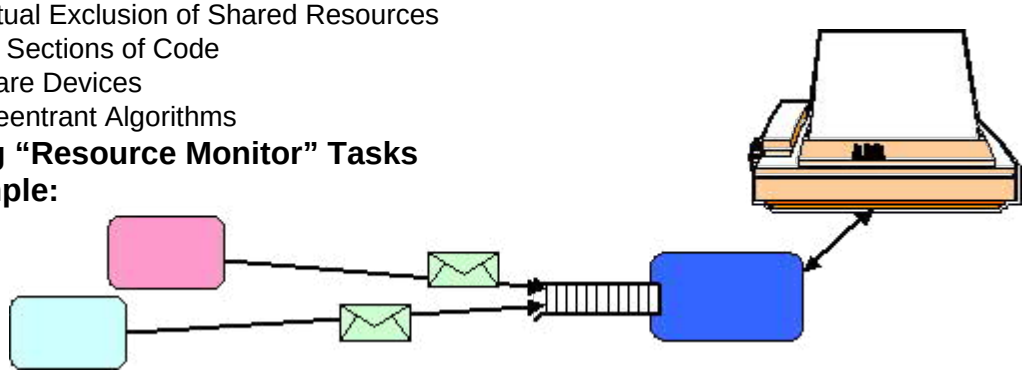
This is a 5-step "cookbook" for what to do to program a task to receive a message. The details are slightly different for different off-the-shelf RTOSs and RTOS categories.

1. Define Content and Structure of the Message
- Make Sure Sender and Receiver Tasks have Same Definition

2. Ask to Receive a Message, taking into account that the task may need to Wait (in “Blocked” or "Waiting" or "Pended" state) for a Message to Arrive
3. If / When Message is Received, Process Appropriately
4. Free the Message Buffer … if the Message was in RTOS-allocated Memory
5. (Optionally) Loop back and Ask for Another Message

Do All Communication With Messages!

- **Define Content and Structure of the Message**
- **Transfer Data between Tasks**
- **Report Events to Tasks**
- **Messages work well in Distributed Multiprocessor Environments**
- Do Mutual Exclusion of Shared Resources
- Critical Sections of Code
- Hardware Devices
- Non-Reentrant Algorithms
- **Using "Resource Monitor" Tasks**
- **Example:**



NOTES:

In conclusion, message passing is a highly popular way of passing data reliably from task to task in embedded and real-time software. It is supported, with slight variants, in most real-time operating systems. On this page, we will discuss the use of message passing beyond its traditional usage in passing data.

Messages can also be used to notify tasks of the occurence of events. They can replace the bug-prone event notification systems of some RTOSs, and they can provide an event-notification mechanism if your RTOS does not have support for events. A message with a single field of payload data is sufficient to notify a receiving task of the occurence of an event -- if you encode an event identifier in the payload data field. Unlike RTOS-supported event mechanisms, messages can be counted reliably. Messages also offer the possibility of carrying large amounts of data together with an event notification, all in the same message.

Messages can also be used as a "mutual exclusion" mechanism -- to ensure that 2 or more tasks do not access a "critical resource" at the same time. This can be seen in the diagram on this page. Here we see two tasks on the left, that would like to share the printer at the far right. When one task is printing, we don't want the other task to join in and print at the same time -- because that would trash the printer output.

A traditional solution for this problem would be to set up a (binary) semaphore between the 2 tasks on the left. Each task would be required to "lock" the semaphore for itself before starting to print; and would "unlock" the semaphore when it finished printing. If a task would attempt to lock an already-locked semaphore, it would be made to wait until the semaphore became unlocked -- thus ensuring mutual exclusion or exclusive access. This solution works, with

only small problems.

An alternative solution for this design would be to tell both tasks on the left that they're forbidden to access the printer at all. Instead, if they want something printed they need to send a print request message to the task shown toward the right side of the diagram (in blue). This kind of task is sometimes called a "Resource Monitor Task", since its job is to hide the detailed workings of the printer while at the same time performing services at the printer at the request of other tasks. Mutual exclusion and exclusive access are taken care of, since the "Resource Monitor Task" handles one printing request (message) at a time, even if a substantial queue of messages has developed. So one print job at a time will be printed, without interference from other print jobs.

Which of these 2 solutions, the semaphore solution or the messaging solution, is more general? Answer: The semaphore solution is good in single-processor and/or single memory-partition environments. But the messaging solution works equally well in those environments and in multi-partitioned memory environments, as well as in multi-CPU multi-core and fully distributed environments. For example, the design shown in the diagram on this page works well in a single-processor environment. And it works equally well if the 3 tasks shown are running on 3 different processors that are linked by message communication.