Technote 1071

Working with Multiprocessing Services

CONTENTS

This Technote discusses some techniques for use with Apple's Multiprocessing Services Library. Methods for sharing information between tasks are discussed and several examples are provided that show how to implement the techniques discussed.

This Technote is primarily directed at developers interested in using Apple's Multiprocessing Services routines.

Overview

Multiprocessing Services provides a set of routines that allow an application to create separate threads of execution called preemptive tasks. Preemptive tasks run simultaneously with the rest of the operating system and are given processor time based on an interrupt-driven scheduling algorithm. Unlike thread manager tasks, the execution of preemptive tasks does not rely on other tasks explicitly yielding processor time by calling either the Event Manager routine `WaitNextEvent` or the Thread Manager routine `YieldToAnyThread`.

Tasks are preemptively scheduled using whatever processors are available to the system. It is not necessary for a machine to be equipped with more than one processor for an application to take advantage of the preemptive process scheduling facilities provided by Multiprocessing Services. Even if a machine is only equipped with one processor, it is possible for an application to schedule and run many simultaneous preemptive tasks.

Multiprocessing Services provides facilities for creating and scheduling tasks along with routines for communicating between tasks. Although access to operating system resources is limited from preemptive tasks, at the time of this writing, it is possible for tasks to allocate memory, make synchronous file manager calls, call deferred tasks, and make remote calls the operating system.

Back to top

Access to Multiprocessing Services

For your application to use Multiprocessing Services, your application must be linked with either CarbonLib or the Multiprocessing Services shared library, "MPLibrary". For best results, when linking with "MPLibrary", your application should use weak links to the Multiprocessing Services routines and then use the `MPLibraryIsLoaded` function call to determine if the library is available for your application to use. If `MPLibraryIsLoaded` returns true, then your application can use the Multiprocessing Services routines and perform preemptive multitasking operations. Otherwise, when `MPLibraryIsLoaded` returns false, your application should use single threaded processing techniques.

All applications using the Multiprocessing Services routines should call the `MPLibraryIsLoaded` routine to determine if Multiprocessing Services is available. This is for two reasons:

1. `MPLibraryIsLoaded` may perform some initialization operations that must be done before other Multiprocessing Services calls can be made.

2. Although CarbonLib exports the symbols required to link your application with Multiprocessing Services, that does not necessarily imply that those routines are available in the context where CarbonLib is running. The call to `MPLibraryIsLoaded` will tell your application if those routines are available for your application to use.

NOTE: Developers linking with CarbonLib who would like to call the routine `DTInstall` from a preemptive task, should make sure that CarbonLib 1.0.2 or later is installed at runtime.

Back to top

Preemptive Tasks

Preemptive tasks are single parameter routines that return a result of type `OSStatus`. Once a task has been created it will run preemptively until it returns a result or until it is explicitly terminated. Tasks are free to perform any type of processing operations they require, however do not have access to the 680x0 emulator. Therefore, it is not possible to place calls to operating system routines that may make use of the 680x0 emulator. A listing of specific Operating system routines that can be called by tasks can be found in the document Adding Multitasking Capability to Applications Using Multiprocessing Services.

Listing 1 shows a simple task that creates a SimpleText file containing 1000 lines of text containing the string "Hello World\n". In this task, a number of the "safe" file manager calls are used to create a file, open its data fork, and write a bunch of strings to the file. When called, this task will run in the background (during mouse clicks, menu selections, et cetera) until it completes. During that time, the application that created this task (and all other applications) will be free to perform any other processing operations it desires.

```
OSStatus ExampleTask( void *parameter ) {
    FSSpec *theFile;
    short refnum;
    Boolean created;
    long i;

         /* the parameter is a FSSpec pointer */
    theFile = (FSSpec *) parameter;

        /* set up locals */
    refnum = 0;
    created = false;

        /* create a file */
    err = FSpCreate(theFile, 'ttxt', 'TEXT', smSystemScript);
    if (err != noErr) goto bail;
    created = true;

        /* open the file for writing */
    err =  FSpOpenDF(theFile, fsRdWrPerm, &refnum);
    if (err != noErr) goto bail;

        /* write out the string 1000 times*/
    for (i=0; i<1000; i++) {
        err = FSWrite(
refnum, (count = 12, &count), "Hello World\n");
        if (err != noErr) goto bail;
    }

        /* close the file and leave */
    FSClose(r
efnum);
    return noErr;

bail:
    if (refnum != 0) FSClose(r
efnum);
    if (created) FSpDelete(theFile);
    return err;
}
```

Listing 1. A task that creates a SimpleText file containing the string "Hello World\n".

The single parameter passed to a task is provided by the caller when the task is created using the `MPCreateTask` routine. As shown in listing 1 and 2, oftentimes the parameter passed to a task will be a pointer to a structure containing information relevant to the operations the task has been designed to perform. In Listing 1, the task assumes that the parameter is a pointer to a `FSSpec` record that refers to a file the task should create. The code snippet shown in Listing 2 illustrates how to pass a pointer to a `FSSpec` record to the task when calling the `MPCreateTask` routine.

```
OSStatus err;
FSSpec targetFile;
MPTaskID taskID;

    /* make a file spec for the target file */
err = FSMakeFSSpec
(0, 0, "\pExample File", &targetFile);

    /* if the file does not exist, call the task to create it */
if (err == fnfErr) {

        /* create the task */
    err = MPCreateTask( ExampleTask,
        &targetFile, /* the parameter passed to the task */
        0, /* use the default stack size - 4K */
        0, /* no notification queue */
        NULL, NULL, /* result parameters - unused */
        0, /* no special task flags */
        &taskID );
}
```

Listing 2. A small sequence of statements that calls the ExampleTask from Listing 1.

Of course, callers will want to know the result codes returned by the tasks they create. To allow for this, Multiprocessing Services provides a mechanism where the result code returned by the task can be passed back to the caller. However, this cannot be done directly as often the time required for a task to execute cannot be determined beforehand. So, to allow Multiprocessing Services to pass back the result returned by the task to its caller, it is possible to designate a queue, by providing it as a parameter to the `MPCreateTask` call, that will be used for communicating the task's result back to the caller. Figure 3 illustrates how one would set up such a queue and provide it as a parameter to `MPCreateTask`.

```
OSStatus err;
FSSpec targetFile;
MPTaskID taskID;
MPQueueID taskQueue;

    /* create a queue for the task to communicate
    results back to the caller */
err = MPCreateQueue( &taskQueue );
if (err != noErr) goto your_error_handler;

    /* make a file spec for the target file */
err = FSMakeFSSpec
(0, 0, "\pExample File", &targetFile);

    /* if the file does not exist, call the task to create it */
if (err == fnfErr) {

        /* create the task */
    err = MPCreateTask( ExampleTask,
        &targetFile, /* the parameter passed to the task */
        0, /* use the default stack size - 4K */
        taskQueue, /* the task's notification queue */
        NULL, NULL, /* first 64 bits of result */
        0, /* no special task flags */
        &taskID );

}
```

Listing 3. A sequence of statements that calls the Example task shown in Listing 1. This listing illustrates how to provide a task notification queue when a task is created so the result code returned by the task can be discovered after the task has completed.

If a task notification queue is provided as a parameter to the `MPCreateTask` routine, then this queue will be used to communicate any result codes returned by the task to its caller. As shown in Listing 4, the task's result code is returned in the third `MPWaitOnQueue` result parameter.

```
    Boolean complete;
    OSStatus err;

    complete = (MPWaitOnQueue(taskQueue,
        NULL, NULL, /* first 64 bits of result from MPCreateTask */
        (void**) &err, /* the result code returned by the task */
        kDurationImmediate) == noErr);

    if (complete) {
        /* the task is complete and has returned the error code
        that has been copied into err... */
```

Listing 4. Using a task's notification queue to find the result code returned by the task.

Task notification queues are necessary in some cases. For instance, calling `MPTerminateTask` does not immediately destroy a task, nor does it stop the task from executing. Instead, `MPTerminateTask` will schedule the task for termination. The actual operations involved in terminating the task will happen as soon as the task scheduler is able to remove the task from the active task queue. As such, a task may remain executing for some time after `MPTerminateTask` has been called. But, once the task has been stopped and disposed of, Multiprocessing Services will notify the caller of this fact by placing a result in the task's notification queue. It will not be safe for the caller to assume the task is not running until this result arrives.

In general, `MPTerminateTask` should be avoided and only used in exceptional circumstances. Well-written tasks and Multiprocessing Services clients should never need to use this routine. However, in unusual circumstances where it is necessary to terminate a task by calling `MPTerminateTask`, you must use a task notification queue to determine when the task has actually terminated.

Back to top

Intertask Communications

Multiprocessing services provides a number of facilities that can be used for communication between tasks. These facilities and how they can be used are discussed in this section. Whenever possible, applications should use these methods for communication between tasks. Other methods, such as polling global variables, are inefficient and often lead to difficult-to-track-down bugs. The methods discussed in this section are fast, efficient, and they provide a well-defined set of operations for passing messages or communicating state information between tasks.

Queues

Queues are first-in-first-out message buffers designed for passing 96-bit messages between tasks. Each message is formatted as a group of three 32-bit integers. The format of the data passed between tasks is entirely up to the programmer. Here, the only requirement is that all tasks accessing the same queue agree on the format of the data being stored in the queue.

Inserting and extracting elements is an atomic operation - many tasks can try to extract the next message from a given queue, but only one will successfully obtain it.

Listing 5 illustrates how a queue can be used to pass commands to a server task for background processing. Here, the server task extracts messages from a queue and then it performs processing operations based on a dispatching mechanism.

```
enum {
    kRunLongComplexTask,
    kQuickTask,
    kShutDown
};

OSStatus ExampleServerTask( void *parameter ) {
    MPQueueID commandQueue;
    Boolean processingCommands;
    long theCommand, param1, param2;

         /* task parameter is the command queue ID */
    commandQueue = (MPQueueID) parameter;

        /* set up locals */
    processingCommands = true;
    err = noErr;

        /* process commands */
    while (processingCommands) {

            /* get the next command from the queue */
        err = MPWaitOnQueue(commandQueue,
            (void**) &theCommand, /* the first parameter is the command */
            (void**) &param1, /* the next two parameters are arguments.. */
            (void**) &param2,
            kDurationForever);
        if (err != noErr) break;

            /* process the command */
        switch (theCommand) {

            case kRunLongComplexTask:
                PerformSomeComplexAction(param1, param2);
                break;

            case kQuickTask:
                PerformSomeSimpleAction(param1, param2);
                break;

            case kShutDown:
                processingCommands = false;
                break;
        }
    }
        /* release the command queue */
    MPDeleteQueue(commandQueue);

        /* any result codes will be returned in the task's
        notification queue. See listing 4 for an example
        showing how to retrieve this result code. */
    return err;
}
```

Listing 5. A sample server task receiving commands by way of a queue.

The server task shown in listing 5 assumes that messages placed in the queue have a particular format. Specifically, the first 32-bit integer is a command selector and the next two 32-bit integers are additional parameters that may or may not be used in command processing. As the server task assumes this will be the format for all messages placed in the queue, it is useful to have a single routine that formats queue entries according to this agreed upon format when sending messages to the server task. The routine shown in Listing 6 provides this mechanism.

```
MPQueueID gServerCommandQueue;

OSStatus SendCommandToServerTask(long theCommand, long param1, long param2) {
    return MPNotifyQueue(gServerCommandQueue,
        (void*) theCommand, /* the first parameter is the command */
        (void*) param1, /* the next two parameters are arguments... */
        (void*) param2);
}
```

Listing 6. A routine for sending commands to the server task shown in listing 5.

Often it is best to "wrap" the routine used to place commands in a server task's command queue in this manner rather than calling `MPNotifyQueue` directly to send commands to the server task. Adding the additional layer of abstraction reduces code maintenance requirements should the format of the messages in the queue change.

Event Groups

Event groups are the fastest method available for communications between tasks. An event group is a 32-bit integer. Each bit in that integer is used to represent an individual event. Event groups can be used as a mechanism for sending simple boolean messages to tasks. When used in this way, each bit in the event group represents a message.

Unlike queues, where messages are received one at a time, it is possible that several "event" messages may be received simultaneously in one call to `MPWaitForEvent`. In other words, several events may accumulate in an event group between calls to `MPWaitForEvent`. As a result, code responsible for decoding and responding to events returned by `MPWaitForEvent` must be aware of the fact that more than one event may be returned by any given call. The event handler shown in listing 7 illustrates an appropriate way to handle event groups returned by `MPWaitForEvent`.

```
/* each event is defined as a single bit in the event group. */
enum {
    kDanglingPointerDetected = (1<<0),
    kUnlockedHandleSighted = (1<<1),
    kResEditDetected = (1<<2),
    kSelfDestruct = (1<<3)
};

OSStatus ExampleEventHandlerTask( void *parameter ) {
    MPEventID eventGroup;
    Boolean processingEvents;
    MPEventFlags theFlags;

         /* task parameter is an event group */
    eventGroup = (MPEventID) parameter;

        /* set up locals */
    processingEvents = true;
    err = noErr;

        /* process events */
    while (processingEvents) {

            /* get the next event */
        err = MPWaitForEvent(eventGroup, &theFlags, kDurationForever);
        if (err != noErr) break;

            /* more than one flag may be set in each
            event we retrieve.  As such, we do separate
            processing for each flag, but we do not
            assume that flags are mutually exclusive. */

        if ((theFlags & kDanglingPointerDetected) != 0) {
            ShieldsUp();
        }

        if ((theFlags & kUnlockedHandleSighted) != 0) {
            PhasersOnStun();
        }

        if ((theFlags & kResEditDetected) != 0) {
            EngageHelmets();
        }

        if ((theFlags & kSelfDestruct) != 0) {
            processingEvents = false;
        }
    }
        /* release the event group */
    MPDeleteEvent(eventGroup);

        /* any result codes will be returned in the task's
        notification queue. See listing 4 for an example
        showing how to retrieve this result code. */
    return err;
}
```

Listing 7. A sample server task receiving commands by way of an event group.

As event groups are the fastest method for passing messages between tasks, their use should always be considered. If the messages your task is designed to handle do not need to be processed in any particular order, then event groups are probably the best method for sending commands to your task. On the other hand, if the commands you are sending to your task must be processed in a definite order (or you must send additional data along with the command), then you should use a queue to send commands to your task.

Back to top

Interrupt-Level Communications

Often it is desirable to communicate between code running at different execution levels. For example, an application may want to send information from its main thread to a task. This section discusses the issues involved in communications between different types of tasks and it provides an example illustrating how a preemptive task can send messages to an interrupt-level task. For the purposes of this discussion, we will define the following three task types and discuss methods that can be used to communicate between them:

1. **Interrupt Level**. This describes any thread of execution that occurs as a result of an interrupt, including hardware interrupt handlers and Deferred tasks running in the classic cooperative environment.

2. **System-Task Level**. This describes the classic application's main thread of execution and Thread Manager tasks.

3. **Preemptive-Task Level**. This refers to a thread of execution, a preemptive task, created by Multiprocessing Services.

Table 1 lists the various methods that can be used for communicating between tasks running at different execution levels. Perhaps the most complex of the types of communication that can be done here is sending a message from a preemptive task to a classic interrupt-level task. The other ones are straight forward, but, as this one is complex, a sample of how it can be done is provided in listing 8, listing 9, and listing 10.

Table 1. Methods for communicating between different execution levels for Mac OS applications.

	The Destination Task is operating at:		
The Source Task is operating at:	Interrupt Level	System-Task Level	Preemptive-Task Level
Interrupt Level	Use the `Enqueue` routine to add messages to an O.S. queue, use the `Dequeue` routine poll for messages in the interrupt routine.	Use the `Enqueue` routine to add messages to an O.S. queue, use the `Dequeue` routine to poll for messages.	Call `MPNotifyQueue` to add messages to the queue and call `MPWaitOnQueue` to extract messages. `MPSetQueueReserve` must be called to ensure there is space in the queue before adding messages to it at interrupt time.
System-Task Level		Any queue mechanism will do. The nature of system-task level ensures mutual exclusion between threads operating at this level.	Call `MPNotifyQueue` to add messages to the queue and call `MPWaitOnQueue` to extract messages.
Preemptive-Task Level	Call `Enqueue` from a deferred task to insert messages into an O.S. queue, use the `Dequeue` routine to poll for messages in the interrupt routine.	Use `MPNotifyQueue` to add messages to a queue, and use the `MPWaitOnQueue` (specifying an immediate duration) to poll the queue for messages.	

Example: sending a message to an interrupt task

In this example, message records are kept in two queues: the unused message buffers available are kept in a Multiprocessing Services queue and the message buffers containing data to be read by the interrupt task are stored in an O.S. Queue. Whenever the preemptive task needs to send a message to the interrupt routine, it can extract an unused message buffer from the Multiprocessing Services queue, copy some data to it, and then place the buffer in the O.S. Queue. The current implementation of Multiprocessing Services does not allow preemptive tasks to call `Enqueue` directly so instead this sample calls `Enqueue` from a deferred task that is installed by the preemptive task.

The interrupt routine extracts messages from the O.S. Queue and once it has finished with a message, it places the message back into the Multiprocessing Services queue so it can be used again by the preemptive task. Listing 8 contains the steps needed to set up the structures and variables used in this example.

```
#define kMaxMessages 20
#define kMessageSize 256

    /* the MessageRecord structure is defined as the
    primary mechanism to storing messages.  */

typedef struct MessageRecord MessageRecord;
typedef MessageRecord, *MessageRecPtr;
struct MessageRecord {

        /* os queue element field at offset zero.  It's
        placed here so we can easily coerce a MessageRecPtr
        to a QElemPtr and vice versa. */
    QElem qLinkField;

        /* deferred task record we will used for
        adding this record to the os queue.*/
    DeferredTask eltTask;

        /* the message buffer */
    unsigned char messageData[kMessageSize];
};


    /* pointer to the storage area we're using for messages */
MessageRecPtr gMsgStorage;


    /* a queue of free buffers */
MPQueueID gFreeMessageQueue;


    /* our message queue */
QHdr gMessageOSQueue;


    /* deferred task upp */
DeferredTaskUPP gInstallMessageDT;



/* InstallBufferDT is the deferred task we use
    for installing messages.  This routine assumes
    that its parameter is a pointer to the message
    record that is to be installed. */
static pascal void InstallBufferDT(long dtParam) {
    MessageRecPtr theElt;

        /* get a pointer to the message */
    theElt = (MessageRecPtr) dtParam;

        /* initialize its fields */
    theElt->qLinkField.qLink = NULL;
    theElt->qLinkField.qType = 0;

        /* add the message to the message queue */

Enqueue((QElemPtr) theElt, &gMessageOSQueue);

        /* in some cases, it may be useful to do
        additional processing at this point.  For
        instance, you may wish to restart a send
        operation that ran out of data, etc... */
}



/* SetUpInstallUPP is a remote procedure call used to
    set up the universal procedure pointer for the
    deferred task. This is done in a remote procedure
    call as the NewDeferredTaskUPP call is not listed
    as a macro/call that can be made from preemptive tasks.*/
static void *SetUpInstallUPP(void *parameter) {

    gInstallMessageDT = NewDeferredTaskUPP(InstallBufferDT);

    return NULL;
}



/* InitMessageQueue is called to set up the message queue
    and related variables.  The purpose of this routine is
    to illustrate what steps need to be done to prepare
    for sending messages from a preemptive task to an interrupt
    task using an O.S. Queue. */
OSStatus InitMessageQueue(void) {
    OSStatus err;
    long i;
    MessageRecPtr rover;

        /* initialize our variables */
    gFreeMessageQueue = 0;
    MPBlockClear(&gMessageOSQueue, sizeof(gMessageOSQueue));

        /* allocate our message storage */
    gMsgStorage = (MessageRecPtr) MPAllocateAligned(
        sizeof(MPSBufferElement) * kMaxMessages,
        kMPAllocateDefaultAligned,
        kMPAllocateClearMask);
    if (gMsgStorage == NULL) { err = memFullErr; goto bail; }

        /* allocate the free message queue */
    err = MPCreateQueue(&gFreeMessageQueue);
    if (err != noErr) goto bail;

        /* pre-allocate  kMaxMessages slots in the queue */
    err = MPSetQueueReserve(gFreeMessageQueue, kMaxMessages);
    if (err != noErr) goto bail;

        /* add message records to the free message queue */
    for (rover = gMsgStorage, i=0; i < kMaxMessages; i++, rover++) {
        err = MPNotifyQueue(gFreeMessageQueue, (void*) rover, NULL, NULL);
        if (err != noErr) goto bail;
    }

        /* set up the upp for the install deferred task */
    MPRemoteCall(SetUpInstallUPP, NULL, kMPOwningProcessRemoteContext);

        /* done */
    return noErr;
bail:
    if (gFreeMessageQueue != 0) MPDeleteQueue(gFreeMessageQueue);
    if (gMsgStorage != NULL) MPFree(gMsgStorage);
    return err;
}
```

Listing 8. Setting up variables and storage for sending messages to an interrupt task using an O.S. queue.

Once the necessary structures and variables have been set up, sending a message from a preemptive task to an interrupt task is simply a matter of obtaining a message buffer from the free message buffer queue, copying some data to it, and then calling the `InstallBufferDT` deferred task to install the message in the O.S. Queue. This method will not require any storage allocations inside of the preemptive task or inside of the interrupt-level task (where that is not possible), but it provides a reasonably dynamic method for passing information between the two routines. Also, if all of the message buffers are currently in the O.S. queue, the call to `MPWaitOnQueue` shown in listing 9 will wait until such a time as when the interrupt task places a buffer into the free message buffer queue.

```
OSStatus SendMessageToInterruptTask(unsigned char *message) {
    MessageRecPtr theMessage;
    OSStatus err;

        /* get a message from the free queue */
    err = MPWaitOnQueue(gFreeMessageQueue,
        (void **) &theMessage,
        NULL, NULL, kDurationForever);
    if (err != noErr) return err;

        /* copy the data to the message record */
    MPBlockCopy(message, theMessage->messageData, kMessageSize);

        /* call the deferred task to install it */
    theMessage->eltTask.qLink = NULL;
    theMessage->eltTask.qType = dtQType;
    theMessage->eltTask.dtFlags = 0;
    theMessage->eltTask.dtAddr = gInstallMessageDT;
    theMessage->eltTask.dtParam = (long) theMessage;
    theMessage->eltTask.dtReserved = 0;
    err = DTIn
stall(&theMessage->eltTask);

        /* done */
    return err;
}
```

Listing 9. Sending a message from a preemptive task to an interrupt task using an O.S. Queue.

Inside of the interrupt routine, messages can be extracted from the queue using the `Dequeue` routine. Once the message has been processed, the interrupt task can call `MPNotifyQueue` to add the message back into the free message buffer queue so it will be available to the preemptive task. Listing 10 illustrates one way an interrupt routine would extract messages from the O.S. Queue and return them to the free message buffer queue once they are no longer needed.

```
    MessageRecPtr theMessage;
    theMessage = (MessageRecPtr) gMessageOSQueue.qHead;
    if (theMessage != NULL) {
        if (
Dequeue((QElemPtr) theMessage, &gMessageOSQueue) == noErr) {

            /* perform some operations using the message... */

                /* add the message back into the free queue */
            MPNotifyQueue(gFreeMessageQueue, (void*) theMessage, NULL, NULL);
        }
    }
```

Listing 10. Receiving a message from a preemptive task inside of an interrupt task.

Back to top

Resource Management

Oftentimes when many preemptive tasks are engaged in a complex task, there must be a mechanism in place to ensure the number of tasks attempting to utilize a limited number of resources does not outnumber the actual number of resources available. For example, if we have ten tasks and two printers, and each one of those tasks must print from time to time, then a mechanism must be in place to ensure that no more than two tasks will be using the printers at any given time.

Semaphores

Semaphores allow you to restrict access to resources in a way that ensures that only a certain number of tasks may access a particular resource at any given time.

Critical regions

Critical regions are a special type of semaphore that allow you to restrict access to a particular resource (section of code) to a single execution thread.

Back to top

Tips & Tricks

- If there is a need to call a toolbox routine from an preemptive task, first check to see if the toolbox routine is interrupt safe. If it is, then call it using a deferred task rather than using a remote procedure call. Deferred tasks have a lower latency time than remote procedure calls. As a result, your code will run quicker and will not be subject to any of the unpredictable delays associated with using remote procedure calls.
- With the above tip in mind, it is worth mentioning that most of Open Transport can be called from deferred tasks.
- Avoid using global variables for sharing information between tasks. Doing so can lead to bugs that are difficult to track down. Use the routines provided by Multiprocessing services to share data between tasks.
- With Multiprocessing Services 2.1 and later, it is safe to call `MPSignalSemaphore` and `MPSetEvent` at interrupt time. It is also possible to call `MPNotifyQueue` at interrupt time if `MPSetQueueReserve` has been called to reserve sufficient space in the queue.
- Event groups are the fastest way to send information between tasks. Always consider the possibility of using them instead of queues.
- Task implementation and design should assume that other tasks are running simultaneously. When there are resources that are shared between tasks, access to them should be controlled using the resource management facilities.

Back to top

Related Materials

Multiprocessing SDK.
Multiprocessing Services Online Documentation.

Back to top

Downloadables

Acrobat version of this Note ()

Back to top

To contact us, please use the Contact Us page.
Updated: 20-March-2000

Technotes | Contents
Previous Technote | Next Technote