ð

Multiprocessor API Specification

Prepared by Apple Computer and DayStar Digital, Inc.

August 1995

Introduction

This specification defines an application programming interface (API) for multiprocessors on Macintosh. The goals are

- to support a limited class of simple, compute-only tasks that do not require Toolbox services
- to run on all current and planned generations of the PowerPC-based Mac OS—namely, System 7, Copland, and Gershwin—as well as on future versions capable of symmetric multiprocessing (SMP)
- to be simple yet powerful

Code written to this API should run without modification on Copland, and will automatically take advantage of SMP-capable systems once they are available.

For the initial implementation, only one CPU, referred to as the *main CPU,* will run the entire Mac OS and all applications. For this version, all I/O interrupts are handled by the main CPU. Additional CPUs, referred to as *attached CPUs,* will run only tasks created by this API. These tasks are known as *MP tasks*. Memory used by MP tasks is mapped into the same logical addresses on the main and attached CPUs. The attached CPUs will run a very simple operating system (*not* the Mac OS) that does nothing but implement this API. On SMP systems this model will be extended, but the API will stay the same. The API is intended for use on both on uni- and multiprocessor systems. On uniprocessor systems MP tasks are treated as preemptive threads.

The CPUs are cache-coherent with respect to each other. This API does not address the case of non-cache-coherent attached processors.

In the initial implementation, MP tasks may not call routines in the Mac OS or Toolbox, and VM must be turned off. The only calls an MP task will be able to make safely, other than to its own subroutines and functions, are to this API. (There may be a limited set of callable Toolbox routines and other utilities, but this set has not been specified yet.) These limitations will be relaxed in Copland. The target applications for the initial MP implementation are CPU-intensive domains such as image processing.

The API can support any number of CPUs, and attached CPUs need not have access to external interrupts, I/O devices, or timers. The only requirements are that all RAM be addressable by all CPUs (not necessarily at the same physical address), that the processors be cache- coherent with respect to each other, and that the CPUs be able to interrupt each other.

Areas Not Specified

Certain aspects of the multiprocessing facility are not specified here. Some must be specified before the product ships, and others have intentionally been left undefined. Although it is possible that third parties may extend this API, developers must understand that extensions to the API may not be supported by Apple in future products. The following areas have not been defined:

- *The debugging interface.* The debugging interface is implementation defined and may vary. Application code must not contain assumptions about a particular debugging facility.

- *The exact set of Toolbox, OS, and nanokernel services that may be called from an MP task.* This set is not yet determined but will be explicitly enumerated before the product ships. To ensure compatibility with future versions of the Mac OS and multiprocessing API, routines not in the list must not be called from MP tasks.

- *Task execution states and scheduling.* Task execution states and scheduling have been intentionally left unspecified. Application code must not contain assumptions about task states, execution priorities, or scheduling algorithms. Code written to this API should be capable of being executed on a variety of configurations, including uniprocessors, multiprocessors, and preemptively scheduled SMP systems.

- *Support for virtual memory (VM).* VM support is implementation defined and may vary. Ideally, the CPUs would coordinate and synchronize their memory management units (MMUs) so that VM can be used to demand-page MP tasks. Because accomplishing this coordination can be quite complex, the initial implementation on System 7 will probably require that VM be off when there is more than one CPU.

- *Environmental inquiries.* The multiprocessor API provides no way of determining the type or speed of the CPUs, or to bind particular MP tasks to particular CPUs or sets of CPUs. Applications written to this API should not depend on a particular configuration, except possibly to adapt at runtime to the number of CPUs.

- *Complete enumeration of error returns.* The only `OSStatus` codes that an application should use programmatically are documented in the sections that follow. Additional error codes may occur and will be enumerated in later documentation.

- *Task and resource cleanup.* This version of the API does not attempt to define the state of resources owned by an MP task when that task terminates. For example, if a task has a critical region locked when it terminates, the critical region may or may not be left in the locked state. To ensure correct operation with all implementations, MP resources such as queues should be explicitly deleted by the same task that allocated them, and tasks should not be asynchronously terminated unless it is known that they do not hold MP resources (other than their stack). Future implementations will provide more robust, and more precisely defined, semantics.

- *Version control.* Testing for the availability of API extensions is presently undefined but will be required before the first extension to the API ships.

Synchronization

The programming model consists of cache-coherent shared memory, with preemptively scheduled tasks running on one or more processors. The API provides a number of facilities to allow MP tasks to synchronize execution with themselves and with the main process, including semaphores, critical regions, and primitive message queues. Ad hoc synchronization methods should be avoided, because they may work on some implementations but not on others.

Typically, an application will create one or more MP tasks, and then eventually wait for those tasks to complete. When the main application thread blocks waiting for an MP event, the blocking application will not get back to its event loop until the MP event finally occurs. This delay may be unacceptable for long-running tasks. Therefore, it will often be necessary for the application to poll for MP events from its event loop, rather than block waiting for them. An MP queue is ideal for this purpose.

MP Object Identifiers

Tasks, semaphores, critical regions, and queues are abstract types represented by 32-bit IDs. Do not try to directly access the underlying data structures associated with an ID. Because it is possible that an ID may be reused soon after deletion, care should be taken to properly synchronize MP calls in order to avoid using a valid but out-of-date ID.

Durations

```
typedef                                    SInt32
                                        Duration;
enum {
                      kDurationForever     = 0x7FFFFFFF,
                             kDurationImmediate = 0
};
```

Several routines in this API take `Duration` as a parameter to limit the time spent blocked waiting for an event to occur. In future implementations, this type will allow specification of the delay in microseconds or milliseconds, much as the Time Manager does. At present, however, only two values are allowed:

`kDurationForever` waits "forever"—either until the event occurs or until the object being waited on (for example, the message queue) is deleted.

`kDurationImmediate` returns immediately, whether or not the event has occurred. The return status will be "`kMPTimeoutErr`" if the event has not occurred (and there are no other errors).

As noted above in the section “Synchronization,” using `kDurationForever` from the main application thread may be inappropriate.

Testing for Availability of the MP API

The multiprocessor API is implemented as one or more PowerPC shared libraries. To test for availability of the API at runtime, applications should weak-link to the imported symbols and use the following method to determine whether or not the library was found:

```
                                  #include "MP.h"

                           if (MPLibraryIsLoaded())
                                              <OK to
use the MP API>
                                          else
                                              <MP
API not found>
```

`MPLibraryIsLoaded` is a macro that compares the address of an API entry point to kUnresolvedCFragSymbolAddress. Testing for the availability of possible optional extensions is not defined in this version of the API.

Determining the Number of Processors

```
                                                  UInt32 MPProcessors(void);
```

The function MPProcessors returns the number of processors in the system, including the main processor.

Creating a Task

```
typedef                                   OSStatus (* TaskProc) (void *p);
typedef                                                             UInt32
MPTaskOptions;

OSStatus                                                     MPCreateTask(

                                                                  TaskProc
                                                           taskEntryPoint,

                                                                     void*

                                                            taskParameter,

                                                                 ByteCount
                                                                stackSize,

                                                                 MPQueueID
                                                                  notifyQ,

                                                                     void*

                                                         notifyParameter1,

                                                                     void*

                                                         notifyParameter2,

                                                     MPTaskOptionsoptions,

                                                                MPTaskID*
                                                               newTask );
```

The function MPCreateTaskcreates an MP task. The initial implementation may not permit this call to be made from an MP task.

The task is immediately scheduled for execution, and it executes until one of three events occur: the task terminates by returning from its entry point, the task is terminated by MPTerminateTask, or the task involuntarily terminates because of a fault or programming exception. The task is a child of the creating application process, in the sense that all extant tasks are terminated when the process terminates and in that the task address space is the process address space.

`taskEntryPoint` specifies the entry point for the task. This parameter should be the address of a subroutine that takes a single 32-bit parameter and returns an `OSStatus` value.

`taskParameter` is the parameter passed to the task entry point when it is called. The parameter is not interpreted by the system.

`stackSize` defines the length of the task's stack. It is the responsibility of the programmer to ensure that the task does not exceed the bounds of the stack. Stack overflows are not necessarily detected. If `stackSize` is 0, then a default size is used.

`notifyQ` names a message queue to which the system will send a message when the task terminates. The first two words of the message are specified when the task is created (`notifyParameter1` and `2`), and the third (an `OSStatus`) will be the return value if the task returns from its entry point, `kMPTaskAbortedErr` if the task aborts, or is supplied as a parameter to `MPTerminateTask` if the task is explicitly terminated. If `notifyQ` is `kMPNoID`, no notification of task termination will occur.

`options` specifies optional attributes of the MP task. No options are currently defined; this parameter must be 0.

`newTask` is set to the ID of the newly created task.

This function returns the following result codes that have defined semantics (additional errors may also be returned):

```
noErr
kMPInsufficientResourcesErr
```

Terminating a Task

```
OSStatus
MPTerminateTask(

MPTaskIDtask,

OSStatus
terminationStatus);
```

The function `MPTerminateTask` causes the MP task to be involuntarily terminated. The `terminationStatus` parameter is passed as the third word of the optional termination message (refer to `MPCreateTask`). Do not assume that the task has completed termination when this call returns; the proper way to synchronize with task termination is to wait for the termination message. The state of MP resources owned by the task, such as locked critical regions, is undefined in the initial implementation. It is currently an error to attempt to terminate a non-MP task.

This function returns the following result codes which have defined semantics (additional errors may also be returned):

```
noErr
kMPInvalidIDErr
```

Self Termination

```
voidMPExit(

OSStatus
terminationStatus);
```

The current MP task is terminated with a status of `terminationStatus`, just as if it had returned from its initial entry point. It is an error to call this routine from a context other than an MP task. `PExit` is equivalent to the following:

```
(void) MPTerminateTask( MPCurrentTaskID(), terminationStatus );
```

Explicitly Yielding the CPU

```
void
MPYield(void);
```

This is a hint to the scheduler, suggesting that another MP task be run. Other than possibly yielding the CPU to another task or application, the call has no effect. Note that since MP tasks are preemptively scheduled, an implicit yield may occur at any point, whether or not this function is called.

Obtaining the ID of a Task

MPTaskID MPCurrentTaskID(void);

The function `MPTaskID` returns the task ID of the current MP task. When called in execution contexts that are not MP tasks, this routine returns an ID that is unspecified but different from the ID of any MP task. Contexts that are not MP tasks may or may not have different task IDs, and future implementations of this API may behave differently in this regard. Note that ownership of a critical region is defined by task ID; therefore if multiple applications or Thread Manager threads attempt to synchronize using critical regions, the resulting behavior is currently undefined.

Task Synchronization

This API provides the following facilities for task synchronization:

- message queues
- counting semaphores
- binary semaphores
- critical regions

MP Message Queues

```
        OSStatus                                        MPCreateQueue(

        MPQueueID*                                           theQueue);
```

The function MPCreateQueue creates an MP queue, which can be used to notify (send) and wait for (receive) messages consisting of three 32-bit words, in an MP-safe manner.

```
        OSStatus                                        MPDeleteQueue(

        MPQueueID                                            theQueue);
```

The function MPDeleteQueue deletes the given queue. The queue ID becomes invalid, and all system resources associated with the queue, including queued messages, are reclaimed. Tasks that are blocked on the queue waiting for a message receive an kMPDeletedErr status.

```
        OSStatus                                        MPNotifyQueue(

        MPQueueID                                            theQueue,

        void*
        param1,

        void*
        param2,

        void*                                                         param3
);
```

The function MPNotifyQueue performs a send operation on the named queue. The system reserves an unspecified amount of memory for each queue, to buffer queued messages. If the queue is full and cannot accept any messages, then kMPInsufficientResourcesErr will be returned. The system does not interpret the three words that make up the text of the message.

```
        OSStatus                                        MPWaitOnQueue(

        MPQueueID                                            theQueue,

        void**                                                 param1,

        void**                                                 param2,

        void**                                                 param3,

        Duration                                             timeout);
```

The function MPWaitOnQueue performs a receive operation on the named queue. The parameter timeout specifies how long to wait for a message if none is queued when the call is made.

The four functions just described return the following result codes that have defined semantics (additional errors may also be returned):

```
noErr
kMPTimeoutErr
kMPDeletedErr
```

```
kMPInsufficientResourcesErr
```

Semaphore Services

```
OSStatus                                    MPCreateSemaphore(
MPSemaphoreCount                            maxVal,
MPSemaphoreCount                            initVal,
MPSemaphoreID                               *semaphore );
```

The function `MPCreateSemaphore` creates a counting semaphore with a maximum value of `maxVal` and a starting value of `initVal`.

```
OSStatus                                    MPCreateBinarySemaphore(
MPSemaphoreID                               *semaphore);
```

The function `MPCreateBinarySemaphore` creates a binary semaphore whose initial value is 1. This is simply an alias for `MPCreateSemaphore(1,1,...)`.

```
OSStatus                                    MPWaitOnSemaphore (
MPSemaphoreID                               semaphore,
Duration
timeout );
```

The function `MPWaitOnSemaphore` performs a wait operation on the semaphore: if the value of the semaphore is 0, then the task is blocked (according to the parameter `timeout`) and is placed on a queue associated with the semaphore. If the value of the semaphore is greater than 0, then the semaphore is decremented and the task proceeds.

```
OSStatus                                    MPSignalSempahore(
MPSemaphoreID                               semaphore);
```

The function `MPSignalSemaphore` performs a signal operation on the semaphore: if a task is waiting on the semaphore, then it is taken off the semaphore's internal queue and made ready to run. Otherwise, if the value of the semaphore is not already equal to its maximum value, it is incremented by one.

```
OSStatus                MPDeleteSemaphore(MPSemaphoreID semaphore);
```

The function `MPDeleteSemaphore` deletes the semaphore. All tasks waiting on the semaphore are unblocked, and `MPWaitOnSemaphore` returns `kMPDeletedErr`.

The four functions just described return the following result codes that have defined semantics (additional errors may also be returned):

```
noErr
kMPTimeoutErr
kMPDeletedErr
kMPInsufficientResourcesErr
```

Critical Region Services

```
OSStatus                                   MPCreateCriticalRegion(
MPCriticalRegionID                                  *theRegion );
```

The function `MPCreateCriticalRegion` creates a critical region object.

```
OSStatus                                    MPEnterCriticalRegion(
MPCriticalRegionID                                     theRegion,
Duration
timeout );
```

The function `MPEnterCriticalRegion` attempts to acquire, or enter, the critical region. If the critical region is being used by another task, the current task is blocked (according to the parameter `timeout`) until the critical region is released. Once a critical region has been entered, a task can then make further calls to `MPEnterCriticalRegion` without blocking. However, each call to `MPEnterCriticalRegion` must be balanced by a call to `MPExitCriticalRegion`. For the purposes of critical region ownership, contexts that are not MP tasks behave as defined by `MPCurrentTaskID`.

```
OSStatus                                     MPExitCriticalRegion(
MPCriticalRegionID                                   theRegion );
```

The function `MPExitCriticalRegion` releases the critical region object. It is an error to release an already free critical region or one that has been entered by another task.

```
OSStatus                                   MPDeleteCriticalRegion(
MPCriticalRegionID                                   theRegion );
```

The function `MPDeleteCriticalRegion` deletes the critical region object. For tasks blocked waiting to enter a critical region, `MPEnterCriticalRegion` will return `kMPDeletedErr`.

The four functions just described return the following result codes that have defined semantics (additional errors may also be returned):

```
noErr
kMPTimeoutErr
kMPDeletedErr
kMPInsufficientResourcesErr
```

Additional Routines Callable From MP Tasks

```
LogicalAddress                    MPAllocate(
      ByteCount                                                         size );
```

```
void                               MPFree(
      LogicalAddress               block );
```

`MPAllocate` and `MPFree` allocate and free non-relocatable blocks of memory in heap storage. The allocated memory will be aligned and padded in order to provide safe and efficient access in an MP system, for example by avoiding cache thrashing due to coherence granularity. The memory is guaranteed accessible, using the returned address, by the main application and all MP tasks that it creates, but not by other applications or their MP tasks. Extant heap blocks are freed when the application terminates. As with all shared memory, access to allocated heap blocks must be explicitly synchronized. If `MPAllocate` is unable to allocate the requested amount of memory, it returns a null pointer.

```
void                              MPBlockCopy(
      LogicalAddress              sourcePtr,
      LogicalAddress               destPtr,
      ByteCount                                                        bytes );
```

This routine copies possibly-overlapping blocks of memory. It is similar in function to `BlockMoveData`.

Note that, in general, no MacOS Toolbox or library routines other than those defined in this ERS may be safely called from an MP task. Even if it appears that some Toolbox (or other) routine works today when called from an MP task, unless explicitly stated otherwise there is no guarantee that subsequent versions of the same routine will continue to work. For System 7 implementations of the MP API, the only exceptions to this rule are the atomic memory operations (such as `AddAtomic`) in Synchronization.h. In Copland, more MacOS functionality will be available from MP tasks.