Process Manager in Macintosh Operating System

Mindfire Solutions

www.mindfiresolutions.com

March 6, 2002

Abstract:

This paper discusses managing processes and takes in Macintosh operating system. This paper is a summarized form of "Inside Macintosh: Processes" and is directed towards developers who are new to Mac development but had previous development experience on other operating system. After going through this article you will be familiar with Process Manager, Time Manager, Vertical Retrace Manager, Notification Manager and several other process related services provided by the system.

INTRODUCTION TO PROCESS AND TASKS

• *The Cooperative Multitasking Environment*

The Macintosh Operating System, the Finder, and several other system software components work together to provide a **multitasking environment** in which a user can have multiple applications open at once and can switch between open applications as desired. To run in this environment, however, your application must follow certain rules governing its use of the available system resources. For example, your application should include a `'SIZE'` resource that specifies how large a memory partition it should be allocated at application launch time. If that much memory is available when your application is launched, the Process Manager allocates it and sets up your application partition. Similarly, your application should periodically make an event call to allow the Operating System the opportunity to schedule other applications for execution. Because the smooth operation of all applications depends on their cooperation, this environment is known as a **cooperative multitasking environment.**

The CPU is available only to the current application, whether it is running in the foreground or the background. The application can be interrupted only by hardware interrupts, which are transparent to the application. However, to give processing time to background applications and to allow the user to interact with your application and others, you must periodically call the Event Manager's `WaitNextEvent` or `EventAvail` function to allow your application to relinquish control of the CPU for short periods. By using these event routines in your application, you allow the user to interact not only with your application, but also with other applications. Although a number of documents and applications can be open at the same time, only one application is the active application.

• *About Processes*

The Process Manager manages the scheduling of processes.
The number of processes is limited only by available memory. The Process Manager maintains information about each process—for example, the current state of the process, the address and size of its partition, its type, its creator, a copy of all process-specific system global variables, information about its `'SIZE'` resource, and a process serial number. This process information is referred to as the **context** of a process. The Process Manager assigns a **process serial number** to identify each process. A process serial number identifies a particular instance of an application; this number is unique during a single boot of the local machine. The foreground process has first priority for accessing the CPU. Other processes can access the CPU only when the foreground process yields time to them. There is only one foreground process at any one time. However, multiple processes can exist in the background.
An application that is in the background can get CPU time but can't interact with the user while it is in the background. (However, the user can bring the application to the

foreground—for example, by clicking in one of the application's windows.) Any application that has the `canBackground` flag set in its `'SIZE'` resource is eligible to obtain access to the CPU when it is in the background.
Applications can be designed without a user interface; these are called **background-only applications.** A background-only application does not call the Window Manager `InitWindows` routine and is identified by having the `onlyBackground` flag set in its `'SIZE'` resource. Background-only applications do not display windows or a menu bar and are not listed in the Application menu.
Your application can relinquish control of the CPU each time you call the Event Manager functions `WaitNextEvent` or `EventAvail`. If, at that time, there are no events pending for your application, the Process Manager may schedule other processes for execution. (You can also call the `GetNextEvent` function; however, you should use `WaitNextEvent` to provide greater support for cooperative multitasking.)

Process Creation

The application **jump table** contains one entry for every externally referenced routine in every code segment of your application.
When you create an application, you specify in its `'SIZE'` resource how much memory you want the Process Manager to allocate for your application's partition. You specify two values: the preferred amount of memory to allocate and the minimum amount of memory to allocate. When a user opens your application from the Finder, the Process Manager first attempts to allocate a partition of the preferred size. If your application cannot be launched in the preferred amount of memory, the Finder might display a dialog box giving the user the option of opening the application using less than the preferred size. The Finder will not launch your application if the minimum amount of memory specified for your application is not available.
The Process Manager assigns the application a process serial number, records its context, and returns control to the launching application (usually the Finder). The Process Manager typically transfers control to the new application after the launching application makes a subsequent call to `WaitNextEvent` or `EventAvail`.

Process Scheduling

When your application is the foreground process, it yields time to other processes in these situations: when the user wants to switch to another application or when no events are pending for your application. Your application can also choose to yield processing time to other processes when it is performing a lengthy operation.

A **major switch** occurs when the Process Manager switches the context of the foreground process with the context of a background process (including the A5 worlds and application-specific system global variables) and brings the background process to the front, sending the previous foreground process to the background.

A **minor switch** occurs when the Process Manager switches the context of a process to give time to a background process without bringing the background process to the front.

A background process should not perform any task that significantly limits the ability of the foreground process to respond quickly to the user. A background process should call `WaitNextEvent` often enough to let the foreground process be responsive to the user. The `sleep` parameter of the `WaitNextEvent` function specifies a length of time, in ticks, during which the application relinquishes the CPU if no events are pending. For example, if you specify a nonzero value in the `sleep` parameter and no events are pending in your application's event queue when you call `WaitNextEven`t, the Process Manager saves the context of your process and schedules other processes until an event becomes available or the time expires. Once the specified time expires or an event becomes available for your application, your process becomes eligible to run. At this time, the Process Manager schedules your process to run at the next available chance. (You can also call the Process Manager's `WakeUpProcess` function to make a process eligible to run before the time in the `sleep` parameter expires.) If the time specified by `sleep` expires and no events are pending for your application, the Process Manager sends your application a null event.

- ***About Tasks***

An **interrupt** is a form of **exception,** an error or special condition detected by the microprocessor in the course of program execution. In particular, an interrupt is an exception that is signaled to the processor by a device.

Interrupts can occur not only between different statements that your application executes but also in the middle of a single call that your application makes. Interrupts are usually sent by a device to notify the microprocessor of a change in the condition of the device. Routines that are executed as a result of an interrupt are known as **interrupt tasks.**

Task Creation

Many interrupt tasks are handled by system software and are transparent to your application. However, our application can use any of several facilities to install its own interrupt tasks that are executed not at regular points in the flow of its code but at intervals determined by hardware devices.

A. The Time Manager allows you to schedule periodic tasks and tasks to be executed after a certain amount of time has elapsed. You can, for example, use the Time Manager to compute elapsed times with great precision.

B. The Vertical Retrace Manager allows you to schedule tasks to be executed between retraces of a video screen. Tasks that you schedule with the Vertical Retrace Manager can reset themselves, just like Time Manager tasks. Although the Vertical Retrace Manager lacks the great precision of the Time Manager, it is available on all Macintosh models.
C. The Notification Manager allows both processes in the background and interrupt tasks to alert the user. For example, your application might need to inform the user that some error has occurred, rendering further background processing impossible. You can pass to

the Notification Manager's installation routine a pointer to a response procedure to be executed as the final stage of notification.

D. The Device Manager allows device drivers for slot cards to install interrupts. If you are writing slot interrupt tasks, you might also wish to use the Deferred Task Manager, which allows you to defer lengthy interrupt tasks that might prevent other interrupt tasks from executing.

All of these managers need to maintain information about multiple interrupt tasks that might have been installed. To hold such information, the Operating System uses data structures known as **operating-system queues.**
When an interrupt causes the microprocessor to suspend normal execution, the processor uses the stack to save the address of the next instruction and the processor's internal status. In this way, when the microprocessor completes execution of interrupt tasks, it can resume the current process where it left off.
After storing these values on the stack, the microprocessor executes an **interrupt handler** to deal with the interrupt. When an interrupt task is executed, the interrupt is said to be **serviced.**

Task Scheduling

As previously indicated, your interrupt tasks are executed in response to an interrupt. Because the execution of an interrupt task is not tied to the normal execution of your application, that task might continue to be executed even when your application is not itself executing. For example, all Time Manager tasks installed by your application continue to be executed as scheduled, whether or not your application is still the current application. If it doesn't make sense to continue executing a particular Time Manager task when your application is no longer receiving processing time, you need to disable the execution of that task whenever your application is switched out and then re-enable the task when your application regains control of the CPU. To disable a Time Manager task, you can remove its entry from the Time Manager queue. To re-enable it, reinstall its entry in the queue. In some cases, the Operating System automatically disables some of your application's interrupt tasks when your application is switched out. All VBL tasks installed by the Vertical Retrace Manager routine `VInstall` (which are known as system-based VBL tasks) are disabled whenever the installing application loses control of the CPU, if the address of the task is in the application partition. If you want to continue executing a system-based VBL task when your application is switched out, you must make sure that the address of the task is in the system partition.

When an interrupt task is executed, the Operating System does not always restore the installing application's context. As a result, you might not be able to read any application-specific system global variables from within the task. In addition, the task will not have access to any application-installed patches (which are part of its context). If your interrupt task depends on any part of your application's context, it should call the Process Manager function `GetCurrentProcess` to make sure that your process is currently in control of the CPU and hence that its context is valid.

PROCESS MANAGER

The Process Manager provides a cooperative multitasking environment, similar to the features provided by the MultiFinder option in earlier versions of system software. You can use the `Gestalt` function to find out if the Process Manager routines are available and to see which features of the `Launch` function are available.

- ***Using the Process Manager***

Getting Information About Other Processes

You can call the `GetNextProcess`, `GetFrontProcess`, or `GetCurrentProcess` functions to get the process serial number of a process. The `GetCurrentProcess` function returns the process serial number of the process currently executing, called the **current process.** This is the process whose A5 world is currently valid; this process can be in the background or foreground. The `GetFrontProcess` function returns the process serial number of the foreground process. For example, if your process is running in the background, you can use `GetFrontProcess` to determine which process is in the foreground.

You can use these constants to specify special processes in process manager routines:

```
CONST
kNoProcess = 0; {process doesn't exist}
kSystemProcess = 1; {process belongs to OS}
kCurrentProcess = 2; {the current process}
```

You can call the `GetProcessInformation` function to obtain information about any process, including your own. For example, for a specified process, you can find

1. the application's name as it appears in the Application menu
2. the type and signature of the application
3. the number of bytes in the application partition
4. the number of free bytes in the application heap
5. the application that launched the application

Launching Other Applications

You can launch other applications by calling the high-level `LaunchApplication` function. This function lets your application control various options associated with launching an application. For example, you can

1. allow the application to be launched in a partition smaller than the preferred size but greater than the minimum size, or allow it to be launched only in a partition of the preferred size
2. launch an application without terminating your own application, bring the launched application to the front, and get information about the launched application

3. request that your application be notified if any application that it has launched terminates

However, if your application includes a desk accessory or another application, you might use either the high-level `LaunchApplication` function to launch an application or the `LaunchDeskAccessory` function to launch a desk accessory.

Note that if you launch another application without terminating your application, the launched application does not actually begin executing until you make a subsequent call to `WaitNextEvent` or `EventAvail`.

By default, `LaunchApplication` brings the launched application to the front and sends the foreground application to the background. If you don't want to bring an application to the front when it is first launched, set the `launchDontSwitch` flag in the `launchControlFlags` field of the launch parameter block. In addition, if you want your application to continue to run after it launches another application, you must set the `launchContinue` flag in the `launchControlFlags` field of the launch parameter block.

Terminating an Application

The Process Manager automatically terminates a process when the process either exits its main routine or encounters a fatal error condition (such as an attempt to divide by 0).

When a process terminates, the Process Manager takes care of any required cleanup operations; these include removing the process from the list of open processes and releasing the memory occupied by the application partition (as well as any temporary memory the process still holds). If necessary, the Process Manager sends an Application Died event to the process that launched the one about to terminate.
Your application can also terminate itself directly by calling the `ExitToShell` procedure. In general, you need to call `ExitToShell` only if you want to terminate your application without having it return from its main routine.

Note: The `ExitToShell` procedure is the only means of terminating a process. It is always called during process termination, whether by your application itself, the Process Manager, or some other process.

Data Structures

Process Serial Number

The Process Manager uses process serial numbers to identify open processes. A process serial number is a 64-bit quantity whose structure is defined by the `ProcessSerialNumber` data type.

Note: The meaning of the bits in a process serial number is internal to the Process Manager. You should not attempt to interpret the value of the process serial number. If you need to compare two process serial
numbers, call the `SameProcess` function.

```
TYPE ProcessSerialNumber =
RECORD
highLongOfPSN: LongInt; {high-order 32 bits of psn}
lowLongOfPSN: LongInt; {low-order 32 bits of psn}
END;
```

Process Information Record

The `GetProcessInformation` function returns information in a process information record, which is defined by the `ProcessInfoRec` data type.

```
TYPE ProcessInfoRec =
RECORD
processInfoLength: LongInt; {length of process info record}
processName: StringPtr; {name of this process}
processNumber: ProcessSerialNumber;{psn of this process}
processType: LongInt; {file type of application file}
processSignature: OSType; {signature of application file}
processMode: LongInt; {'SIZE' resource flags}
processLocation: Ptr; {address of partition}
processSize: LongInt; {partition size}
processFreeMem: LongInt; {free bytes in heap}
processLauncher: ProcessSerialNumber; {process that
launched this one}
processLaunchDate: LongInt; {time when launched}
processActiveTime: LongInt; {accumulated CPU time}
processAppSpec: FSSpecPtr; {location of the file}
END;
```

TIME MANAGER

- ## *About the Time Manager*

You can use the TimeManager to

1. schedule routines for execution after a specified delay
2. set up tasks that run periodically
3. compute the time a routine takes to run
4. coordinate and synchronize actions in the Macintosh computer

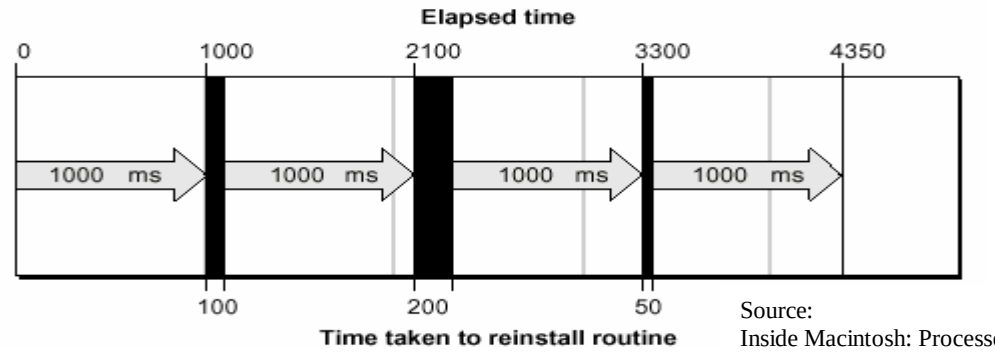
To use the Time Manager, you must first issue a request by passing the Time Manager the address of a task record, one of whose fields contains the address of the routine that is to run. Then you need to activate that request by specifying the delay until the routine is

to run. The Time Manager uses a **Time Manager queue** to maintain requests that you issue. The structure of this queue is similar to that of standard operating-system queues. The Time Manager queue can hold any number of outstanding requests, and each application can add any number of entries to the queue. If there are several requests scheduled for execution at exactly the same time, the Time Manager schedules them for execution as close to the specified time as possible, in the order in which they entered the Time Manager queue.
The routine you place in the queue can perform any desired action so long as it does not call the Memory Manager, either directly or indirectly. (You cannot call the Memory Manager because Time Manager tasks are executed at interrupt time.)

The Time Manager introduced in system software version 7.0 is the third version released. The three versions are known as the original Time Manager, the revised Time Manager, and the extended Time Manager. The three versions are all upwardly compatible— You can use the `Gestalt` function to determine which version of the Time Manager is present. You should pass `Gestalt` the selector `gestaltTimeMgrVersion`.

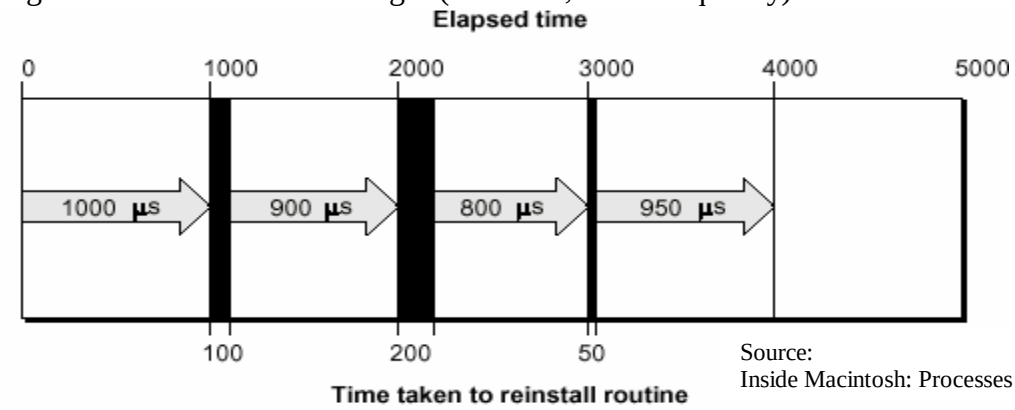
The Extended Time Manager

Fig: Original and revised Time Managers (drifting, unpredictable frequency)
Source: Inside Macintosh: Processes



Source:
Inside Macintosh: Processes

Fig: The extended Time Manager (drift-free, fixed frequency)



Source:
Inside Macintosh: Processes

- ***Using the Time Manager***

The Time Manager is automatically initialized when the system starts up. At that time, the queue of Time Manager task records is empty. The Operating System, applications, and other software components may place records into the queue. Because the delay time for a given task can be as small as 20 microseconds, you need to install an element into the Time Manager queue before actually issuing a request to execute it at some future time. You place elements into the queue by calling the `InsTime` procedure or (if you need the fixed-frequency services of the extended Time Manager) the `InsXTime` procedure. To activate the request, call `PrimeTime`. The Time Manager then marks the specified task record as active by setting the high-order bit in the `qType` field of that record. The `tmAddr` field of the Time Manager task record contains the address of a task. The Time Manager calls this task when the time delay specified by a previous call to `PrimeTime` has elapsed. The task can perform any desired actions, as long as it does not call the Memory Manager (either directly or indirectly) and does not depend on the validity of handles to unlocked blocks.

Note: If the routine specified in the Time Manager task record is located in your application's heap, then your application must still be active when the specified delay elapses, or the application should call `RmvTime` before it terminates. Otherwise, the Time Manager does not know that the address of that routine is not valid when the routine is called. The Time Manager then attempts to call the task, but with a stale pointer. If you want to let the application terminate after it has installed and activated a Time Manager task record, load the routine into the system heap.

There are two ways for an active queue element to become inactive. First, the specified time delay can elapse, in which case the routine pointed to by the `tmAddr` field is called. Second, your application can call the `RmvTime` procedure, in which case the amount of time remaining before the delay would have elapsed (the unused time) is reported in the `tmCount` field of the task record. This feature allows you to use the Time Manager to compute elapsed times, which is useful for obtaining performance measurements. Calling `RmvTime` removes an element from the queue whether or not that task is active when `RmvTime` is called.

Installing and Activating Tasks

Listing 3-1 shows how to install and activate a Time Manager task. It assumes that the procedure `MyTask` has already been defined; see Listing 3-3 and Listing 3-4 for examples of simple task definitions.

Listing: Installing and activating a Time Manager task

```
PROCEDURE InstallTMTask;
CONST
kDelay = 2000; {delay value}
BEGIN
gTMTask.tmAddr := @MyTask; {get address of task}
gTMTask.tmWakeUp := 0; {initialize tmWakeUp}
```

```
gTMTask.tmReserved := 0; {initialize tmReserved}
InsXTime(@gTMTask); {install the task record}
PrimeTime(@gTMTask, kDelay); {activate the task record}
END;
```

In this example, `InstallTMTask` installs an extended Time Manager task record into the Time Manager queue and then activates the task.

Using Application Global Variables in Tasks

When a Time Manager task executes, the A5 world of the application that installed the corresponding task record into the Time Manager queue might not be valid (for example, the task might execute at interrupt time when that application is not the current application). If so, an attempt to read the application's global variables returns erroneous results because the A5 register points to the application global variables of some other application. When a Time Manager task uses an application's global variables, you must ensure that register A5 contains the address of the boundary between the application global variables and the application parameters of the application that launched it. You must also restore register A5 to its original value before the task exits. It is relatively straightforward to read the current value of the A5 register when a Time Manager task begins to execute (using the `SetCurrentA5` function) and to restore it before exiting (using the `SetA5` function). It is more complicated, however, to pass to a Time Manager task the value to which it should set A5 before accessing its application's global variables. The problem is that neither the original nor the extended Time Manager task record contains an unused field in which your application could pass this information to the task.

Here is a source code snippet showing use of timers in an application

FILE: M_Timer.h

```
#pragma once
#include <Timer.h>

// Data structures
typedef  struct _TaskRecord //timer task running during our computations
{
TMTask  theTask;     // task queue entry
long    A5;          // applications A5
} TaskRecord;

// Tools for our application
void InstallTimer(void);
void RemoveTimer(void);
void StartTimer(void);
void StopTimer(void);
```

```
// Assembler Glue for 68k ONLY !!!
#ifndef POWERMAC
static pascal TaskRecord *GetTaskInfo(void) = 0x2E89;   // MOVE.L    A1,(SP)
#endif

// Timer Routine
#ifdef POWERMAC
   static void         MyTimerTask(TMTaskPtr recPtr);
#else
   static pascal void  MyTimerTask(void);
#endif
```

FILE: M_Timer.c

```
#include "M_Timer.h"
// Globals

// application Flag
extern long TIMER_EVENT_OCCURED;

// Time between two Interrupts in ms
#define kTaskTime   200L

// Record pointing to our interrupt routine
static TaskRecord  gBgndTask;

// Universal Proc Pointer for PPC
#ifdef POWERMAC
UniversalProcPtr    PPC_MyTimerTask;
#endif

void InstallTimer(void)
{
#ifdef POWERMAC
   PPC_MyTimerTask = NewRoutineDescriptor( (ProcPtr)MyTimerTask,
uppTimerProcInfo, GetCurrentISA() );
#endif
   gBgndTask.theTask.qType         = 0;
   gBgndTask.theTask.tmAddr        = NULL;
   gBgndTask.theTask.tmCount       = 0L;
   gBgndTask.theTask.tmWakeUp      = 0L;
   gBgndTask.theTask.tmReserved    = 0L;
   gBgndTask.A5 = SetCurrentA5();
  InsTime((QElemPtr) &gBgndTask);
}
```

```
void RemoveTimer(void)
{
   if (gBgndTask.theTask.tmAddr)
   {
      RmvTime((QElemPtr) &gBgndTask);
      gBgndTask.theTask.tmAddr = NULL;
   }
#ifdef POWERMAC
   DeleteRoutineDescriptor(PPC_MyTimerTask);
#endif
}

void StartTimer(void)
{
#ifdef POWERMAC
   gBgndTask.theTask.tmAddr = PPC_MyTimerTask;
#else
   gBgndTask.theTask.tmAddr = MyTimerTask;
#endif
   PrimeTime((QElemPtr) &gBgndTask, kTaskTime);
}

void StopTimer(void)
{
   gBgndTask.theTask.tmAddr = NULL;
}

#ifdef POWERMAC //PowerPC Version
static void MyTimerTask(TMTaskPtr recPtr)
{
   TIMER_EVENT_OCCURED = 1L;
   PrimeTime((QElemPtr) recPtr, kTaskTime); //PERIODIC TIMER
}
#else   // not POWERMAC – old 68K-Version
static pascal void MyTimerTask(void)
{
   TaskRecord  *recPtr;
   long        oldA5;
   recPtr = GetTaskInfo();
   oldA5 = SetA5(recPtr->A5);
   TIMER_EVENT_OCCURED = 1L;
   PrimeTime((QElemPtr) recPtr, kTaskTime);
   oldA5 = SetA5(oldA5);
}
#endif  // not POWERMAC
```

Computing Elapsed Time

In the revised and extended Time Managers, the `RmvTime` procedure returns, in the `tmCount` field of the task record, a value representing any unused time. This feature makes the Time Manager extremely useful for computing elapsed times. To compute the amount of time that a routine takes to run, call `PrimeTime` at the beginning of the interval to be measured and specify a delay greater than the expected elapsed time. Then call `RmvTime` at the end of the interval and subtract the unused time returned in `tmCount` from the original delay passed to `PrimeTime`.

- ***Important Routines***

InsTime

You can install a task record into the Time Manager task queue using the `InsTime` procedure.

```
PROCEDURE InsTime (tmTaskPtr: QElemPtr);
```

`tmTaskPtr` A pointer to an original task record to be installed in the queue.

DESCRIPTION: The `InsTime` procedure adds the Time Manager task record specified by `tmTaskPtr` to the Time Manager queue. Your application should fill in the `tmAddr` field of the task record and should set the `tmCount` field to 0. The `tmTaskPtr` parameter must point to an original Time Manager task record.

InsXTime

Use the `InsXTime` procedure to install a task if you want to take advantage of the drift-free, fixed-frequency timing services of the extended Time Manager.

```
PROCEDURE InsXTime (tmTaskPtr: QElemPtr);
```

`tmTaskPtr` A pointer to an extended task record to be installed in the queue.

DESCRIPTION: The `InsXTime` procedure adds the Time Manager task record specified by `tmTaskPtr` to the Time Manager queue. The `tmTaskPtr` parameter must point to an extended Time Manager task record. Your application must fill in the `tmAddr` field of that task. You should set the `tmWakeUp` and `tmReserved` fields to 0 the first time you call `InsXTime`.

PrimeTime:

Use the `PrimeTime` procedure to activate a task in the Time Manager queue.

```
PROCEDURE PrimeTime (tmTaskPtr: QElemPtr; count: LongInt);
```

`tmTaskPtr` A pointer to a task record already installed in the queue.

`count` The desired delay before execution of the task.

DESCRIPTION: The `PrimeTime` procedure schedules the task specified by the `tmAddr` field of `tmTaskPtr` for execution after the delay specified by the `count` parameter has elapsed. If the `count` parameter is a positive value, it is interpreted as milliseconds. If `count` is a negative value, it is interpreted in negated microseconds. (Microsecond delays are allowable only in the revised and extended Time Managers.)

The task record specified by `tmTaskPtr` must already be installed in the queue (by a previous call to `InsTime` or `InsXTime`) before your application calls `PrimeTime`. `PrimeTime` returns immediately, and the specified task is executed after the specified delay has elapsed. If you call `PrimeTime` with a time delay of 0, the task runs as soon as interrupts are enabled.

RmvTime

Use the `RmvTime` procedure to remove a task from the Time Manager queue.

```
PROCEDURE RmvTime (tmTaskPtr: QElemPtr);
```

`tmTaskPtr` A pointer to a task record to be removed from the queue.

Time Manager Tasks

You pass the address of an application-defined Time Manager task in the `tmAddr` field of the Time Manager task record.

MyTimeTask

A Time Manager task has the following syntax:

```
PROCEDURE MyTimeTask;
```

DESCRIPTION: The `tmAddr` field of a Time Manager task record contains the address of a task procedure that is executed after the delay time passed to `PrimeTime`.

VERTICAL RETRACE MANAGER

This chapter describes the Vertical Retrace Manager, the part of the Operating System that schedules and executes recurrent tasks during vertical retrace interrupts. You can use the Vertical Retrace Manager to execute simple, repetitive tasks and avoid having to execute those tasks repeatedly in your application's main event loop.

You should read the information in this chapter if you want your application to schedule tasks for execution during a vertical retrace interrupt. For example, you can use the Vertical Retrace Manager to cycle among a series of cursors while some lengthy operation is happening, thus presenting the illusion of a spinning cursor.

you can use the Vertical Retrace Manager to

1. install a simple task to be executed during vertical retrace interrupts
2. access information about a task record installed in the vertical retrace queue fromwithin that task
3. access your application's global variables in a vertical retrace task
4. spin the cursor to indicate that the user must wait while the computer completes some lengthy processing
5. install a vertical retrace task in the system heap so that it continues to be executed even when your application is switched out

- ***About the Vertical Retrace Manager***

The video circuitry in a Macintosh computer, whether built-in or external, refreshes the screen at regular intervals. To refresh the screen, the monitor's electron beam draws one pixel at a time, starting at the upper-left corner of the screen and moving quickly to the lower-right corner. When the electron beam returns from the lower-right corner of the screen to the upper-left corner, the video circuitry generates a **vertical retrace interrupt** or **vertical blanking (VBL) interrupt.**

The Vertical Retrace Manager is the part of the Operating System that schedules and executes tasks—known as **VBL task**s—during a vertical retrace interrupt. The Operating System itself uses the Vertical Retrace Manager to perform some important housekeeping operations, such as moving the cursor in response to mouse movements and checking whether the current application's stack has expanded into its heap. In general, the Vertical Retrace Manager is useful for small, repetitive tasks that do not allocate or release memory and that you do not want to execute in your main event loop.

The principal limitation on VBL tasks (aside from the limitations on any interrupt-time processing) is that they cannot execute more frequently than once per VBL interrupt. The exact amount of time between successive VBL interrupts depends on the refresh frequency of the screen, which varies. On Macintosh computers that have a built-in screen (such as the Macintosh Plus or Macintosh Classic), the vertical retrace frequency is approximately 60.15 Hz, resulting in a period of approximately 16.63 milliseconds. If you need a task to be executed more often than that, you should use the Time Manager, which has a much finer resolution (up to 250 microseconds for drift-free task execution). Unlike the Time Manager, the Vertical Retrace Manager is not an absolute timing mechanism.

VBL Tasks Installed by the Operating System

The Operating System uses the Vertical Retrace Manager to accomplish a number of repetitive tasks at uniform intervals. These are some of the VBL tasks installed by the Operating System, grouped by the intervals at which they execute:

1. Every interrupt
2. Update the value of the global variable `Ticks`, which a program may access through the routine `TickCount`.
3. Call the "stack sniffer" to see if the current application's stack and heap have collided. If so, the task calls the System Error Handler.
4. Update the position of the cursor.
5. Every 30 interrupts
6. Check whether the user has inserted a disk or mounted a volume. If so, the task posts a disk-inserted event.
7. Every 32 interrupts
8. Check whether a keyboard has been reattached after having been detached. If so, the task resets the keyboard.

Types of VBL Tasks

There are two general types of VBL tasks. A **slot-based VBL task** is linked to an external video monitor. Because different monitors can have different refresh rates and hence might execute VBL tasks at different times, the Vertical Retrace Manager maintains a separate task queue for each video device attached to the computer. When a VBL interrupt occurs for a particular device, the Vertical Retrace Manager executes any tasks in the queue for the slot holding that monitor's video card. You can install a slot-based VBL task by calling the `SlotVInstall` function.
For Macintosh computers that have only a built-in monitor (such as a Macintosh Plus or Macintosh Classic), there is no need to isolate VBL tasks into separate queues. Instead, the Operating System maintains just one task queue and processes the tasks in that queue when it receives a VBL interrupt. A VBL task that is not linked to an external video device is known as a **system-based VBL task.** You can install a system-based VBL task by calling the `VInstall` function.

To maintain compatibility on modular Macintosh computers for software that uses the `VInstall` function, the Operating System generates a special interrupt at a frequency identical to the retrace rate on compact Macintosh computers. This special interrupt is generated approximately 60.15 times a second and mimics the vertical retrace interrupt on compact models. This ensures that application tasks installed using the `Vinstall` function, as well as periodic system tasks such as updating the tick count and checking whether the stack has expanded into the heap, are performed as usual.

Like all interrupt tasks, VBL tasks cannot do everything that ordinary routines can. The following list summarizes the operations that VBL tasks should not perform. A VBL task that violates one of these rules may cause a system crash:

1. A VBL task must not allocate, move, or purge memory, or call any Toolbox routines that might do so.
2. A VBL task must preserve all registers other than A0–A3 and D0–D3.
3. A VBL task cannot call a routine from another code segment unless it sets up the application's A5 world properly. In addition, that segment must already be loaded in memory.
4. A VBL task cannot access your application global variables unless it sets up the application's A5 world properly.
5. A VBL task's code and any data accessed during the execution of the task must be locked into physical memory if virtual memory is in operation.

The VBL Task Record

You install a VBL task by passing the Vertical Retrace Manager the address of a **VBL task record,** which holds information about your VBL task. This information includes the address of the procedure the Vertical Retrace Manager is to execute at interrupt time and the number of interrupts before it should next execute the task. The `VBLTask` data type defines a VBL task record.

```
TYPE VBLTask =
RECORD
qLink: QElemPtr; {next entry in vertical retrace queue}
qType: Integer; {queue type}
vblAddr: ProcPtr; {pointer to task procedure}
vblCount: Integer; {interrupts until next execution}
vblPhase: Integer; {task phase}
END;
```

Your application needs to fill in only the `qType`, `vblAddr`, `vblCount`, and `vblPhase` fields of the VBL task record. The `qLink` field, which contains a pointer to the next entry in the VBL task's vertical retrace queue, is set by the Vertical Retrace Manager when you install the task by calling `VInstall` or `SlotVInstall`. Your application does not need to set up the `qLink` field.

When installing a VBL task, you specify, in the `vblCount` field, the number of interrupts before the routine first executes. The Vertical Retrace Manager lowers this number by 1 during each interrupt. If decrementing `vblCount` produces a value of 0, the Vertical Retrace Manager executes the procedure specified in the task record's `vblAddr` field. If you want the procedure to be executed again, that procedure is responsible for resetting the value of the `vblCount` field to the desired value.
If you do not want the Vertical Retrace Manager to execute the task again, your task should leave the value of `vblCount` at 0. Setting the `vblCount` field to 0 is one way of disabling a task. (A more common approach is to remove the task record from its queue by calling `VRemove` or `SlotVRemove`, but this should not be done by the VBL task itself.)

Vertical Retrace Queues

The Vertical Retrace Manager stores application-defined VBL task records in **vertical retrace queue**s, which are standard operating-system queues. If multiple tasks in the same vertical retrace queue are scheduled to be executed during the same interrupt, the Vertical Retrace Manager will execute the tasks in the order they were installed in the queue.

Compact Macintosh computers maintain only one vertical retrace queue, because these computers have only one screen. However, computers with multiple screens require multiple vertical retrace queues. Because slot-based task installation and removal routines apply to just one slot, the Vertical Retrace Manager maintains a separate vertical retrace queue for each slot that contains a video card.

Ordinarily, you do not need to inspect or manipulate the contents of vertical retrace queues directly. Instead, you can use the Vertical Retrace Manager routines for installing task records in and removing them from vertical retrace queues. In one case, however, you might need to inspect the header of a vertical retrace queue. If you need to know

whether some code is being called in response to a VBL interrupt, you can inspect the `qFlags` field of the queue header. The Vertical Retrace Manager sets bit 6 of the `qFlags` field in the queue header to indicate that a VBL task in the queue is being executed.

VBL Tasks and Application Execution

Often, a VBL task performs services that are useful only to the application that installed it. For instance, consider the VBL task for "Spinning the Cursor". This task spins the cursor while your application performs some lengthy operation and should be executed only if your application is in the foreground. If the user switches your application into the background while it is occupied with that lengthy operation, you probably want to disable that task for as long as your application is in the background. Otherwise, the cursor will continue to spin, probably confusing the user.
In other cases, a VBL task should continue to be executed even when the application that installed it is no longer in the foreground. For instance, you probably wouldn't want to disable a VBL task that periodically checks for the arrival of electronic mail just because your application is moved to the background.
The Process Manager automatically disables a system-based VBL task when the application that installed it is swapped out in a major or minor switch, if the address of the VBL task is anywhere in the application's partition. Then, when that application regains control of the processor, the Process Manager reenables that VBL task. If, however, the address of a system-based VBL task is in the system partition, the VBL task continues to be executed, regardless of the processing status of the application that launched it.

By contrast, the Process Manager never disables a slot-based VBL task, no matter where the task is located. As a result, if you want to disable a slot-based VBL task when your application is in the background, you must do so yourself, either by removing the task record from the VBL queue or by setting the `vblCount` field of the task record to 0. You can do this in response to a suspend event. Then, when your application receives a resume event, you can reenable the VBL task by reinstalling the task record or by resetting the `vblCount` field of the task record to the appropriate value. In some cases, you might want to disable a *syste*m-based VBL task manually, even though the Process Manager also disables it when your application is switched out. This is because the Process Manager reenables system-based VBL tasks when your application receives processing time as a result of a minor switch, when your application is still in the background. If the VBL task should be executed only when your application is in the foreground, you need to disable it when your application receives a suspend event and reenable it when your application receives a resume event. The easiest way to do this is to set and reset the `vblCount` field of the task record, as described in the previous paragraph.

The Process Manager treats VBL tasks slightly differently when your application quits or crashes than when it is switched out. If either the task record for a VBL task or the code

of the VBL task is located in your application partition, the Process Manager removes that task record from its VBL queue. (This is true for both slot-based and system-based VBL tasks.) Conversely, if both a VBL task record and the task itself are located in the system partition, the Process Manager doesn't remove the task record from its VBL queue when the application that installed them quits or crashes.

Note: Failure to remove VBL task records installed in the system partition from their queues can lead to a system crash if the VBL task is located in the system partition but accesses data in your application partition. Because the Process Manager deallocates your application partition when your application quits or crashes, the VBL task may attempt to access undefined data. The easiest way to avoid this problem is to patch the Process Manager's `ExitToShell` procedure so that it removes all VBL task records installed by your application.

- ***Using the Vertical Retrace Manager***

You can use the Vertical Retrace Manager to install VBL task records in and remove VBL task records from system-based or slot-based vertical retrace queues. To install a task record, you must first fill in some of its fields and then call either `VInstall` or `SlotVInstall`.
If it is to be executed more than once, a VBL task must access the task record and reset the value of the task record's `vblCount` field. To disable a task temporarily, you can simply set the `vblCount` field of its task record to 0. To remove a VBL task from its VBL queue, call `VRemove` if you installed the task by calling `VInstall` or call `SlotVRemove` if you installed the task by calling `SlotVInstall`. If your VBL task needs to access your application global variables, you can put the application's A5 value or the global variables themselves into the second field of a record whose first field contains the VBL task itself.

Installing a VBL Task

For any particular VBL task, you need to decide whether to install it as a system-based VBL task or as a slot-based VBL task. You need to install a task as a slot-based VBL task only if the execution of the task needs to be synchronized with the retrace rate of a particular external monitor. If the task performs no processing that is likely to affect the appearance of the screen or that depends on the state of an external monitor, it is probably safe to install the task as a system-based VBL task.

If you are uncertain whether to install a task as a system-based or as a slot-based VBL task, you should first install it as a system-based task (by calling `VInstall`). Then test your application on a modular Macintosh computer with an external monitor whose refresh rate is different from the refresh rate on a compact Macintosh computer (approximately 60.15 Hz). If any screen updating that occurs as a result of processing done by your VBL task has an unacceptable appearance, you probably need to install the task as a slot-based VBL task (by calling `SlotVInstall`).

The `InstallVBL` function defined in Listing below shows how to fill in a VBL task record and install it in the system-based VBL queue. It assumes that the task record `gMyVBLTask` is a global variable of type `VBLTask` and that you have already defined the procedure `DoVBL`, the actual VBL task. That procedure is subject to all of the usual limitations on VBL and other interrupt tasks. Also, if `DoVBL` is to be executed recurrently, it must reset the `vblCount` field of the task record each time it is executed.

Listing: Initializing and installing a task record

```
FUNCTION InstallVBL: OSErr;
CONST
kInterval = 6; {frequency in interrupts}
BEGIN
WITH gMyVBLTask DO {initialize the VBL task}
BEGIN
qType := ORD(vType); {set queue type}
vblAddr := @DoVBL; {set address of VBL task}
vblCount := kInterval; {set task frequency}
vblPhase := 0; {no phase}
END;
InstallVBL := VInstall(@gMyVBLTask);
END;
```

Accessing a Task Record at Interrupt Time

A repetitive VBL task must access its task record so that it can reset the `vblCount` field. The Vertical Retrace Manager decrements the `vblCount` field during each interrupt and executes the task when that field reaches 0. The task is removed from its queue if the value of the `vblCount` field is left at 0. When the Vertical Retrace Manager executes the VBL task, it places the address of the VBL task record into the A0 register.

Accessing Application Global Variables in a VBL Task

The Operating System stores the address of the boundary between the current application's global variables and its application parameters in the microprocessor's A5 register. For this reason, most compilers generate references to application global variables as offsets from the address contained in the A5 register. Therefore, if the value in register A5 does not point to the boundary between your application's global variables and its application parameters, your attempts to access your application's global variables will fail.

Because VBL tasks are interrupt routines, they might be executed when the value in the A5 register does not point to the A5 world of your application. As a result, if you want to access your application's global variables in a VBL task, you need to set the A5 register to its correct value when your VBL task begins executing and restore the previous value upon exit. Following code snippet shows how to install a VBL task.

LISTING:

```
OSErr InstallVBL ()
{
long anErr;
Str255 Temp;
Short MainSlotNumber,  mainDeviceRefNum;

gMyVBLRec.myVBLTask.qType = vType;
gMyVBLRec.myVBLTask.vblAddr = (ProcPtr) DoVBL;// address of
task
gMyVBLRec.myVBLTask.vblCount = kInterval; // Set the
interval
gMyVBLRec.vblA5 = (long) SetCurrentA5; // Save app's A5 in
structure

mainDeviceRefNum = (**GetMainDevice()).gdRefNum;
MainSlotNumber =
(**(AuxDCEHandle)GetDCtlEntry(mainDeviceRefNum)).dCtlSlot;

anErr = SlotVInstall ((QElemPtr) &gMyVBLRec.myVBLTask,
MainSlotNumber);
return anErr;
}

void DoVBL()
{
long curA5;
VBLRecPtr recPtr;

recPtr = (VBLRecPtr) GetVBLRec ();
curA5 = SetA5 (recPtr->vblA5);//read app A5 from structure
and save current value of A5
// ...  now that it's OK to access application variables,
do the task ...

fastDraw(recPtr);

 // Reset vblCount so that this procedure executes again
recPtr->myVBLTask.vblCount = kInterval;
curA5 = SetA5(curA5);
}
```

Installing a Persistent VBL Task

A **persistent VBL task** continues to be executed even when the Process Manager switches out the application that installed it and that application is no longer in control of the CPU. If you want to install a persistent system-based VBL task, you need to load its VBL task record into the system partition. (Slot-based VBL tasks are always persistent, no matter where you put the task record.)

/* Following is an example picked from Develop Issue 16 which looks at a VBL task, that works when the application is in the background */

```
VBLTask VBL;
#define kVBLInterval 1
#define isPersistent TRUE
long timeCount;
short slot;

void StartTimer (void)
{
OSErr err;
DefVideoRec VideoInfoRec;
THz   savedZone;

timeCount=0; /* init time count to zero */

/*  For a VBL task that operates when the application is in the  background (i.e., that's
persistent) we can simply create the  * UniversalProcPtr in the system heap. This causes
the Process  Manager to treat the code as though it were in the system heap  and the VBL
will always get executed. */

if (isPersistent)
 {
 savedZone = GetZone();
 SetZone(SystemZone());
 }
VBL.vblAddr = NewRoutineDescriptor((ProcPtr) DoTime,
 uppVBLProcInfo, GetCurrentISA());
err = MemError();
if (isPersistent)
 SetZone(savedZone);  /* Restore the application zone. */
if (VBL.vblAddr != nil)
 {
 GetVideoDefault(&VideoInfoRec);
 slot=VideoInfoRec.sdSlot;
 VBL.qType  = (short)vType;  /* init VBL task data structure */
 VBL.vblCount = kVBLInterval;  /* increment timecount every vertical retrace */
```

```
 VBL.vblPhase = 0;
 err= SlotVInstall((QElemPtr)&VBL,slot);
 } return;
}
```

/* The isPersistent Boolean variable controls whether the VBL functions persistently. If it's persistent, you can control where the memory is allocated by first setting the zone to the system zone (because NewRoutineDescriptor calls the Memory Manager to allocate memory for the routine descriptor). Here's the code for the VBL task: */
/* MyVBLProc */

```
pascal void DoTime (VBLTaskPtr theVBLTask)
{
theVBLTask->vblCount = kVBLInterval; /* re-execute the task every vertical
retrace */
timeCount++;   /* increment the timer variable */
return;
}
```

/* This very simple example alters only a global variable, but it illustrates two points. First, no complicated setup for global variables is required. For a 680x0 VBL task, messy saving and restoring of register A5 would be necessary for correct access to global variables. In the example, because the code resides in a code fragment, global variables are always accessible. Second, the procedure is called with a VBLTaskPtr parameter. For a 680x0 VBL task, a pointer to the VBLTask record resides in register A0 and requires special handling to get to the data from a high-level language. Because PowerPC code uses strict C calling conventions, the required data is passed as a standard parameter. Finally, of course, you have to remove the VBL task correctly: */

```
void StopTimer (void)
{
OSErr err;

THz savedZone;
err= SlotVRemove((QElemPtr)&VBL,slot);
if (VBL.vblAddr)
 {
 savedZone = GetZone();
 /* Make sure we're in the right zone. */
 SetZone(PtrZone((Ptr) VBL.vblAddr));
 DisposeRoutineDescriptor(VBL.vblAddr);
 SetZone(savedZone);
 } return;
}
/*
```

Although it may not be necessary to deallocate a VBL task created in the application heap, this code practices safe memory management by being sure the memory gets deallocated no matter where it is - in other words, whether it's persistent or not.
*/

NOTIFICATION MANAGER

About the Notification Manager

The Notification Manager provides a notification service. It allows software running in the background (or otherwise unseen by the user) to communicate information to the user.

In the same way, relatively invisible operations such as Time Manager tasks, VBL tasks, or device drivers might need to inform the user that some previously started routine is complete or perhaps that some error has rendered further execution undesirable or impossible.
In all these cases, the communication generally needs to occur in one direction only, from the background application (or task, or driver) to the user. The Notification Manager, included in system software versions 6.0 and later, allows you to alert the user by posting a **notification,** which is an audible or visible indication that your application (or other piece of software) requires the user's attention. You post a notification by issuing a **notification request** to the Notification Manager, which places your request in a queue. When your request reaches the top of the queue, the Notification Manager posts a notification to the user.

You can request three types of notification:

1. **Polite notification.** A small icon blinks, by periodically alternating with the Apple menu icon (the Apple logo) or the Application menu icon in the menu bar.
2. **Audible notification.** The Sound Manager plays the system alert sound or a sound contained in an `'snd '` resource.
3. **Alert notification.** An alert box containing a short message appears on the screen. The user must dismiss the alert box (by clicking the OK button) before foreground processing can continue.

These types of notification are not mutually exclusive; for example, an application can request both audible and alert notifications.

Note that the Notification Manager provides a one-way communications path from an application to the user. There is no provision for carrying information back from the user to the requesting application, although it is possible for the requesting application to determine if the notification was received. If you require this secondary communications link, do not use the Notification Manager. Instead, you should wait until the user switches your application into the foreground and then use standard means (for example, a dialog box) to obtain the required information.

Using the Notification Manager

To issue a notification to the user, you need to create a notification request and install it in the notification queue. The Notification Manager interprets the request and presents the notification to the user at the earliest possible time. After you have notified the user in the desired manner (that is, placed a diamond mark in the Application menu, added a small blinking icon to the menu bar, played a sound, or displayed an alert box), you might want the Notification Manager to call a response procedure. The response procedure is useful for determining that the user has indeed seen the notification or for reacting to the successful posting of the notification. Eventually, you need to remove the notification request from the notification queue; you can do this in the response procedure or when your application returns to the foreground.

Creating and deleting a Notification Request

Example code:

```
void NotifyUser()
{
   NMRec the_note;

   the_note.qType    = nmType;
   the_note.nmMark   = 0;
   the_note.nmIcon   = 0L; //Icon to be flashed
   the_note.nmSound  = 0L; //Handle to sound resource
   the_note.nmStr    = "\pHello, there"; //string to be shown in Alert box
   the_note.nmResp   = NewNMProc( myResponse); // memory leak, but we quit
anyway
   the_note.nmRefCon = false;

   if( NMInstall( &the_note) != noErr)
   {
      SysBeep( 9);
   }
}

pascal void myResponse( struct NMRec *theNMRec)
{
   (void)NMRemove( theNMRec);
   DisposeRoutineDescriptor( (UniversalProcPtr)theNMRec->nmResp);
}
```

DEFERRED TASK MANAGER

The Deferred Task manager on the macintosh gives you a way to delay some of your interupt processing until a less time critcal point. If you do something lengthy in your interupt routine, you'll mask other interupts from occuring. The Deferred Task manager allows you to specify a routine that should run after all other interupts have finished processing. This will allow the machine to function better, but still get you some time to process before normal applications get the CPU. Your task will still run at "interupt time" so no memory moving, etc... you have a really low priority so you wont mask other interupts.

Fine and great, the problem is that you dont have your own A5 inside this task. The parameter block passed does have space for a parameter, but you need asm to get at it. In addition you need to get the memory for the deferred task parameter block from somewhere, and you cant allocate it inside the interupt vector your making the call from. My interface here preallocates a block of paramater records, and arranges to restore the current A5 when your task is called. This saves you from thinking about exactly whats going on, and writing evil assembly.

SEGMENT MANAGER

About the Segment Manager

Your application’s executable code is stored in its resource fork as one or more resources of type `'CODE'`. These code resources are known as **segments** because the division of routines into code resources is controlled by segmentation directives you provide to your development system.
The Process Manager loads some code segments into memory when your application is launched. The Segment Manager loads other segments whenever you call any externally referenced routine contained in those segments. Both of these operations occur completely automatically and rely on information stored in your application’s jump table and in the individual code segments themselves.
The Segment Manager loads segments into relocatable, purgeable blocks in your application heap. A segment is locked when it is first read into memory and at any time thereafter when routines in the segment are executing. This locking prevents the block from being moved during heap compaction and from being purged during heap purging.
Although needed code segments are loaded into memory automatically, it is your application’s responsibility to unload any segments that are not currently being used. The segment Manager provides a single procedure, `UnloadSeg`, that you can call to unload a segment.

Code Segmentation

Your development system’s linker divides your application’s executable code into segments according to directives that you provide. The **main segment** contains the main

program. This segment is loaded into memory when your application starts to run and is never purged or unlocked as long as the application is running. The main event loop and other frequently needed small routines are generally stored in the main segment.

There are also some less obvious guidelines to follow when grouping routines into segments.

1. Put your main event loop into the main segment.
2. Put any routines that handle low-memory conditions into a locked segment (commonly the main segment). For example, if your application provides a grow-zone function, put that function into a locked segment.
3. Put any routines that execute at interrupt time, including VBL tasks and Time Manager tasks, into a locked segment (commonly the main segment).
4. Put into a separate segment any initialization routines that are executed exactly once at application startup time. Then unload that segment after those routines are executed.

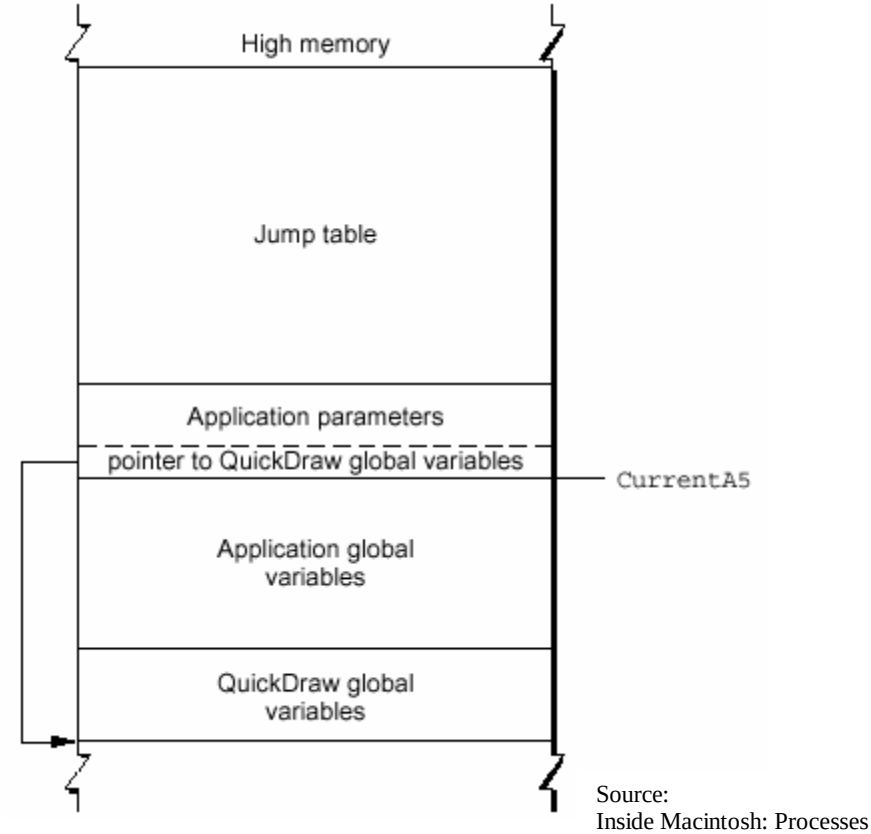
The Jump Table

This section describes how the Segment Manager works internally and is included for informational purposes only. You don't need this information to use the Segment Manager routine. Moreover, the information presented here might not be accurate for your development system.

The loading and unloading of segments are implemented through your application's **jump table,** an area of memory in your application's partition that contains one entry for every externally referenced routine in every code segment of your application. The location of the jump table is illustrated in Figure below

The jump table is accessed through the A5 register and is therefore part of your application's A5 world. The jump table is created by your development system's linker and is stored in segment 0 of your application (which is the `'CODE'` resource with an ID of 0). Segment 0 is a special segment created by the linker for every application; it contains information about the A5 world and the jump table. Figure 7-2 illustrates the structure of segment 0.

Fig: The location of the jump table

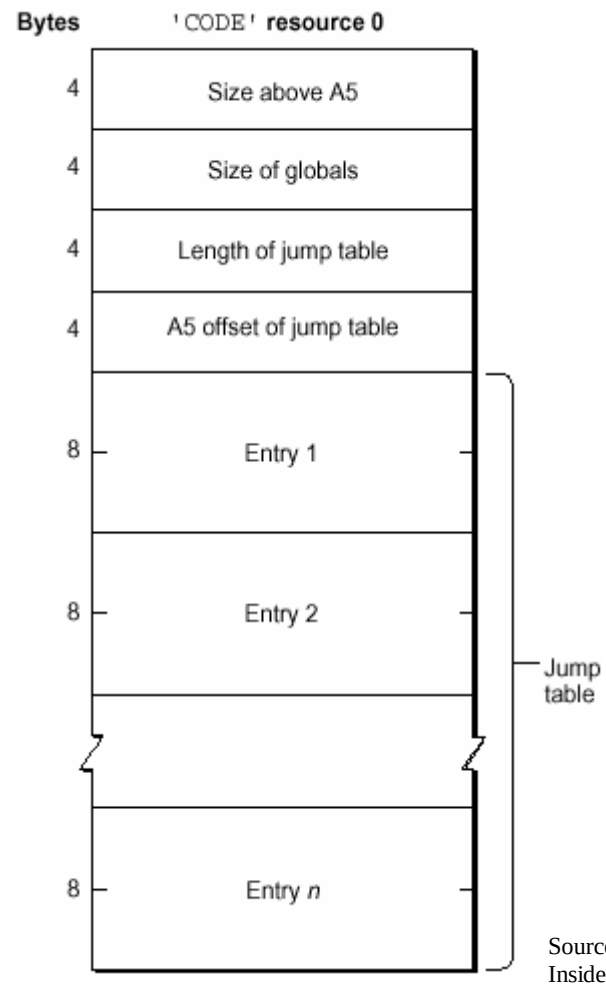


Source:
Inside Macintosh: Processes

Segment 0 consists of these elements:

1. Size above A5. The size (in bytes) from the location pointed to by register A5 to the upper end of the application space.
2. Size of globals. The size (in bytes) of the application global variables plus the QuickDraw global variables.
3. Length of jump table. The size (in bytes) of the jump table.
4. A5 offset of jump table. The offset (in bytes) to the jump table from the location pointed to by register A5. This offset is stored in the global variable `CurJTOffset`.
5. Jump table. A contiguous list of jump table entries.

Fig: The structure of segment 0

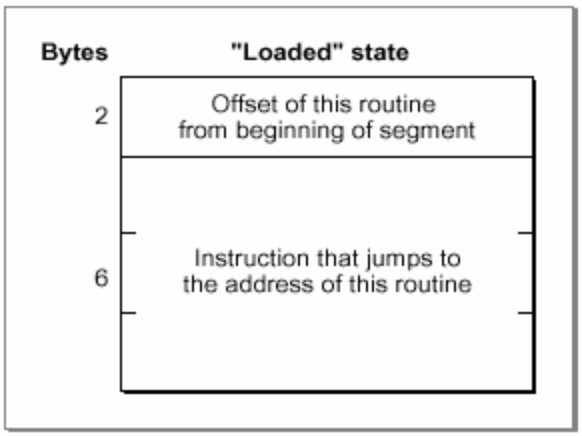


Source:
Inside Macintosh: Processes

When the MPW linker encounters a call to a routine in another code segment, it creates a **jump table entry** for that routine. (All entries for a particular segment are stored contiguously in the jump table.) The structure of a jump table entry varies according to whether the segment it references is loaded or unloaded.

The jump table refers to segments by segment numbers assigned by the linker. If the segment isn't loaded, the entry contains code that loads the segment. When a segment is unloaded, all its jump table entries are in the "unloaded" state. When a call to a routine in an unloaded segment is made, the code in the last 6 bytes of its jump table entry is executed. This code calls the `_LoadSeg` trap, which loads the segment into memory, transforms all of its jump table entries to a "loaded" state, and invokes the routine by executing the instruction in the last 6 bytes of its jump table entry. Figure 7-4 illustrates the format of a jump table entry in the "loaded" state.

Fig: Format of an MPW jump table entry when the segment is loaded



Source:
Inside Macintosh: Processes

Subsequent calls to the routine also execute this instruction. When you call `UnloadSeg`, it restores the jump table entries to their "unloaded" state. Notice that the last 6 bytes of the jump table entry are always executed; the effect depends on the state of the entry at the time.

Unloading Code Segments

You can use the `UnloadSeg` procedure to unload segments. To unload a particular segment, pass `UnloadSeg` the address of any externally referenced routine contained in that segment. For example, to unload the segment that contains the procedure `DoPrintFile`, execute this line of code:

```
UnloadSeg(@DoPrintFile);
```

You can call `UnloadSeg` at any time except when you are executing code contained in the segment to be unloaded.

Note: Before you unload a segment, make sure that your application no longer needs it. Never unload a segment that contains a completion routine or other interrupt task (such as a Time Manager task or VBL task) that might be executed after the segment is unloaded. Never unload a segment that contains routines in the current call chain.

The `UnloadSeg` procedure does not actually remove the segment from memory. Instead, it unlocks the segment, thereby making the segment relocatable and purgeable. This permits the Memory Manager to relocate or purge the segment if necessary to gain some space in the application heap.

Mindfire Solutions is an IT and software services company in India, providing expert capabilities to the global market. Mindfire possesses experience in multiple platforms, and has built a strong track record of delivery. Our continued focus on fundamental strengths in computer science has led to a unique technology-led position in software services.

To explore the potential of working with Mindfire, please drop us an email at info@mindfiresolutions.com. We will be glad to talk with you.

To know more about Mindfire Solutions, please visit us on www.mindfiresolutions.com