Time Manager

This chapter describes how you can use the Time Manager to schedule execution of a routine after a specified amount of time has elapsed. It includes information about the original Time Manager, as well as information about the revised Time Manager introduced in system software version 6.0.3 and the extended Time Manager introduced in system software version 7.0.

Because different versions of the Time Manager are available under different system software versions, your application may need to determine which version is available in its current environment. To do so, use the `Gestalt` function explained in the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*.

To use this chapter, you should be familiar with the Vertical Retrace Manager because it provides an alternative (and sometimes preferable) method for scheduling routines for future or periodic execution. For details on the Vertical Retrace Manager, see the chapter "Vertical Retrace Manager" in this book.

About the Time Manager

The Time Manager allows applications and other software to schedule routines for execution at a later time. By suitably defining the routine that is to be executed later, you can use the Time Manager to accomplish a wide range of time-related activities. For example, because a routine can reschedule itself for later execution, the Time Manager allows your application to perform periodic or repeated actions. You can use the Time Manager to

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

The Time Manager provides a hardware-independent method of performing these time-related tasks. In general, you should use the Time Manager instead of timing loops, which can vary in duration because they depend on clock speed and interrupt-handling speed.

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—that is, each succeeding Time Manager version is a functional superset of the previous one. However, code written for the extended Time Manager may not run properly with either the original or revised version. For this reason, it is sometimes important to know which Time Manager version is available on a specific computer.

You can use the `Gestalt` function to determine which version of the Time Manager is present. You should pass `Gestalt` the selector `gestaltTimeMgrVersion`.

```
CONST
   gestaltTimeMgrVersion      = 'tmgr';   {Time Manager version}
```

If `Gestalt` executes successfully, it returns one of three constants:

```
CONST
   gestaltStandardTimeMgr     = 1;        {original Time Manager}
   gestaltRevisedTimeMgr      = 2;        {revised Time Manager}
   gestaltExtendedTimeMgr     = 3;        {extended Time Manager}
```

If `Gestalt` returns an error, you should assume that the original Time Manager is present. The following sections describe the features of each version of the Time Manager.

The Original Time Manager

The Time Manager was first introduced with the Macintosh Plus ROMs (which are also used in Macintosh 512K enhanced models) and was intended for use internally by the Operating System. The original Time Manager allows delays as small as 1 millisecond, resulting in a maximum range of about 24 days.

To schedule a task for later execution, place an entry into the Time Manager queue and then activate it. All Time Manager routines manipulate elements of the Time Manager queue, which are stored in a **Time Manager task record.** The task record for the original Time Manager is defined by the `TMTask` data type.

```
TYPE TMTask =     {original and revised Time Manager task record}
   RECORD
      qLink:      QElemPtr;      {next queue entry}
      qType:      Integer;       {queue type}
      tmAddr:     ProcPtr;       {pointer to task}
      tmCount:    LongInt;       {reserved}
   END;
```

Of the four fields in this record, you need to fill in only the `tmAddr` field, which contains a pointer to the routine that is to be executed at some time in the future. The remaining fields are used internally by the Time Manager or are reserved by Apple Computer, Inc. However, you should set the `tmCount` field to 0 when you set up a task record.

The original Time Manager includes three routines:

- The `InsTime` procedure installs a task record into the Time Manager queue.
- The `PrimeTime` procedure schedules a previously queued task record for future execution.
- The `RmvTime` procedure removes a task record from the Time Manager queue.

Note that installing a request into the Time Manager queue (by calling the `InsTime` procedure) does not by itself schedule the specified routine for future execution. After you queue a request, you still need to activate (or **prime**) the request by specifying the desired delay until execution (by calling the `PrimeTime` procedure). Note also that the task record is not automatically removed from the Time Manager queue after the routine is executed. For this reason, you can reactivate the task by subsequent calls to `PrimeTime`; you do not have to reinstall the task record.

To remove a task record from the queue, you must call the `RmvTime` procedure. The `RmvTime` procedure removes a task record from the Time Manager queue whether or not that task was ever activated and whether or not its specified time delay has expired.

The Revised Time Manager

System software version 6.0.3 introduced a revised version of the Time Manager. This version provides better time resolution and more accurate measurements of elapsed time. You can represent time delays in the revised Time Manager as microseconds (µsec) as well as milliseconds (msec), with a finest resolution of 20 microseconds. The external programming interface did not change from the original to the revised Time Manager, although the revised version provides a means to distinguish microsecond delays from millisecond delays.

The revised Time Manager interprets negative time values (which were not formerly allowed) as negated microseconds. For example, a value of –50 is interpreted as a delay of 50 microseconds. Positive time values continue to represent milliseconds. When specified as microseconds, the maximum delay is about 35 minutes. When specified as milliseconds, the maximum delay is about 1 day. (This differs from the maximum delay in the original Time Manager because of the finer resolution of the revised Time Manager.) When passed to `PrimeTime`, the time value is converted to an internal form. For this reason, it makes no difference which unit you use if the delay falls within the ranges of both.

The revised Time Manager provides additional features. The principal change concerns the `tmCount` field of the Time Manager task record (previously reserved for use by Apple Computer, Inc.). When you remove an active task from the revised Time Manager's queue, any time remaining until the scheduled execution time is returned in the `tmCount` field. This change allows you to use the Time Manager to compute elapsed times (as explained in the section “Computing Elapsed Time” on page 3-14). In addition, the high-order bit of the `qType` field of the task record is used as a flag to indicate whether the task timer is active. The `InsTime` procedure initially clears this bit, and `PrimeTime` sets it. This bit is cleared when the time expires or when your application calls `RmvTime`.

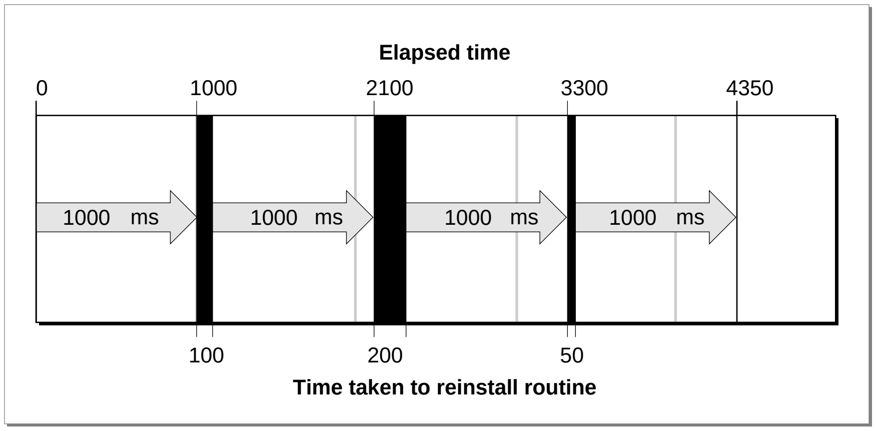
Although the revised Time Manager supports the specification of delay times in microseconds, you should use this feature primarily for the more accurate measurement of elapsed times. You should avoid specifying very small delay times as a way to execute a routine repeatedly at frequent intervals because this technique may use a considerable amount of processor time. The amount of processor time consumed by such timing services varies, depending largely on the performance of the CPU. With low-performance CPUs, little or no time may be left for other processing on the system (for instance, moving the mouse or running the application).

The Extended Time Manager

The extended Time Manager (available with system software version 7.0 and later) contains all the features of earlier Time Managers, with several extensions intended primarily to provide **drift-free, fixed-frequency** timing services. These services, which ensure that a routine is executed promptly after a specified delay, are important for sound and multimedia applications requiring precise timing and real-time synchronization among different events.

In the original and revised Time Managers, the value passed to `PrimeTime` indicates a delay that is relative to the current time (that is, the time when you execute `PrimeTime`). This presents problems if you attempt to implement a fixed-frequency timing service by having the task call `PrimeTime`. The problem is that the time consumed by the Time Manager and by any interrupt latency (which is not predictable) causes the task to be called at a slightly slower and unpredictable frequency, which drifts over time. In Figure 3-1, the desired fixed frequency of 1000 microseconds cannot be achieved because the Time Manager overhead and interrupt latency cause a small and unpredictable delay each time the task is reactivated.

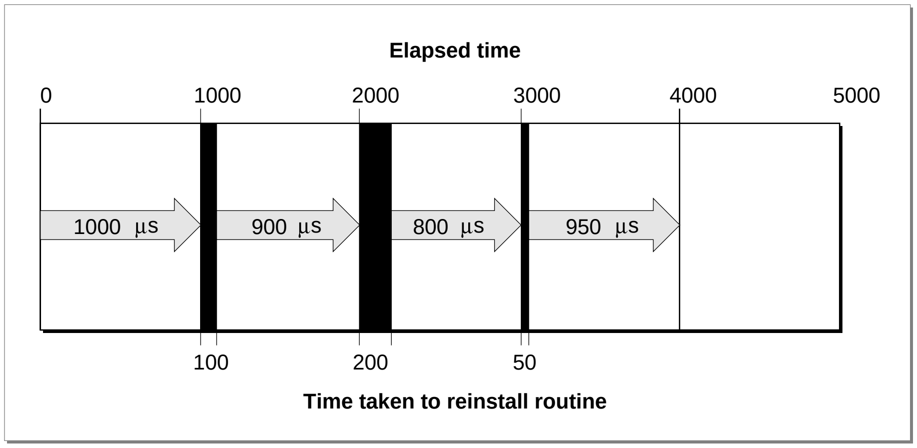
Figure 3-1 Original and revised Time Managers (drifting, unpredictable frequency)



The extended Time Manager solves this problem by allowing you to reinstall a task with an execution time that is relative to the time when the task last expired—not relative to the time when the task is reinstalled. The extended Time Manager compensates for the delay between the time when the task last expired and the time at which it was reinstalled, thereby providing a truly drift-free, fixed-frequency timing service.

For example, if your application needs to execute a routine periodically at 1-millisecond intervals, it can reactivate the existing Time Manager queue element by calling `PrimeTime` in the task with a specified delay of 1 millisecond. When the Time Manager receives this new execution request, it determines how long ago the previous `PrimeTime` task expired and then decrements the specified delay by that amount. For instance, if the previous task expired 100 microseconds ago, then the Time Manager installs the new task with a delay of 900 microseconds. This technique is illustrated in Figure 3-2.

Figure 3-2 The extended Time Manager (drift-free, fixed frequency)



The extended Time Manager implements these features by recognizing an expanded task record and providing a new procedure, `InsXTime`. The Time Manager task record for the extended Time Manager looks like this:

```
TYPE TMTask =        {extended Time Manager task record}
   RECORD
      qLink:         QElemPtr;   {next queue entry}
      qType:         Integer;    {queue type}
      tmAddr:        ProcPtr;    {pointer to task}
      tmCount:       LongInt;    {unused time}
      tmWakeUp:      LongInt;    {wakeup time}
      tmReserved:    LongInt;    {reserved for future use}
   END;
```

Once again, your application fills in the `tmAddr` field. You should set `tmWakeUp` and `tmReserved` to 0 when you first install an extended Time Manager task. The remaining fields are used internally by the Time Manager. As in the revised Time Manager, the `tmCount` field holds the time remaining until the scheduled execution of the task (this field is set by `RmvTime`).

The `tmWakeUp` field contains the time at which the Time Manager task specified by `tmAddr` was last executed (or 0 if it has not yet been executed). Its principal intended use is to provide drift-free, fixed-frequency timing services, which are available only when you use the extended Time Manager and only when you install Time Manager tasks by calling the new `InsXTime` procedure.

When your application installs an extended Time Manager task (by calling the `InsXTime` procedure), the behavior of the `PrimeTime` procedure changes slightly, as described earlier in this section. If the value of the `tmWakeUp` field is zero when `PrimeTime` is called, the delay parameter to `PrimeTime` is interpreted as relative to the current time (just as in the original Time Manager), but the Time Manager sets the `tmWakeUp` field to a nonzero value that indicates when the delay time should expire. When your application calls `PrimeTime` with a Time Manager task whose `tmWakeUp` field contains a nonzero value, the Time Manager interprets the specified delay as relative to the time that the last call to `PrimeTime` on this task was supposed to expire.

Note
Nonzero values in `tmWakeUp` are in a format that is used internally by the Time Manager. This format is subject to change. Your application should never use the value stored in this field and should either set it to 0 or leave it unchanged. When you first create an extended Time Manager task record, make sure that the value of the `tmWakeUp` field is 0; otherwise, the Time Manager may interpret it as a prior execution time. ◆

The extended Time Manager allows for a previously impossible situation that may lead to undesirable results. It is possible to call `PrimeTime` with an execution time that is in the past instead of in the future. (In the original and revised Time Managers, only future execution times are possible.) This situation arises when the value of the `tmWakeUp` field specifies a time in the past and you issue a new `PrimeTime` request with a delay value that is not large enough to cause the execution time to be in the future. This may occur when fixed, high-frequency execution is required and the time needed to process each execution, including the Time Manager overhead, is greater than the delay time between requests.

When your application issues a `PrimeTime` request with a `tmWakeUp` value that would result in a negative delay, the actual delay time is set to 0. The Time Manager updates the `tmWakeUp` field to indicate the time when the task should have been performed (in the past). Because the actual delay time is set to 0, the task is executed immediately. If your application continually issues `PrimeTime` requests for times in the past, the Time Manager and the `tmAddr` tasks consume all of the processor cycles. As a result, no time is left for the application to run. Because this situation is a function of processor speed, you should ensure compatibility by using the slowest processors to test applications that use extended Time Manager features. Another solution to this problem is to vary the wakeup frequency according to the processing power of the computer.

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 (see the section "Computing Elapsed Time" on page 3-14), 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.

To use the Time Manager for periodic execution of a task, simply have the routine pointed to by `tmAddr` call `PrimeTime` again. This technique is illustrated in the section "Performing Periodic Tasks" on page 3-13. Similarly, you can execute a Time Manager task a specific number of times by keeping a count of the number of times the task has been called. In cases where the task needs access to your application's global variables (such as a count variable), make sure that the A5 register points to your application's global variables when the task is executed and that A5 is restored to its original value when your task exits. A technique for this purpose is illustrated in "Using Application Global Variables in Tasks" on page 3-11.

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 3-1 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. (The extended Time Manager task record, `gTMTask`, is a global variable of type `TMTask`.) After the specified delay has elapsed (in this case, 2000 milliseconds, or 2 seconds), the procedure `MyTask` runs.

In cases where no task is to run after the specified delay has elapsed, you should set the `tmAddr` field to `NIL`. To determine if the time has expired, you can check the task-active bit in the `qType` field.

Avoid calling `PrimeTime` with a Time Manager task record that has not yet expired, because the results are unpredictable. If you wish to reactivate a prior unexpired request in the Time Manager queue and specify a different delay, call `RmvTime` to cancel the prior request, then call `InsTime` to reinstall the timer task, and finally call `PrimeTime` to reschedule the task. Note, however, that it is possible and sometimes desirable to call `PrimeTime` with a Time Manager task that you want to reactivate, because the timer will have expired before the task is called.

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. The situation here is unlike the situation with Notification Manager tasks or Sound Manager callback routines (both of which provide an easy way to pass the address of the application's A5 world to the task), but it is similar to the situation with vertical retrace tasks.

Note
For a more detailed discussion of setting and restoring your application's A5 world, see the chapter "Memory Management Utilities" in *Inside Macintosh: Memory.* ◆

One way to gain access to the global variables of the application that launched a Time Manager task is to pass to `InsTime` (or `InsXTime`) and `PrimeTime` the address of a structure, the first segment of which is simply the corresponding Time Manager task record and the remaining segment of which contains the address of the application's A5 world. For example, you can define a new data structure, a Time Manager information record, as follows:

```
TYPE TMInfo =              {Time Manager information record}
   RECORD
      myTMTask:   TMTask;  {original and revised TM task record}
      tmRefCon:   LongInt; {space to pass address of A5 world}
   END;

TMInfoPtr = ^TMInfo;
```

Note
The `TMInfo` record defined above is intended for use with the extended Time Manager. ◆

Then you can install and activate your Time Manager task as illustrated in Listing 3-2. The global variable `gTMInfo` is an information record of type `TMInfo`.

Listing 3-2 Passing the address of the application's A5 world to a Time Manager task

```
PROCEDURE InstallTMTask;
CONST
   kDelay = 2000;                       {delay value}
BEGIN
   gTMInfo.myTMTask.tmAddr := @MyTask; {get address of task}
   gTMInfo.myTMTask.tmWakeUp := 0;     {initialize tmWakeUp}
   gTMInfo.myTMTask.tmReserved := 0;   {initialize tmReserved}
   gTMInfo.tmRefCon := SetCurrentA5;   {store address of A5 }
                                       { world}
   InsTime(@gTMInfo);                  {install the info record}
   PrimeTime(@gTMInfo, kDelay);        {activate the info record}
END;
```

With the revised and extended Time Managers, the task is called with register A1 containing the address passed to `InsTime` (or `InsXTime`) and `PrimeTime`. Thus, the Time Manager task simply needs to retrieve the `TMInfo` record and extract the appropriate value of the application's A5 world. Listing 3-3 illustrates a task definition for this purpose.

Listing 3-3 Defining a Time Manager task that can manipulate global variables

```
FUNCTION GetTMInfo: TMInfoPtr;
   INLINE $2E89;                       {MOVE.L A1,(SP)}

PROCEDURE MyTask;
VAR
   oldA5:   LongInt;                   {A5 when task is called}
   recPtr:  TMInfoPtr;
BEGIN
   recPtr := GetTMInfo;                {first get your record}
   oldA5 := SetA5(recPtr^.tmRefCon);   {set A5 to app's A5 world}

   {Do something with the application's globals here.}

   oldA5 := SetA5(oldA5);              {restore original A5 }
                                       { and ignore result}
END;
```

This technique works primarily because the revised and extended Time Managers do not care if the record whose address is passed to `InsTime` (or `InsXTime`) and `PrimeTime` is larger than expected. If you use this technique, however, be sure to retrieve the address of the task record from register A1 as soon as you enter the Time Manager task (because some compilers generate code that uses registers A0 and A1 to dereference structures).

IMPORTANT

You cannot use the technique illustrated in Listing 3-3 with the original Time Manager because it does not pass the address of the task record in register A1. To gain access to your application's global variables when using the original Time Manager, you would need to store your application's A5 value in one of the application's code segments (in particular, in the code segment that contains the Time Manager task). This technique involves the use of self-modifying code segments and is not in general recommended. Applications that attempt to modify their own `'CODE'` resources may crash in operating environments (for example, A/UX) that restrict an application's access to its own code segments. ▲

Performing Periodic Tasks

One way to install a periodic Time Manager task is to have the task reactivate itself. Because the task record is already inserted into the Time Manager task queue, the task can simply call `PrimeTime` to reactivate itself. To call `PrimeTime`, however, the task needs to know the address of the corresponding task record. In the revised and extended Time Managers, the task record's address is placed into register A1 when the task is called. Listing 3-4 illustrates how the task can reactivate itself by retrieving the address in register A1 and passing that address to `PrimeTime`.

Listing 3-4 Defining a periodic Time Manager task

```
FUNCTION GetTMInfo: TMInfoPtr;
   INLINE $2E89;                 {MOVE.L A1,(SP)}

PROCEDURE MyTask;                {for revised and extended TMs}
VAR
   recPtr:        TMInfoPtr;
CONST
   kDelay = 2000;                {delay value}
BEGIN
   recPtr := GetTMInfo;          {first get your own address}

   {Do something here.}

   PrimeTime(QElemPtr(recPtr), kDelay);
END;
```

IMPORTANT
You cannot use the technique illustrated in Listing 3-4 with the original Time Manager because it does not pass the address of the task record in register A1. ▲

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`.

To obtain the most accurate results, you should calculate all times in microseconds (in which case the `tmCount` field of the task record has a range of about 35 minutes). To get an exact measurement, compute the overhead associated with calling the Time Manager and subtract it from the preliminary result. Listing 3-5 illustrates a technique for calculating that overhead.

Listing 3-5 Calculating the time required to install and activate a Time Manager task

```
FUNCTION TMOverhead: LongInt;
VAR
   myTask:     TMTask;     {a Time Manager task record}
   myStart:    LongInt;    {initial delay passed to PrimeTime}
   myElapsed:  LongInt;    {elapsed time}
BEGIN
   myStart := -(MAXLONG);  {use a large negative number}

   WITH myTask DO          {set up the task record}
      BEGIN
         tmAddr := NIL;    {no task to execute}
         tmWakeUp := 0;
         tmReserved := 0;
      END;

   InsTime(@myTask);             {install the task}
   PrimeTime(@myTask, myStart);  {prime the task}
   RmvTime(@myTask);             {remove the task}

   myElapsed := myStart - myTask.tmCount;
   TMOverhead := -(myElapsed);   {the elapsed time}
END;
```

The `TMOverhead` function defined in Listing 3-5 sets up a Time Manager task record with no completion routine. In this case, you can allocate the task record as a local variable on the stack because the task record is removed before the function exits. Then the task is activated by calling `PrimeTime` with a very large negative value. (The negative value represents microseconds.) Immediately the task is deactivated and removed. The function determines the elapsed time by subtracting the value returned in the `tmCount` field of the task record from the original delay time.

Listing 3-6 illustrates how to measure the elapsed time associated with a request to delay program execution by 1 tick.

Listing 3-6 Calculating the time consumed by a 1-tick delay

```
FUNCTION CheckDelayTiming: LongInt;
VAR
   myTask:     TMTask;     {a Time Manager task record}
   myStart:    LongInt;    {initial delay passed to PrimeTime}
   myEnd:      LongInt;    {unused time}
   myTicks:    LongInt;    {ignored; needed for Delay procedure}
   myElapsed:  LongInt;    {elapsed time}
BEGIN
   myStart := -(MAXLONG);  {use a large negative number}

   WITH myTask DO          {set up the task record}
      BEGIN
         tmAddr := NIL;    {no task to execute}
         tmWakeUp := 0;
         tmReserved := 0;
      END;

   InsTime(@myTask);             {install the task}
   PrimeTime(@myTask, myStart);  {prime the task}
   Delay(1, myTicks);            {delay for 1 tick}
   RmvTime(@myTask);             {remove the task}

   myEnd := myTask.tmCount;      {get unused part of myStart}

   IF myEnd < 0 THEN             {myEnd is in microseconds}
      myElapsed := ABS(myStart - myEnd) - TMOverhead
   ELSE                          {myEnd is in milliseconds}
      myElapsed := ABS(myStart + (myEnd * 1000)) - TMOverhead;

   CheckDelayTiming := myElapsed;{the elapsed time}
END;
```

The `CheckDelayTiming` function is similar to the `TMOverhead` function except that the section of code to be timed occurs between the calls to `PrimeTime` and `RmvTime`. The `CheckDelayTiming` function simply times a call to the `Delay` procedure with a 1-tick delay time. Once `Delay` has completed and the task record has been deactivated, `CheckDelayTiming` determines whether the unused time returned in the `tmCount` field represents microseconds or milliseconds. The value returned by `CheckDelayTiming` is in microseconds.

Time Manager Reference

This section describes the data structures and routines that are specific to the Time Manager. It also describes the application-defined Time Manager task procedure whose address is specified in the task record.

Data Structures

All Time Manager routines require that you pass the address of a Time Manager task record, defined by the `TMTask` data type. If you are using the original or revised Time Manager, the task record has this structure:

```
TYPE TMTask =     {original and revised Time Manager task record}
   RECORD
      qLink:      QElemPtr;      {next queue entry}
      qType:      Integer;       {queue type}
      tmAddr:     ProcPtr;       {pointer to task}
      tmCount:    LongInt;       {reserved}
   END;
```

Field descriptions

`qLink`	A pointer to the next element in the Time Manager queue. This field is used internally by the Time Manager.
`qType`	The type of queue. The Time Manager automatically sets this field to the appropriate value. In the revised Time Manager, the high-order bit of this field is a flag that indicates whether the task is active.
`tmAddr`	A pointer to the routine to be executed after the delay specified in a call to `PrimeTime`.
`tmCount`	Reserved in the original Time Manager. In the revised Time Manager, the amount of time remaining until the task's scheduled execution time; this field is valid only after you call `RmvTime` with a task that has not yet executed.

If you are using the extended Time Manager, the task record has this structure:

```
TYPE TMTask =        {extended Time Manager task record}
   RECORD
      qLink:         QElemPtr;   {next queue entry}
      qType:         Integer;    {queue type}
      tmAddr:        ProcPtr;    {pointer to task}
      tmCount:       LongInt;    {unused time}
      tmWakeUp:       LongInt;    {wakeup time}
      tmReserved:     LongInt;    {reserved for future use}
   END;
```

Field descriptions

`qLink`	A pointer to the next element in the Time Manager queue. This field is used internally by the Time Manager.
`qType`	The type of queue. The Time Manager automatically sets this field to the appropriate value. The high-order bit of this field is a flag that indicates whether the task is active.
`tmAddr`	A pointer to the routine that is to be executed after the delay specified in a call to `PrimeTime`.
`tmCount`	The time remaining until the task's scheduled execution time. This field is valid only after you call `RmvTime` with a task that has not yet executed.
`tmWakeUp`	The time when the task specified in the `tmAddr` field was last executed. This field is used internally by the Time Manager. You should set it to 0 when you first install a task record.
`tmReserved`	Reserved.

Time Manager Routines

You can insert a task record into the Time Manager's queue by calling `InsTime` or `InsXTime`. Use `InsXTime` only if you wish to use the drift-free, fixed-frequency timing services of the extended Time Manager; use `InsTime` in all other cases. After you have queued a task record, you can activate it by calling `PrimeTime`. You can remove a task record from the queue by calling `RmvTime`.

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.

With the revised and extended Time Managers, you can set `tmAddr` to `NIL` if you do not want a task to execute when the delay passed to `PrimeTime` expires. Also, the revised Time Manager resets the high-order bit of the `qType` field to 0 when you call `InsTime`.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `InsTime` are

Registers on entry

A0	Address of the task record

Registers on exit

D0	Result code

RESULT CODES

`noErr`	0	No error

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`.

With the extended Time Manager, you can set `tmAddr` to `NIL` if you do not want a task to execute when the delay passed to `PrimeTime` expires. Also, `InsXTime` resets the high-order bit of the `qType` field to 0.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `InsXTime` are

Registers on entry

A0	Address of the task record

Registers on exit

D0	Result code

RESULT CODES

`noErr`	0	No error

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.

In the revised and extended Time Managers, `PrimeTime` sets the high-order bit of the `qType` field to 1. In addition, any value of the `count` parameter that exceeds the maximum millisecond delay is reduced to the maximum. If you stop an unexpired task (by calling `RmvTime`) and then reinstall it (by calling `InsXTime`), you can continue the previous delay by calling `PrimeTime` with the `count` parameter set to 0.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `PrimeTime` are

Registers on entry

A0	Address of the task record
D0	Specified delay time (long)

Registers on exit

D0	Result code

RESULT CODES

`noErr`	0	No error

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.

DESCRIPTION

The `RmvTime` procedure removes the Time Manager task record specified by `tmTaskPtr` from the Time Manager queue. In both the revised and extended Time Managers, if the specified task record is active (that is, if it has been activated but the specified time has not yet elapsed), the `tmCount` field of the task record returns the amount of time remaining. To provide the greatest accuracy, the unused time is reported as negated microseconds if that value is small enough to fit into the `tmCount` field (even if the delay was originally specified in milliseconds); otherwise, the unused time is reported in positive milliseconds. If the time has already expired, `tmCount` contains 0.

In the revised and extended Time Managers, `RmvTime` sets the high-order bit of the `qType` field to 0.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `RmvTime` are

Registers on entry

A0	Address of the task record

Registers on exit

D0	Result code

RESULT CODES

`noErr`	0	No error

Application-Defined Routine

The Time Manager allows your software to install an application-defined routine that is executed after a specified delay.

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`.

SPECIAL CONSIDERATIONS

Because the task procedure is executed at interrupt time, it should not allocate, move, or purge memory (either directly or indirectly) and should not depend on the validity of handles to unlocked blocks.

ASSEMBLY-LANGUAGE INFORMATION

In the revised and extended Time Managers, when the task procedure is called, register A1 contains a pointer to the Time Manager task record associated with that procedure.

A task procedure must preserve all registers other than A0–A3 and D0–D3.

SEE ALSO

See the section “Using Application Global Variables in Tasks” on page 3-11 for instructions on how to access your application’s global variables from within a task procedure. See “Performing Periodic Tasks” on page 3-13 for instructions on how to define a periodic task procedure.

Summary of the Time Manager

Pascal Summary

Constants

```
CONST
   {Gestalt selector}
   gestaltTimeMgrVersion      = 'tmgr';   {Time Manager version}

   {values returned by Gestalt}
   gestaltStandardTimeMgr     = 1;        {original Time Manager}
   gestaltRevisedTimeMgr      = 2;        {revised Time Manager}
   gestaltExtendedTimeMgr     = 3;        {extended Time Manager}
```

Data Types

Original and Revised Time Manager Task Record

```
TYPE TMTask =
   RECORD
      qLink:         QElemPtr;      {next queue entry}
      qType:         Integer;       {queue type}
      tmAddr:        ProcPtr;       {pointer to task}
      tmCount:       LongInt;       {reserved}
   END;
```

Extended Time Manager Task Record

```
TYPE TMTask =
   RECORD
      qLink:         QElemPtr;      {next queue entry}
      qType:         Integer;       {queue type}
      tmAddr:        ProcPtr;       {pointer to task}
      tmCount:       LongInt;       {unused time}
      tmWakeUp:      LongInt;       {wakeup time}
      tmReserved:    LongInt;       {reserved for future use}
   END;
```

```
TMTaskPtr = ^TMTask;
```

Time Manager Routines

```
PROCEDURE InsTime           (tmTaskPtr: QElemPtr);
PROCEDURE InsXTime          (tmTaskPtr: QElemPtr);
PROCEDURE PrimeTime         (tmTaskPtr: QElemPtr; count: LongInt);
PROCEDURE RmvTime           (tmTaskPtr: QElemPtr);
```

Application-Defined Routine

```
PROCEDURE MyTimeTask;
```

C Summary

Constants

```
/*Gestalt selector*/
#define gestaltTimeMgrVersion    'tmgr'      /*Time Manager version*/

/*values returned by Gestalt*/
#define gestaltStandardTimeMgr   1           /*original Time Manager*/
#define gestaltRevisedTimeMgr    2           /*revised Time Manager*/
#define gestaltExtendedTimeMgr   3           /*extended Time Manager*/
```

Data Types

```
typedef pascal void (*TimerProcPtr)(void);
```

Original and Revised Time Manager Task Record

```
struct TMTask {
      QElemPtr       qLink;         /*next queue entry*/
      short          qType;         /*queue type*/
      TimerProcPtr   tmAddr;        /*pointer to task*/
      long           tmCount;       /*reserved*/
};
```

Extended Time Manager Task Record

```
struct TMTask {
      QElemPtr       qLink;         /*next queue entry*/
      short          qType;         /*queue type*/
      TimerProcPtr   tmAddr;        /*pointer to task*/
      long           tmCount;       /*unused time*/
      long           tmWakeUp;      /*wakeup time*/
      long           tmReserved;    /*reserved for future use*/
};

typedef struct TMTask TMTask;
typedef TMTask *TMTaskPtr;
```

Time Manager Routines

```
pascal void InsTime          (QElemPtr tmTaskPtr);
pascal void InsXTime         (QElemPtr tmTaskPtr);
pascal void PrimeTime        (QElemPtr tmTaskPtr, long count);
pascal void RmvTime          (QElemPtr tmTaskPtr);
```

Application-Defined Routine

```
pascal void MyTimeTask       (void);
```

Assembly-Language Summary

Data Structures

Structure of Original and Revised Time Manager Queue Entry

0	`qLink`	long	pointer to next queue entry
4	`qType`	word	queue type
6	`tmAddr`	long	pointer to task
10	`tmCount`	long	unused time; returned to caller

Structure of Extended Time Manager Queue Entry

0	`qLink`	long	pointer to next queue entry
4	`qType`	word	queue type
6	`tmAddr`	long	pointer to task
10	`tmCount`	long	unused time; returned to caller
14	`tmWakeUp`	long	wakeup time; used internally by the Time Manager
18	`tmReserved`	long	reserved for future use

Result Codes

`noErr`	0	No error