Date - March 4, 1986

Author - Cheryl Ewy

Subject - Event Manager ERS

Document Version Number - 00:40

<u>Revision History</u>

00:00	(11-04-85)	Initial Release
00:10	(11-25-85)	GetKeys call removed Version, DoWindows and SetSwitch calls added StartUp, Button, StillDown and WaitMouseUp calls modified Desk Accessory event added Support for two buttons added Switch event handling changed
00:20	(01-27-86)	GetEvQHdr call removed ChangeFlag added to modifiers StartUp call modified BootInit, StartUp, ShutDown and Version calls renamed Alarm handling removed Journaling information added
00:30	(02-14-86)	EMReset call added EMStartUp call modified Call numbers changed
00:40	(03-04-86)	EMActive call added EMReset call modified Error numbers added

OVERVIEW

This document describes the Event Manager which allows applications to monitor the user's actions, such as those involving the mouse, keyboard, and keypad. The Event Manager is also used by other parts of the Toolbox; for instance, the Window Manager uses events to coordinate the ordering and display of windows on the screen. Although the Event Manager is a single Toolset, it is conceptually divided into two parts; the Operating System Event Manager and the Toolbox Event Manager.

The Operating System Event Manager detects low-level, hardware-related events such as mouse button presses and keystrokes. It stores information about these events in the event queue and provides routines that access the queue.

The Operating System Event Manager also allows an application to:

- post its own events into the event queue
- remove events from the event queue
- set the system event mask, to control which types of events get posted into the queue

The Toolbox Event Manager calls the Operating System Event Manager to retrieve events from the event queue. In addition, it reports window and switch events, which aren't kept in the queue. The Toolbox Event Manager is the application's link to its user. A typical event-driven application decides what to do from moment to moment by asking the Toolbox Event Manager for events and responding to them one by one in whatever way is appropriate.

The Toolbox Event Manager also allows an application to:

- restrict some of the routines to apply only to certain event types
- directly read the current state of the mouse button
- monitor the location of the mouse
- find out how much time has elapsed since the system last started up

In general, events are collected from a variety of sources and reported to the application on demand, one at a time. Events aren't necessarily reported in the order they occurred since some have a higher priority than others.

Note - In the remainder of this document, "OSEM" denotes the Operating System Event Manager and "TBEM" denotes the Toolbox Event Manager.

EVENT TYPES

Events are of various types. Some report actions by the user; others are generated by the Window Manager, the Control Manager, device drivers, or the application itself for its own purposes. Some events are handled by the system before the application ever sees them; others are left for the application to handle. The event types are as follows:

Mouse Events

Pressing the mouse button generates a **mouse-down event**, while releasing the button generates a **mouse-up event**. Movements of the mouse cause the cursor position to be updated but are not reported as events. Whenever an event is posted, the location of the mouse at that time is reported in a field of the event record. The application can obtain the current mouse position if needed by calling the TBEM routine GetMouse. Because relative pointing devices such as joysticks must also be supported, the Event Manager differentiates between button 0 and button 1.

Keyboard Events

The character keys on the keyboard and keypad generate **key-down events** when pressed; this includes all keys except Shift, Caps Lock, Control, Option and Open-Apple, which are called modifier keys. Modifier keys are treated differently and generate no keyboard events of their own. Whenever an event is posted, the state of the modifier keys is reported in a field of the event record.

The character keys on the keyboard and keypad also generate **auto-key events** when held down. Two different time intervals are associated with auto-key events. The first auto-key event is generated after a certain initial delay has elapsed since the key was originally pressed; this is called the delay to repeat. Subsequent auto-key events are then generated each time a certain repeat interval has elapsed since the last such event; this is called the repeat speed. The user can change these values with the Control Panel.

Window Events

The Window Manager generates events to coordinate the display of windows on the screen. **Activate events** are generated whenever an inactive window becomes active or an active window becomes inactive. They generally occur in pairs (that is, one window is deactivated and then another is activated).

Update events occur when all or part of a window's contents need to be drawn or redrawn, usually as a result of the user opening, closing, activating, or moving a window.

Other Events

A **device driver event** may be generated by device drivers in certain situations; for example, a driver might be set up to report an event when its transmission of data is interrupted. Device driver events are placed in the event queue with the OSEM procedure PostEvent.

An application can define as many as four **application events** of its own and use them for any desired purpose. Application-defined events are placed in the event queue with the OSEM procedure PostEvent.

A **switch event** is generated by the Control Manager whenever a button-down event has occured on the switch control.

A **desk accessory event** is generated whenever the user enters the special keystoke to invoke a "classic" deck accessory (currently control-open apple-escape).

A **null event** is returned by the Event Manager if it has no other events to report.

PRIORITY OF EVENTS

Events are retrieved from the event queue in the order they were originally posted. However, the way that various types of events are generated and detected causes some events to have higher priority than others. Also, not all events are kept in the event queue. Furthermore, when an application asks the TBEM for an event, it can specify particular types that are of interest which can cause some events to be passed over in favor of others that were actually posted later.

The TBEM always returns the highest-priority event available of the requested types. The priority ranking is as follows:

1. activate (window becoming inactive before window becoming active)
2. switch
3. mouse-down, mouse-up, key-down, auto-key, device driver, application-defined, desk accessory (all in FIFO order)
4. update (in front-to-back order of windows)

Activate events take priority over all others; they're detected in a special way, and are never actually placed in the event queue. The TBEM checks for pending activate events before looking in the event queue, so it will always return such an event if one is available. Because of the special way activate events are detected, there can never be more than two such events pending at the same time; at most there will be one for a window becoming inactive followed by another for a window becoming active.

Next in priority are switch events which are generated by the Control Manager and are also not placed in the event queue. If no activate events are pending, the TBEM checks for a switch event before looking in the event queue. If a switch event is available, the TBEM then checks to see if any update events are pending, and if so, it returns the update event to the application. The switch event is not returned to the application until there are no pending update events. This is to insure that all of the windows are updated before the application is switched.

Category 3 includes most of the event types. Within this category, events are retrieved from the queue in the order they were posted.

Next in priority are update events. Like activate and switch events, these are not placed in the event queue, but are detected in another way. If no higher-priority event is available, the TBEM checks for windows whose contents need to be drawn. If it finds one, it returns an update event for that window. Windows are checked in the order in which they're displayed on the screen, from front to back, so if two or more windows need to be updated, an update event will be returned for the frontmost such window.

Finally, if no other event is available, the TBEM returns a null event.

> **Note:** If the queue should become full, the OSEM will begin discarding old events to make room for new ones as they're posted. The events discarded are always the oldest ones in the queue.

EVENT RECORDS

Every event is represented internally by an event record containing all pertinent information about that event. The event record includes the following information:

- the type of event
- the time the event was posted (in ticks since system startup)
- the location of the mouse at the time the event was posted (in global coordinates)

- the state of the mouse buttons and modifier keys at the time the event was posted
- any additional information required for a particular type of event, such as which key the user pressed or which window is being activated

Every event, including null events, has an event record containing this information.

Event records are defined as follows:

```
what          INTEGER    {event code}
message       LONGINT    {event message}
when          LONGINT    {ticks since startup}
where         Point      {mouse location}
modifiers     INTEGER    {modifier flags}
```

The when field contains the number of ticks since the system last started up, and the where field gives the location of the mouse, in global coordinates, at the time the event was posted. The other three fields are described below.

Event Code

The what field of an event record contains an event code identifying the type of the event. The event codes are assigned as follows:

0 - null event
1 - mouse down event
2 - mouse up event
3 - key down event
4 - undefined
5 - auto-key event
6 - update event
7 - undefined
8 - activate event
9 - switch event
10 - desk accessory event
11 - device driver event
12 - application-defined event
13 - application-defined event
14 - application-defined event
15 - application-defined event

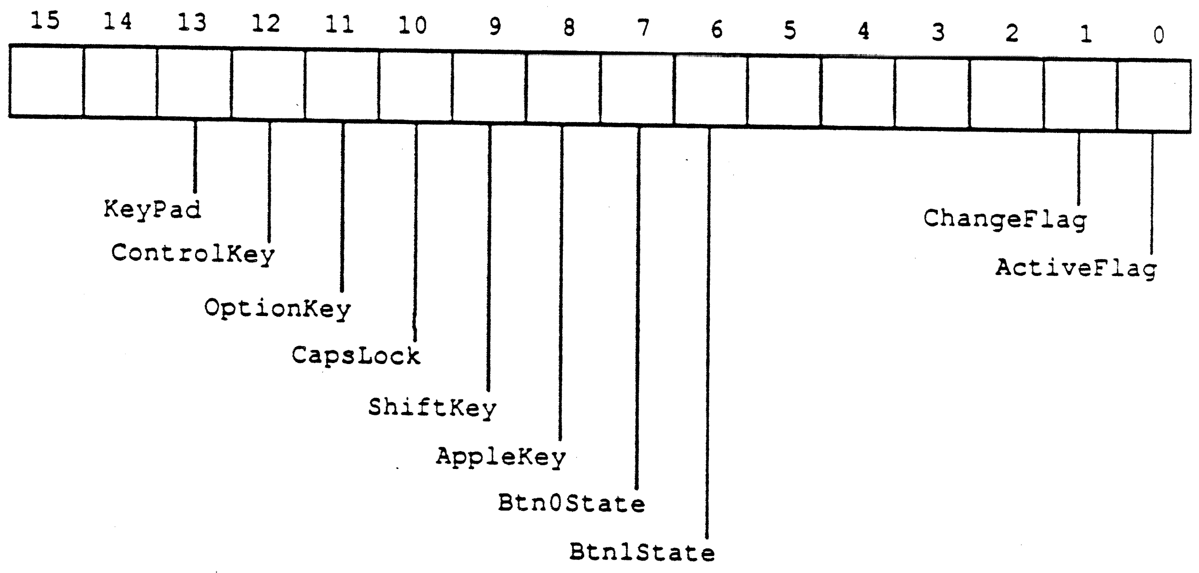
Event Message

The message field of an event record contains the event message, which conveys additional information about the event. The nature of this information depends on the event type, as shown in the following table.

Event type	**Event message**
Key-down	ASCII character code in low-order byte
Auto-key	ASCII character code in low-order byte
Activate	Pointer to window
Update	Pointer to window
Mouse-down	Button number (0 or 1) in low-order word
Mouse-up	Button number (0 or 1) in low-order word
Device driver	Defined by the device driver
Application	Defined by the application
Switch	Undefined
Desk Accessory	Undefined
Null	Undefined

Modifier Flags

The modifiers field of an event record contains further information about activate events and the state of the modifier keys and mouse buttons at the time the event was posted, as shown below. The application might look at this field to find out, for instance, whether the Open-Apple key was down when a mouse-down event was posted (which could affect the way objects are selected) or when a key-down event was posted (which could mean the user is choosing a menu item by typing its keyboard equivalent).

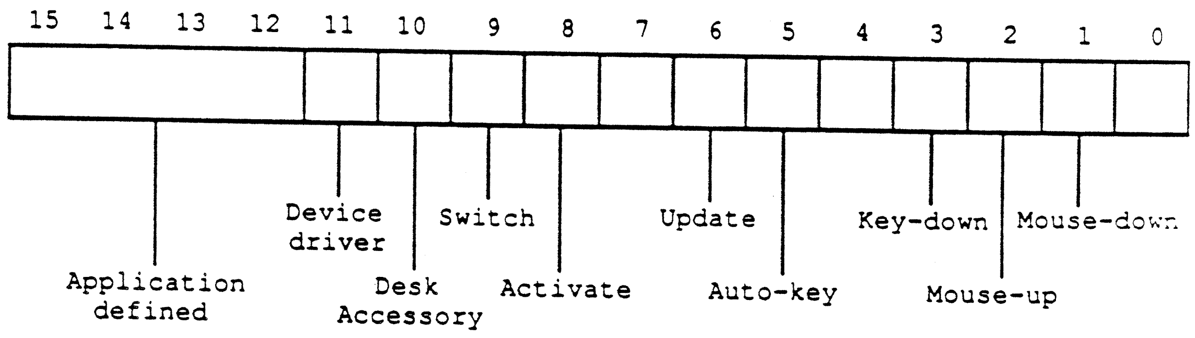


The ActiveFlag and ChangeFlag bits give further information about activate events. The ActiveFlag bit is set to 1 if the window pointed to by the event message is being activated, or 0 if the window is being deactivated. The ChangeFlag bit is set to 1 if the active window is changing from an application window to a system window or vice versa. Otherwise, it's set to 0. The KeyPad bit gives further information about key-down events; it's set to 1 if the key pressed was on the keypad, or 0 if the key pressed was on the keyboard. The remaining bits indicate the state of the mouse button and modifier keys. Note that the Btn0State and Btn1State bits are set to 1 if the corresponding mouse button is *up*, whereas the bits for the five modifier keys are set to 1 if their corresponding keys are *down*.

EVENT MASKS

Some of the TBEM and OSEM routines can be restricted to operate on a specific event type or group of types; in other words, the specified event types are enabled while all others are disabled. For instance, instead of just requesting the next available event, the application can specifically ask for the next keyboard event.

An application can specify which event types a particular call applies to by supplying an **event mask** as a parameter. This is an integer in which there's one bit position for each event type, as shown below. The bit position representing a given type corresponds to the event code for that type—for example, update events (event code 6) are specified by bit 6 of the mask. A 1 in bit 6 means that this call applies to update events; a 0 means that it doesn't.



Note: Null events can't be disabled; a null event will always be reported when none of the enabled types of events are available.

There's also a global **system event mask** that controls which event types get posted into the event queue by the OSEM. Only event types corresponding to bits set in the system event mask are posted; all others are ignored. When the system starts up, the system event mask is set to post all events.

USING THE EVENT MANAGER

If an application will be using the Event Manager and the Window Manager, it must initialize the Event Manager **before** initializing the Window Manager. The Event Manager is initialized by calling the TBEM routine EMStartUp. Since the TBEM needs to share data with the Window Manager, they must both use the same zero page work area. When the Window Manager is initialized, it must call the TBEM routine DoWindows to obtain the address of the zero page work area which has been assigned to the Event Manager. If DoWindows is not called, the TBEM will assume that windows are not being used and will not attempt to return window events.

Event-driven applications have a main loop that repeatedly calls GetNextEvent to retrieve the next available event, and then takes whatever action is appropriate for each type of event. Some typical responses to commonly occurring events are described below. The program is expected to respond only to those events that are directly related to its own operations. After calling GetNextEvent, it should test the Boolean result to find out whether it needs to respond to the event: TRUE means the event may be of interest to the application; FALSE usually means it will not be of interest.

In some cases, the application may simply want to look at a pending event while leaving it available for subsequent retrieval by GetNextEvent. It can do this with the TBEM routine EventAvail.

Responding to Mouse Events

On receiving a mouse-down event, an application should first call the Window Manager to find out where on the screen the mouse button was pressed, and then respond in whatever way is appropriate. Depending on the part of the screen in which the button was pressed, this may involve calls to Toolbox routines in the Menu Manager, the Desk Manager, the Window Manager, or the Control Manager.

If the application attaches some special significance to pressing a modifier key along with the mouse button, it can discover the state of that modifier key when the mouse button was down by examining the appropriate flag in the modifiers field of the event record.

If the application wishes to respond to mouse double-clicks, it will have to detect them itself. It can do so by comparing the time and location of a mouse-up event with those of the immediately following mouse-down event. It should assume a double-click has occurred if both of the following are true:

- The times of the mouse-up event and the mouse-down event differ by a number of ticks less than or equal to the value returned by the TBEM function GetDblTime.

- The locations of the two mouse-down events separated by the mouse-up event are sufficiently close to each other. Exactly what this means depends on the particular application. For instance, in a word-processing application, two locations might be considered essentially the same if they fall on the same character, whereas in a graphics application they might be considered essentially the same if the sum of the horizontal and vertical changes in position is no more than five pixels.

Mouse-up events may be significant in other ways; for example, they might signal the end of dragging to select more than one object. Most simple applications, however, will ignore mouse-up events.

Responding to Keyboard Events

For a key-down event, the application should first check the modifiers field to see whether the character was typed with the Open-Apple key held down; if so, the user may have been choosing a menu item by typing its keyboard equivalent.

If the key-down event was not a menu command, the application should then respond to the event in whatever way is appropriate. For example, if one of the windows is active, it might want to insert the typed character into the active document; if none of the windows is active, it might want to ignore the event.

Usually the application can handle auto-key events the same as key-down events. It may, however, want to ignore auto-key events that invoke commands that shouldn't be continually repeated.

Responding to Window Events

When the application receives an activate event for one of its own windows, the Window Manager will already have done all of the normal "housekeeping" associated with the event, such as highlighting or unhighlighting the window. The application can then take any further action that it may require, such as showing or hiding a scroll bar or highlighting or unhighlighting a selection.

On receiving an update event for one of its own windows, the application should usually update the contents of the window.

Responding to Other Events

An application will never receive a desk accessory event since these are intercepted and handled by the Desk Manager.

If the application receives a switch event, it should call a (currently unnamed) routine in the Switcher which will save the current state and switch to the next application.

Posting and Removing Events

If an application is using application-defined events, it will need to call the OSEM function PostEvent to post them into the event queue. Device drivers can post events the same way. This function is sometimes also useful for reposting events that have been removed from the event queue with GetNextEvent.

In some situations an application may want to remove from the event queue some or all events of a certain type or types. It can do this with the OSEM procedure FlushEvents.

Other Operations

In addition to receiving the user's mouse and keyboard actions in the form of events, applications can directly read the mouse location and state of the mouse buttons by calling the TBEM routines GetMouse and Button, respectively. To follow the mouse when the user moves it with the button down, the application can use the TBEM routines StillDown or WaitMouseUp.

The TBEM function TickCount returns the number of ticks since the last system startup. This value can be compared to the when field of an event record to discover the delay since that event was posted.

The TBEM function GetCaretTime returns the number of ticks between blinks of the "caret" (usually a vertical bar) marking the insertion point in editable text. An application should call GetCaretTime if it is causing the caret to blink itself. The application would check this value each time through the main event loop to ensure a constant frequency of blinking.

Applications should never call the TBEM routines DoWindows and SetSwitch, and will probably never call the OSEM routines GetOSEvent, OSEventAvail and SetEventMask.

TOOLBOX EVENT MANAGER ROUTINES

HouseKeeping

```
EMBootInit          Call # 1
```

EMBootInit is called at boot time. It does nothing.

```
EMStartUp               Call # 2

     input              ZeroPageAdrs     INTEGER
     input              QueueSize        INTEGER
     input              XMinClamp        INTEGER
     input              XMaxClamp        INTEGER
     input              YMinClamp        INTEGER
     input              YMaxClamp        INTEGER
     input              ProgramID        INTEGER
```

EMStartUp initializes the Event Manager. ZeroPageAdrs is the starting address in Bank 0 of a 1-page work area assigned to the Event Manager. QueueSize specifies the maximum number of event records the queue can hold. If QueueSize is 0, a default size of 20 will be used. If QueueSize is greater than 3639, an error will be returned and the Event Manager will not be initialized. The Clamp inputs specify the minimum and maximum X and Y clamps for the mouse. Before the Event Manager passes the clamp values to the mouse, it will decrement XMaxClamp and YMaxClamp by one. ProgramID is the ID the Event Manager will use when getting memory from the Memory Manager. If the event queue cannot be allocated due to insufficient memory, an error will be returned and the Event Manager will not be initialized. Duplicate EMStartUp calls will cause an error to be returned.

```
EMShutDown              Call # 3
```

EMShutDown shuts down the Event Manager and releases any workspace allocated to it.

```
EMVersion               Call # 4

     output             VersionInfo      INTEGER
```

EMVersion returns identifying information for the Event Manager.

```
EMReset                 Call # 5
```

EMReset returns an error if the Event Manager is active, otherwise it does nothing.

```
EMActive                Call # 6

     output             ActiveFlag       INTEGER
```

EMActive returns a non-zero value if the Event Manager is active, otherwise it returns a 0.

```
DoWindows               Call # 9

     output             ZeroPageAdrs     INTEGER
```

DoWindows is called by the Window Manager when it is initialized. It returns the address of the zero page work area used by the Event Manager. The Window Manager will use the high end of this area while the Event Manager will use the low end. This routine should not be called by an application.

Accessing Events

```
GetNextEvent            Call # 10

     input              EventMask        INTEGER
     input              EventPtr         POINTER to an EventRecord
     output             Result           BOOLEAN
```

GetNextEvent returns the next available event of a specified type or types and, if the event is in the event queue, removes it from the queue. The event is returned in the event record pointed to by EventPtr. EventMask specifies which event types are of interest. GetNextEvent returns the next available event of any type designated by the mask, subject to the priority rules discussed above under "Priority of Events". If no event of any of the designated types is available, GetNextEvent returns a null event.

Events in the queue that aren't designated in the mask are left in the queue. They can be removed by calling the OSEM procedure FlushEvents.

Before reporting an event to the application, GetNextEvent first calls the Desk Manager function SystemEvent to see whether the system wants to intercept and respond to the event. If so, or if the event being reported is a null event, GetNextEvent returns a function result of FALSE; a function result of TRUE means that the application should handle the event itself. The Desk Manager intercepts the following events:

- desk accessory events
- activate and update events directed to a desk accessory
- mouse-up and keyboard events, if the currently active window belongs to a desk accessory

In each case, the event is intercepted by the Desk Manager only if the desk accessory can handle that type of event; however, as a rule all desk accessories should be set up to handle activate, update, and keyboard events and should not handle mouse-up events.

```
EventAvail              Call # 11

        input           EventMask         INTEGER
        input           EventPtr          POINTER to an EventRecord
        output          Result            BOOLEAN
```

EventAvail works exactly the same as GetNextEvent except that if the event is in the event queue, it's left there.

> **Note:** An event returned by EventAvail will not be accessible later if in the meantime the queue becomes full and the event is discarded from it; since the events discarded are always the oldest ones in the queue.

Reading the Mouse

```
GetMouse                Call # 12

        input           MouseLocPtr       POINTER to a Point
```

GetMouse returns the current mouse location in the record pointed to by MouseLocPtr. The location is given in the local coordinate system of the current grafPort (which might be, for example, the currently active window). Notice that this differs from the mouse location stored in the where field of an event record; that location is always in global coordinates.

```
Button                  Call # 13

        input           ButtonNum         INTEGER
        output          Result            BOOLEAN
```

The Button function returns TRUE if the mouse button is currently down, and FALSE if it isn't. ButtonNum contains the number (0 or 1) of the button to check. An error is returned if ButtonNum is not 0 or 1.

```
StillDown               Call # 14

        input           ButtonNum         INTEGER
        output          Result            BOOLEAN
```

Usually called after a mouse-down event, StillDown tests whether the mouse button is still down. ButtonNum contains the number (0 or 1) of the button to check. StillDown returns TRUE if the button is currently down and there are no more mouse events (for the specified button) pending in the event queue. This is a true test of whether the button is still down from the original press—unlike Button (above), which returns TRUE whenever the button is currently down, even if it has been released and pressed again since the original mouse-down event. An error is returned if ButtonNum is not 0 or 1.

```
WaitMouseUp              Call # 15

     input               ButtonNum        INTEGER
     output              Result           BOOLEAN
```

WaitMouseUp works exactly the same as StillDown (above), except that if the button is not still down from the original press, WaitMouseUp removes the preceding mouse-up event before returning FALSE. If, for instance, the application attaches some special significance both to mouse double-clicks and to mouse-up events, this function would allow the application to recognize a double-click without being confused by the intervening mouse-up. An error is returned if ButtonNum is not 0 or 1.

Miscellaneous Routines

```
TickCount                Call #16

     output              Count            LONGINT
```

TickCount returns the current number of ticks (sixtieths of a second) since the system last started up. Applications should not rely on the tick count being exact. The tick count is incremented during the VBL interrupt, but it's possible for this interrupt to be disabled. Furthermore, applications should not rely on the tick count being incremented to a certain value, such as testing whether it has become equal to its old value plus 1. They should check instead for "greater than or equal to" (since an interrupt task may keep control for more than one tick.)

```
GetDblTime               Call # 17

     output              MaxTicks         LONGINT
```

GetDblTime returns the suggested maximum difference (in ticks) that should exist between the times of a mouse-up event and a mouse-down event for those two mouse clicks to be considered a double-click. The user can adjust this value by means of the Control Panel.

```
GetCaretTime             Call # 18

     output              NumTicks         LONGINT
```

GetCaretTime returns the time (in ticks) between blinks of the "caret" (usually a vertical bar) marking the insertion point in editable text. If an application is not using TextEdit, it will need to cause the caret to blink itself. On every pass through the program's main event loop, it should check this value against the elapsed time since the last blink of the caret. The user can adjust this value by means of the Control Panel.

```
SetSwitch               Call # 19
```

SetSwitch is called by the Control Manager to inform the TBEM of a pending switch event. This routine should not be called by an application.

OPERATING SYSTEM EVENT MANAGER ROUTINES

Posting and Removing Events

```
PostEvent               Call # 20

        input           EventCode       INTEGER
        input           EventMsg        LONGINT
        output          Result          INTEGER
```

PostEvent places in the event queue an event of the type designated by EventCode, with the event message specified by EventMsg and with the current time, mouse location, and state of the modifier keys and mouse buttons. It returns a result code equal to one of the following values:

0 - event posted
1 - event type not designated in system event mask

An error is returned if EventCode is greater than 15.

When PostEvent is called to post a keyboard or mouse event, the current state of the modifier keys and mouse buttons **must** be supplied in the high-order word of the event message. The information is then moved into the modifiers word of the event record.

Applications should be very careful when posting any events other than their own application-defined events into the queue. Attempting to post an activate or update event, for example, will interfere with the internal operation of the TBEM, since such events aren't normally placed in the queue at all.

Note: If PostEvent is used to repost an event, the event time, mouse location, and state of the modifier keys and mouse buttons will all be changed from their values when the event was originally posted, possibly altering the meaning of the event.

```
FlushEvents             Call # 21

        input           EventMask       INTEGER
        input           StopMask        INTEGER
        output          Result          INTEGER
```

FlushEvents removes events from the event queue as specified by the given event masks. It removes all events of the type or types specified by EventMask, up to but not including the first event of any type specified by StopMask. If the event queue doesn't contain any events of the types specified by EventMask, it does nothing. To remove all events specified by EventMask, use a StopMask value of 0. On exit from this routine, Result contains 0 if all events were removed from the queue or, if not, an event code specifying the type of event that caused the removal process to stop.

Accessing Events

```
GetOSEvent              Call # 22

        input           EventMask       INTEGER
        input           EventPtr        POINTER to an EventRecord
        output          Result          BOOLEAN
```

GetOSEvent returns the next available event of a specified type or types and removes it from the event queue. The event is returned in the event record pointed to by EventPtr. EventMask specifies which event types are of interest. GetOSEvent will return the next available event of any type designated by the mask. If no event of any of the designated types is available, GetOSEvent returns a null event and a function result of FALSE; otherwise it returns TRUE.

Unlike the TBEM function GetNextEvent, GetOSEvent doesn't call the Desk Manager to see whether the system wants to intercept and respond to the event.

```
OSEventAvail            Call # 23

        input           EventMask       INTEGER
        input           EventPtr        POINTER to an EventRecord
        output          Result          BOOLEAN
```

OSEventAvail works exactly the same as GetOSEvent (above) except that it doesn't remove the event from the event queue.

Note: An event returned by OSEventAvail will not be accessible later if in the meantime the queue becomes full and the event is discarded from it; since the events discarded are always the oldest ones in the queue.

Miscellaneous Routines

`SetEventMask`	`Call # 24`	
`input`	`TheMask`	`INTEGER`

SetEventMask sets the system event mask to the specified event mask. The OSEM will post only those event types that correspond to bits set in the mask. (As usual, it will not post activate, update or switch events, which are not stored in the event queue.) The system event mask is initially set to post all events.

Note: Because desk accessories may rely on receiving certain types of events, an application should not change the system event mask.

THE JOURNALING MECHANISM

The Event Manager contains hooks to support journaling. Journaling "decouples" the Event Manager from the user and feeds it events from a file which contains a recording of all the events that occurred during some portion of a user's session. Specifically, this file is a recording of all calls to the TBEM routines GetNextEvent, EventAvail, GetMouse, Button and TickCount. When a journal is being recorded, every call to any of these routines is sent to a journaling device driver, which records the call (and the results of the call) in a file. When the journal is played back, these recorded TBEM calls are taken from the journal file and sent directly to the TBEM. The result is that the recorded sequence of user-generated events is reproduced when the journal is played back.

The journaling device driver does not currently exist but hooks are in the Event Manager so that one could be written in the future. The Event Manager calls the journaling device driver by jumping through the long address stored at $E100E9. This address is set to $000000 when EMStartUp is executed. A word at $E100E7 (JournalFlag) controls whether journaling is active, and if so, if it is in recording or play back mode. If JournalFlag is set to 0, journaling is not active. If JournalFlag is non-zero, journaling is active. A positive value indicates recording mode and a negative value indicates play back mode. JournalFlag is set to $00 when EMStartUp is executed.

If journaling is active, the TBEM routines GetNextEvent, EventAvail, GetMouse, Button and TickCount will push information on the stack and do a JSL to the journaling device driver whose address is at $E100E9. The journaling driver should remove the information from the stack before returning. The information pushed on the stack is as follows -

JournalFlag	INTEGER
JournalCode	INTEGER
ResultPtr	POINTER

JournalFlag is the current value stored at $E100E7. JournalCode is a code indicating which routine the journaling driver is being called from. ResultPtr is a pointer to the actual data being returned by the routine (for example, a pointer to an event record for GetNextEvent).

Routine	**ResultPtr points to**	**JournalCode**
TickCount	Long Word	0
GetMouse	Point	1
Button	Boolean	2
GetNextEvent	Event Record	4
EventAvail	Event Record	4

EVENT MANAGER ERROR CODES

$0601	Duplicate EMStartUp call
$0602	Reset error
$0603	Event Manager not active
$0604	Illegal event code
$0605	Illegal button number
$0606	Queue size too large
$0607	Not enough memory available for queue