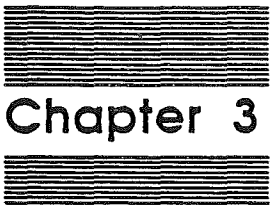


Chapter 3

Event Manager

Overview

The Event Manager allows applications to monitor the user's actions, such as those involving the mouse, keyboard, and keypad.

The routines available in the Event Manager are summarized in the following table:

Table X.X - Event Manager Routines and Their Functions

Event Manager Standard Housekeeping Routines

EMBootInit	Initializes Event Manager at boot time.
EMStartUp	Initializes the Event Manager when an application starts up.
EMShutDown	Shuts down the Event Manager and releases any workspace allocated to it.
EMVersion	Returns the version of the Event Manager.
EMReset	Returns an error if the Event Manager is active, but otherwise does nothing.
EMActive	Returns status indicating whether the Event Manager is active.
DoWindows	Returns address of the Event Manager 's zero page work area to the Window Manager.

Toolbox Event Manager Routines :These routines check events to see if they are of interest to the application. If the events are of interest, and the Desk Manager doesn't want them, the routines return with the event.

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.
EventAvail	Returns the next available event of a specified type or types, but if the event is in the event queue, leaves it in the queue.

Mouse Reading Routines: These routines provide the ability to read the status of the mouse.

GetMouse	Returns the current location of the mouse.
Button	Checks the status of a specified mouse button.
Stilldown	Checks a specified mouse button to see if it is still down.
WaitMouseUp	Checks a specified mouse button to see if it is still down, and, if not, removes preceding mouse-up event.

Posting and Removing Events

PostEvent	Places an event in the event queue.
FlushEvents	Removes all events of the type or types specified up to but not including the first event of any type specified by a mask.

Accessing Events Routines: These routines check events to see if they are of interest to the application. If the events are of interest, the routines return with the event.

GetOSEvent	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.
OSEventAvail	Returns the next available event of a specified type or types, but if the event is in the event queue, leaves it in the queue.

Miscellaneous Event Manager Routines:

TickCount	Returns a count of the number of ticks since the system last started up.
GetDblTime	Returns suggested maximum difference of ticks which determines a double mouse-click..
GetCaretTime	Returns the number of ticks between blinks of the caret marking the insertion point.
~~GetCaretTime~~	~~Returns the number of ticks between blinks of the caret marking the insertion point.~~
SetSwitch	Called by the Control Manager to inform Event Manager of a pending switch event. Should not be called by an application.
SetEventMask	Specifies the system event mask. Should not be called by an application.

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 tool set, it can be 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 because 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 discussed in the following sections.

Mouse events

Pressing the mouse button generates a mouse-down event; 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. These events are either Activate or Update events.

Activate events

These 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

These 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

Device driver events

These events 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.

Application-defined events

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.

Switch events

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

Desk accessory events

A desk accessory event is generated whenever the user enters the special keystoke to invoke a "classic" desk accessory .

Null events

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

Event Priority

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. Specifying such events 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 ensure 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 in the following sections.

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.

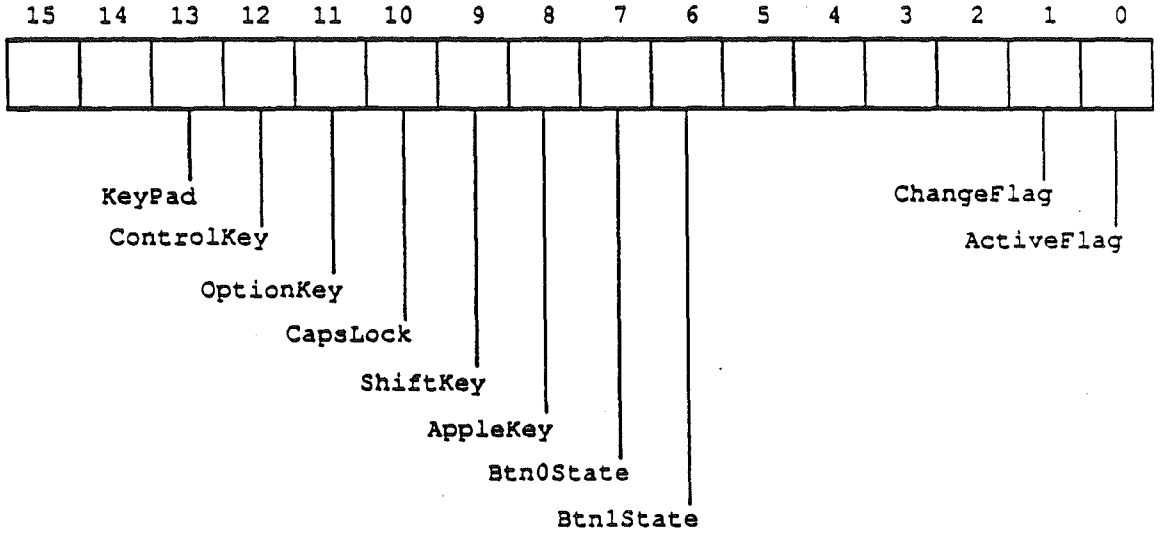
Table X-X: Event Messages

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 modifier flags are shown in the following figure.

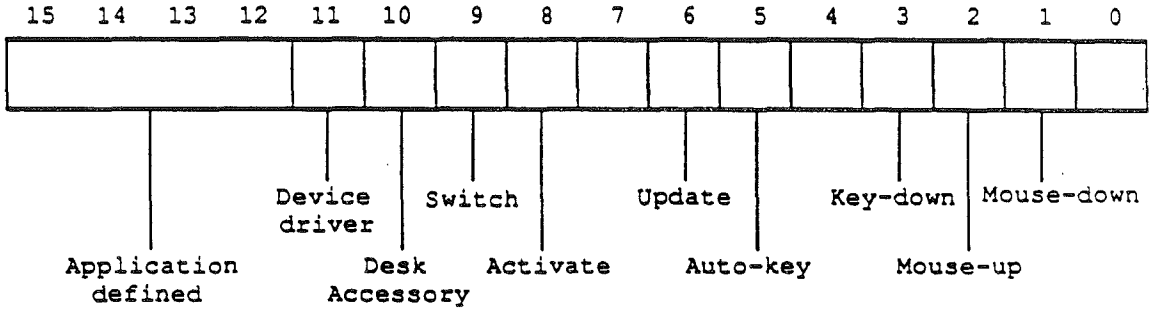


The ActiveFlag and ChangeFlag bits give further information about activate events. The ActiveFlag bit is set to 1 if the window pointed to by the event message is being activated, or 0 if the window is being deactivated. The ChangeFlag bit is set to 1 if the active window is changing from an application 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. The



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 Managers and the Window Manager, it must initialize the Event Managers before initializing the Window Manager. The TBEM and OSEM are initialized by calling the TBEM routine EMStartUp. Because 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 that has been

assigned to the Event Managers. 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 in the next section. 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 EventAvail call.

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, the application may have to call 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. The application 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 way as key-down events. You may, however, want it 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 because 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 that 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, you may want your application 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 routine TickCount returns the number of ticks since the last system start-up. 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.

USING ALTERNATIVE POINTING DEVICES

The Event Manager can use an alternative pointing device, such as a graphics tablet, instead of the mouse. When an alternative pointing device is being used, its XY

location and button status will appear in the event records instead of the mouse information. Mouse-up and Mouse-down events will be posted when the alternative device's buttons change state. An application which uses the Event Manager will not know that an alternative pointing device is being used; it will work the same as it does with the mouse.

More than one pointing device can also be used. In this case, whichever device is currently moving or changing state will be the device whose XY location appears in the event records. The cursor will also correspond to the device which is currently moving or changing state.

Installing Device Drivers

In order to use an alternative pointing device, a device driver must be written for it and installed in the system. The user should install the device driver by executing a startup program. If the startup program is a desk accessory, the user can install the driver while using an application. The startup program should initialize the device and install the device driver into the system as detailed in the following paragraphs.

Devices Using Their Own Cards

If the device communicates using its own card, install the device driver by taking the following steps:

1. Determine which slot the device's card is in. Store the slot number in the appropriate byte of the device driver header (described below).
2. Next, perform any initialization needed by the device such as setting up scaling and offset values, setting the correct operation mode, etc.
3. Install the driver into either the Heartbeat Queue or the IRQ_Other interrupt vector depending on whether or not the device generates interrupts.

   If the device does not generate interrupts, the driver should be installed as a task in the Heartbeat queue. Install the driver using the SetHeartBeat routine in the Miscellaneous Tool Set.

   If the device does generate interrupts, the driver should be installed in the IRQ_Other interrupt vector after first saving the previous contents of the vector. The contents of the vector are obtained by calling the GetVector routine, in the Miscellaneous Tool Set, with a reference number of $17. The driver is then installed by calling the SetVector routine, in the Miscellaneous Tool Set, with a reference number of $17.

Devices Communicating Through the Serial Port

If the device communicates through the Serial port, install the device driver by taking the following steps:

1. Determine which port the device is connected to. Store the port number in the appropriate byte of the device driver header (described below).
2. Initialize the device by calling the Serial Init routine.
3. Install the driver in the Serial firmware's completion vector. This is done by issuing a SetIntInfo call to the Serial firmware. The command list for the call should specify that 'character available' interrupts should be passed to the driver.
3. Turn on buffering by calling the Serial Write routine with the following three characters - control I, B, E.

Devices Communicating Through the Apple Desktop Bus

If the device communicates though Apple Desktop Bus, install the device driver by taking the following steps:

1. Determine the address number assigned to the device. Store the address number in the appropriate byte of the device driver header (described below).
2. Install the driver in the ADB firmware's SRQ List completion vector. This is done by calling the SRQPL routine in the ADB Tool Set.
3. Enable SRQ for the device using the Send routine in the ADB Tool Set.

Removing Device Drivers

The user should remove the device driver by executing a shutdown program. If the shutdown program is a desk accessory, the user can remove the driver while using an application. The shutdown program should shut down the device and remove the device driver from the system as follows -

Devices Using Their Own Cards

If the device communicates using its own card, remove the device driver by taking the following steps:

1. Shut down the device if possible.
2. If the driver is installed in the Heartbeat queue, remove it by calling the DelHeartBeat routine in the Miscellaneous Tool Set. If the driver is installed in the IRQ_Other interrupt vector, restore the previous contents of the vector.
3. Remove the driver from memory.

Devices Communicating Through the Serial Port

If the device communicates through the Serial port, remove the device driver by taking the following steps:

1. Turn off buffering by calling the Serial Write routine with the following three characters - Control I, B, D.
2. Remove the driver from memory.

Devices Communicating Through the Apple Desktop Bus

If the device communicates through Apple Desktop Bus, remove the device driver by taking the following steps:

1. Disable SRQ for the device.
2. Remove the driver from the ADB firmware's SRQ List completion vector. This is done by calling the SRQRMV routine in the ADB Tool Set.
3. Remove the driver from memory.

Device drivers will be called with the processor in native 8-bit mode and must exit in native 8-bit mode. If a device driver will be installed as a Heartbeat task, it must be written in accordance with the instructions in the Miscellaneous Tool Set, under the Heartbeat Interrupt Tools. All other device drivers must be written according to interrupt routine guidelines.

All device drivers should begin with a 6 byte header as follows -

BRA CodeStart (this generates 2 bytes of code)
2 bytes of device information
2 bytes initialized to $8989 which is the device driver signature

If the device driver is installed as a Heartbeat task, the driver header should be immediately after the Heartbeat task header.

The low byte of device information should be set up as follows -

Bit 0 - Set if the device has its own card and does not generate interrupts
Bit 1 - Set if the device has its own card and does generate interrupts
Bit 2 - Set if the device communicates through the Serial port
Bit 3 - Set if the device communicates through Apple Desktop Bus
Bit 4 - Reserved for future use
Bit 5 - Reserved for future use
Bit 6 - Set if the device is a relative device
Bit 7 - Set if the device is an absolute device

The high byte of device information should be initialized to $FF. The startup program should then set up this byte differently depending on the type of device being installed -

Card device - byte will contain the slot # where the card was found
Serial device - byte will contain the port # the device is connected to
ADB device - byte will contain the address # assigned to the device

A device driver should perform the following steps:

1. Call the GetAddr routine in the Miscellaneous Tool Set to obtain the address of the relative or absolute clamp values (depending on whether the driver is for a relative or absolute device). Save the address so that this call only has to be made the first time the device driver is executed.
2. If the driver is installed as a Heartbeat task, reset the heartbeat task counter to 1 or 2. Poll the device to obtain its current XY position and button state.

   If the driver is for a serial device, issue an InQStatus call to determine how many characters are in the serial firmware's input queue. Read the characters by calling the Serial Read routine.

   If the driver is for an ADB device, there will be a buffer pointer on the stack at offset 7. The first byte in the buffer specifies the number of data bytes in the buffer. Read the data bytes.
3. Determine if the device's XY position or button state has changed. If no changes, exit.
4. Push a word on the stack which is set up as follows -
   Bit 1 - set if XY position has changed, else clear
   Bit 2 - set if button state has changed, else clear
5. Read the keyboard modifiers latch at $C025 (must be done in 8-bit mode) and push the byte on the stack. Push a byte of 0 on the stack.
6. Determine the device's absolute X position. Get the current X clamps, using the address saved above, and clamp the X position. Push a word containing the clamped, absolute X position on the stack.
7. Determine the device's absolute Y position. Get the current Y clamps, using the address saved above, and clamp the Y position. Push a word containing the clamped, absolute Y position on the stack.
8. Push a word on the stack which is set up as follows -
   Bit 8 - Previous state of button 1 (0 if up, 1 if down)
   Bit 12 - Current state of button 1
   Bit 14 - Previous state of button 0
   Bit 15 - Current state of button 0
9. Call the FakeMouse routine in the Event Manager (must be called in native 16-bit mode).
10. Go back to native 8-bit mode.
11. RTL

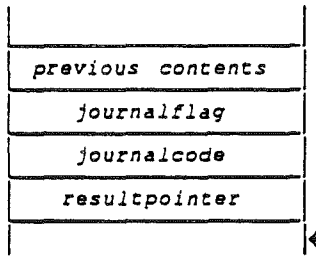
The Journaling Mechanism

The Event Manager has a journaling mechanism that can be accessed through assembly language. The journaling mechanism "decouples" the Event Manager from the user and feeds it events from a file that 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 exist , but hooks are present in the Event Manager which allow one to be written. In order to use journaling, the address of the journaling driver must be placed in the EM variable *journalptr*. The Event Manager calls the journaling device driver by jumping through *journalptr*. *Journalptr* is set to $00000000 when EMStartUp is executed.

The information pushed on the stack is as follows:



Word indicating current value stored at $E100E7.

Word indicating Code for the routine calling the journaling driver.

Pointer to the actual data being returned by the routine.

The locations of *journalptr* and *journalflag* should be obtained by calling the Miscellaneous Tools routine GetAddr. The *journalflag* controls whether journaling is active, and, if so, whether it is in recording or playback 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 playback 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 values for the *journalcode* and *resultpointer* are summarized in the following table:

Table X-X: Journal Codes and Result Pointers

Journal Code	Routine	Result Pointer points to:
0	TickCount	LONG
1	GetMouse	Point
2	Button	BOOLEAN
4	GetNextEvent	Event Record
4	EventAvail	Event Record

EMBootInit

Called at boot time. Does nothing. Should not be called by an application.

Stack & Parameter Definition

Stack Before and After Call

Call does not affect the stack.

C

Call should not be made from an application.

Pascal

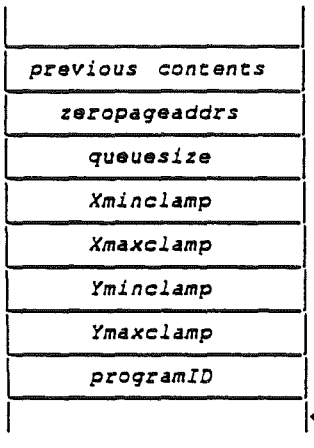
Call should not be made from an application.

EMStartUp

Initializes the Event Manager, sets size of event queue, specifies minimum and maximum mouse clamp values, and defines ID program will use to get memory from the Memory Manager.

Stack & Parameter Definition

Stack Before Call



Integer specifying starting address in bank 0 for EM's one-page work area.

Integer specifying maximum number of event records the queue can hold.

Integer specifying minimum X clamp value for the mouse.

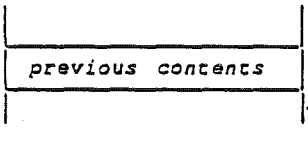
Integer specifying maximum X clamp value for the mouse.

Integer specifying minimum Y clamp value for the mouse.

Integer specifying maximum Y clamp value for the mouse.

Integer specifying IDto use to get memory from Memory Manager.

Stack After Call



If *queuesize* is equal to zero, 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.

Before the Event Manager passes the clamp values to the mouse, it decrements XMaxClamp and YMaxClamp by one.

If the event queue cannot be allocated due to insufficient memory, an error is returned and the Event Manager is not initialized. Duplicate EMStartUp calls also cause an error to be returned.

C

Pascal

EMShutDown

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

Stack & Parameter Definition

Stack Before and After Call

Call does not affect the stack.

C

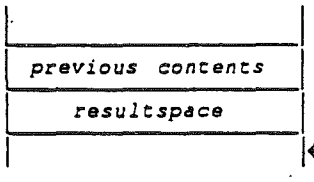
Pascal

EMVersion

Returns the version of the Event Manager.

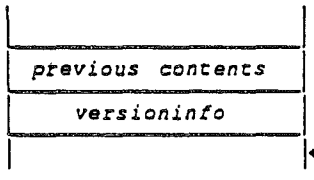
Stack & Parameter Definition

Stack Before Call



Word allowing space for the output.

Stack After Call



Word indicating which version of the Event Manager is present.

C

Pascal

EMReset

Returns an error if the Event Manager is active; otherwise does nothing.

Stack & Parameter Definition

Stack Before and After Call

Call does not affect the stack.

C

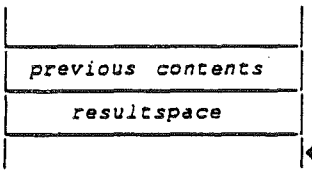
Pascal

EMActive

Returns a non-zero value if the Event Manager is active; returns zero if the Event Manager is inactive.

Stack & Parameter Definition

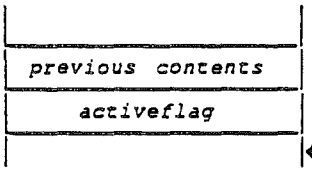
Stack Before Call



Word allowing space for the output.

← **SP**

Stack After Call



Integer indicating zero if Event Manager not active, non-zero if active.

← **SP**

C

Pascal

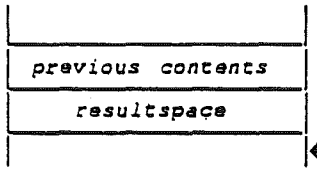
DoWindows

Returns the address of the zero-page work area used by the Event Manager to the Window Manager.

DoWindows is called by the Window Manager when the Window Manager is initialized. The Window Manager uses the high end of the ZeroPageAdrs returned by DoWindows; the Event Manager uses the low end. This routine should not be used by an application.

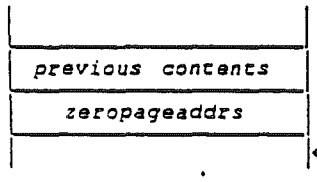
Stack & Parameter Definition

Stack Before Call



Word allowing space for the output.

Stack After Call



Integer returning the zero page address of the EM's work area.

C

Call should not be made from an application.

Pascal

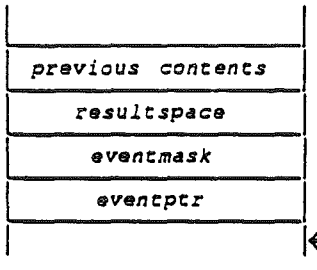
Call should not be made from an application.

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.

Stack & Parameter Definition

Stack Before Call

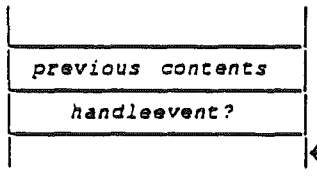


Word allowing space for the output.

Integer specifying which types of events are to be retrieved.

Pointer to event record in which the event will be placed..

Stack After Call



Boolean returning TRUE if application should handle event;
FALSE if system or null event.

C

Pascal

GetNextEvent returns the next available event of any type designated by the mask, subject to the following priority order:

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).

If no event of any of the designated types is available, GetNextEvent returns a null event. This priority order is further discussed in "Event Priority ".

Events in the queue that aren't designated in the mask are left in the queue. The events can be removed by calling the FlushEvents tool.

Before reporting an event to the application, GetNextEvent first calls the Desk Manager tool 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 Boolean result of FALSE; a Boolean 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. As a rule, all desk accessories should be set up to handle activate, update, and keyboard events and should not handle mouse-up events.

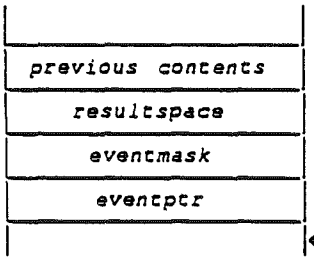
EventAvail

Returns the next available event of a specified type or types. If the event is in the event queue, leaves the event in the queue. Otherwise, the call works exactly like GetNextEvent.

An event returned by EventAvail cannot be accessed if, in the meantime, the queue becomes full and the event is discarded. However, because the oldest events are the ones discarded, useful events will be discarded only in an unusually busy environment.

Stack & Parameter Definition

Stack Before Call

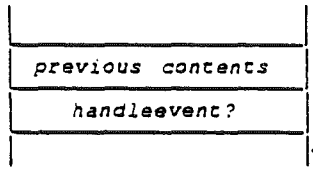


Word allowing space for the output.

Integer specifying which types of events are to be retrieved.

Pointer to event record in which the event will be placed..

Stack After Call



Boolean returning TRUE if application should handle event; FALSE if system or null event.

C

Pascal

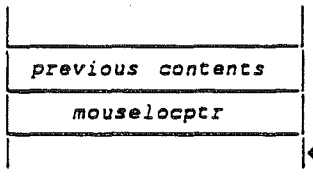
GetMouse

Returns the current mouse location.

The location is given in the local coordinate system of the current GrafPort (for example, the currently active window). This differs from the mouse location stored in the *where* field of an event record; that location is always in global coordinates.

Stack & Parameter Definition

Stack Before Call



Pointer to a record in which the current mouse location will be returned..

Stack After Call

C

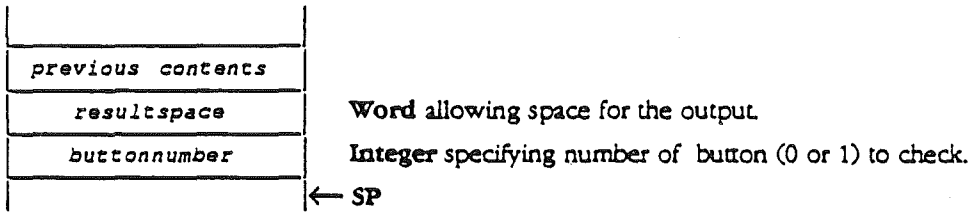
Pascal

Button

Returns the current state of a specified mouse button.

Stack & Parameter Definition

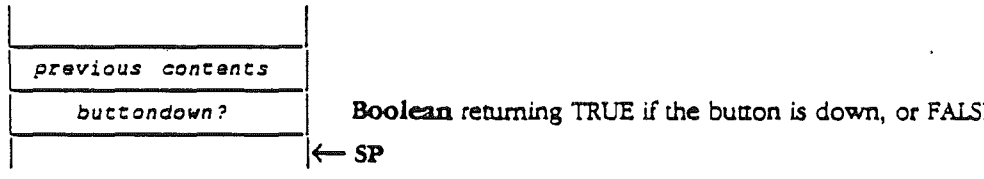
Stack Before Call



Word allowing space for the output.

Integer specifying number of button (0 or 1) to check.

Stack After Call



Boolean returning TRUE if the button is down, or FALSE if it isn't.

An error is returned if *buttonnumber* is not 0 or 1.

C

Pascal

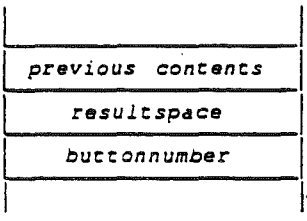
StillDown

Tests whether a specified mouse button is still down.

Usually called after a mouse-down event, StillDown is a true test of whether the mouse button is still down from the original press. (Button is not a true test, because it returns TRUE whenever the mouse button is currently down, even if the button was released and pressed again since the original mouse-down event.)

Stack & Parameter Definition

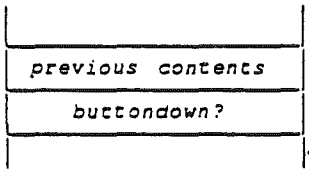
Stack Before Call



Word allowing space for the output.

Integer specifying number of button (0 or 1) to check.

Stack After Call



Boolean returning TRUE if the button is down,and there are no mouse events pending in the event queue.

An error is returned if *buttonnumber* is not 0 or 1.

C

Pascal

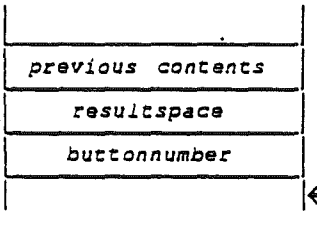
WaitMouseUP

Tests whether the mouse button is still down, and, if the button is not still down from the original press, removes the preceding mouse-up event before returning FALSE. An error is returned if ButtonNum is not 0 or 1.

WaitMouseUp could be used, for example, if an application attached some special significance to mouse double-clicks and to mouse-up events. WaitMouseUp would allow the application to recognize a double-click without being confused by the intervening mouse-up.

Stack & Parameter Definition

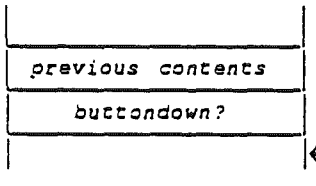
Stack Before Call



Word allowing space for the output.

Integer specifying number of button (0 or 1) to check.

Stack After Call



Boolean returning TRUE if the button is down,and there are no more mouse events pending in the event queue.

An error is returned if *buttonnumber* is not 0 or 1.

C

Pascal

PostEvent

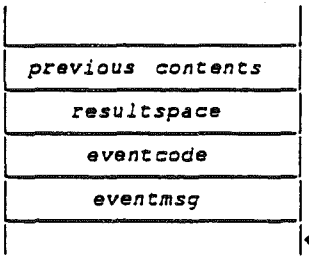
Places an event in the event queue.

An application must be careful when it posts any events other than its own application-defined events into the queue. Attempting to post an activate or update event (which aren't normally placed in the queue), for example, will interfere with the normal operation of the Event Manager.

If PostEvent is used to repost an event, the event time, mouse location, state of the modifier keys, and state of the mouse buttons will all be changed from the originally posted event. This can alter the meaning of the event.

Stack & Parameter Definition

Stack Before Call

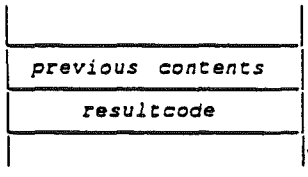


Word allowing space for the output.

Integer specifying type of event to be placed in the queue.

Longint specifying event message.

Stack After Call



Integer returning 0 if event posted, 1 if event type not designated in system event mask.

In the *eventmsg*, the current state of the modifier keys and mouse buttons are supplied in the high-order word of the message. The current time and mouse location are also recorded in the message.

C

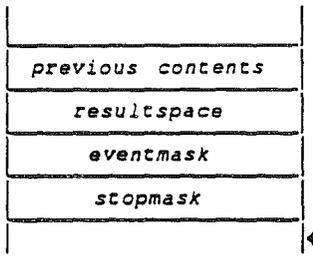
Pascal

FlushEvents

Removes all events of the type or types specified up to but not including the first event of any type specified by *stopmask*. If the event queue doesn't contain any event of the types specified by *eventmask*, FlushEvents does nothing.

Stack & Parameter Definition

Stack Before Call

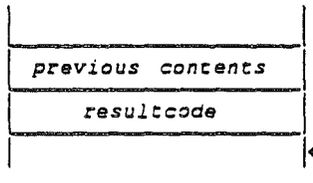


Word allowing space for the output.

Integer specifying type or types of events to be removed from the queue.

Integer specifying the first event type to not be removed (0 to remove. all events).

Stack After Call



Integer returning 0 if all events removed, or the event code indicating what type of event caused the process to stop.

C

Pascal

GetOSEvent

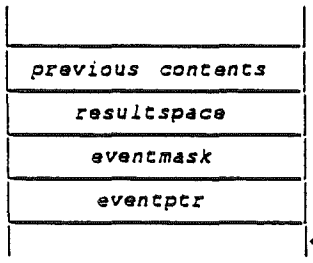
Returns the next available event of a specified type or types and, if the event is in the event queue, removes it from the queue.

GetOSEvent returns 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.

Events in the queue that aren't designated in the mask are left in the queue. The events can be removed by calling the FlushEvents tool.

Stack & Parameter Definition

Stack Before Call

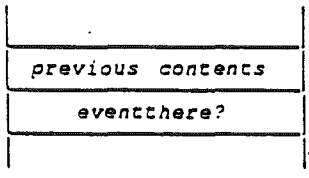


Word allowing space for the output.

Integer specifying which types of events are to be retrieved.

Pointer to event record in which the event will be placed..

Stack After Call



Boolean returning TRUE if any of the events specified are available; FALSE if null event.

C

Pascal

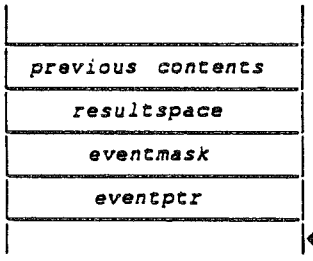
OSEventAvail

Returns the next available event of a specified type or types. If the event is in the event queue, leaves the event in the queue. Otherwise, the call works exactly like GetOSEvent.

An event returned by OSEventAvail cannot be accessed if, in the meantime, the queue becomes full and the event is discarded. However, because the oldest events are the ones discarded, useful events will be discarded only in an unusually busy environment.

Stack & Parameter Definition

Stack Before Call

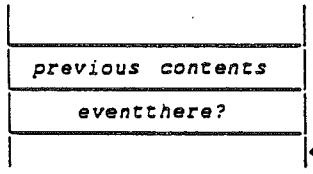


Word allowing space for the output.

Integer specifying which types of events are to be retrieved.

Pointer to event record in which the event will be placed..

Stack After Call



Boolean returning TRUE if any of the events specified are available;
FALSE if null event.

C

Pascal

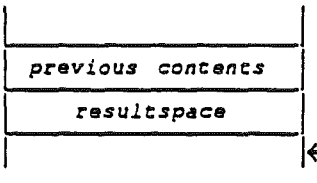
TickCount

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 that interrupt can be disabled. Also, since an interrupt task can keep control for more than one tick, an application should also not rely on the tick count being incremented to a certain value (for example, testing whether the tick count has become equal to its old value plus 1). Instead, the application should check for a "greater than or equal to" condition.

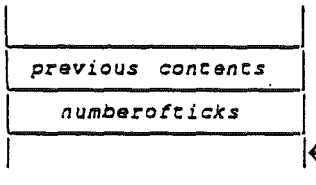
Stack & Parameter Definition

Stack Before Call



Long allowing space for the output.

Stack After Call



Longint returning the number of ticks since system start-up.

C

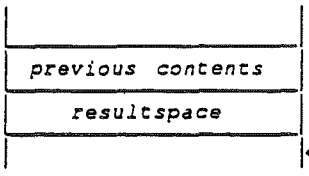
Pascal

GetDblTime

Returns the suggested maximum difference (in ticks) between mouse-up and mouse-down events in order for the mouse clicks to be considered a double click.The user can adjust this value by using the Control Panel.

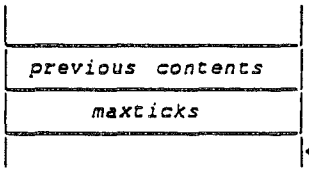
Stack & Parameter Definition

Stack Before Call



Long allowing space for the output.

Stack After Call



Longint returning the maximum number of ticks between mouse clicks.

C

Pascal

GetCaretTime

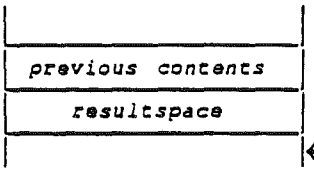
Returns the time (in ticks) between blinks of the "caret" (usually a vertical bar) marking the insertion point in text.

If an application is not using TextEdit, the application must cause the caret to blink. On every pass through the program's main event loop, the application should check *numticks* against the elapsed time since the last blink of the caret .

The user can adjust this value by using the Control Panel.

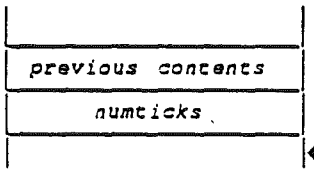
Stack & Parameter Definition

Stack Before Call



Long allowing space for the output.

Stack After Call



Longint returning the number of ticks between blinks of the caret.

C

Pascal

SetSwitch

Informs the Event Manager of a pending switch event. SetSwitch is called by the Control Manager and should not be called by an application.

Stack & Parameter Definition

Stack Before and After Call

Call does not affect the stack.

C

Call should not be made from an application.

Pascal

Call should not be made from an application.

SetEventMask

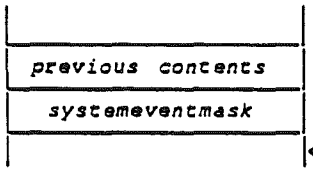
Sets the system event mask to the specified event mask.

The Event Manager will post only those event types that correspond to bits set in the mask. It will not post activate, update, or switch events, because those events are not stored in the event queue.

The system event mask is initially set to post all events. An application should not change the system event mask, because desk accessories may depend upon receiving certain types of events.

Stack & Parameter Definition

Stack Before Call



Integer specifying the system event mask.

Stack After Call

C

Pascal

Second digit ↓	First digit 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	space	0	@	P	`	p	Ä	ê	†	∞	¿	–		
1	SOH	DC1	!	1	A	Q	a	q	Å	ë	°	±	¡	—		
2	STX	DC2	"	2	B	R	b	r	Ç	í	¢	≤	¬	“		
3	ETX	DC3	#	3	C	S	c	s	É	ì	£	≥	√	”		
4	EOT	DC4	$	4	D	T	d	t	Ñ	î	§	¥	ƒ	‘		
5	ENQ	NAK	%	5	E	U	e	u	Ö	ï	•	µ	≈	’		
6	ACK	SYN	&	6	F	V	f	v	Ü	ñ	¶	∂	Δ	÷		
7	BEL	ETB	'	7	G	W	g	w	á	ó	ß	Σ	«	◇		
8	BS	CAN	(	8	H	X	h	x	à	ò	®	Π	»	Ÿ		
9	HT	EM	)	9	I	Y	i	y	â	ô	©	π	…			
A	LF	SUB	*	:	J	Z	j	z	ä	ö	™	∫	␣			
B	VT	ESC	+	;	K	[	k	{	ã	õ	´	ª	À			
C	FF	FS	,	<	L	\	l	|	å	ú	¨	º	Ã		
D	CR	GS	–	=	M	]	m	}	ç	ù	≠	Ω	Õ			
E	SO	RS	.	>	N	^	n	~	é	û	Æ	æ	Œ			
F	SI	US	/	?	O	_	o	DEL	è	ü	Ø	ø	œ			

␣ stands for a nonbreaking space, the same width as a digit.

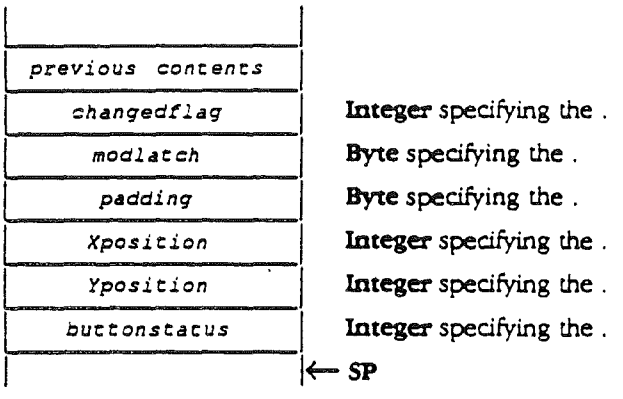
The shaded characters cannot normally be generated from the Macintosh keyboard or keypad.

FakeMouse

FakeMouse allows an alternative pointing device, such as a graphics tablet, to be used in place of or in conjunction with the mouse.

Stack & Parameter Definition

Stack Before Call



Integer specifying the .
Byte specifying the .
Byte specifying the .
Integer specifying the .
Integer specifying the .
Integer specifying the .

Stack After Call

C

Pascal

Event Manager Error Codes

The error codes returned by the Event Manager are summarized in the following table:

Table X-X: Event Manager Error Codes

Error Code	Means
$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.
$0681	Fatal system error - event queue damaged.
$0682	Fatal system error - queue handle damaged.