3.3 Release Notes: Event Status Driver

This file contains release notes for the 3.2, 3.1, and 3.0 releases of the Event Status Driver. Items specific to the 3.1 release are listed first, and the Release 3.0 notes follow. There are no items specific to the 3.3 or 3.2 releases.

Notes Specific to Release 3.1

New Features

The following new features have been added to the Event Status Driver since Release 3.0.

- The **NXEventSystemInfo()** call can return some new interface types:

```
NX_EVS_DEVICE_INTERFACE_ACE        // For x86 PC keyboards
NX_EVS_DEVICE_INTERFACE_SERIAL_ACE // For PC serial mice
```

```
NX_EVS_DEVICE_INTERFACE_BUS_ACE    // For PC bus mice
```

Bugs Fixed in Release 3.1

These bugs have been fixed in Release 3.1:

Reference	29522
Problem	**NXMouseButtonEnabled()** returns the opposite of what it should.
Description	When asked for the current enabled status of the buttons, the Event Status Driver returned NX_LeftButton when the right button was enabled for menus. Left was similarly reversed..

Known Problems

These new bugs have appeared only in NEXTSTEP for Intel Processors.

- Bringing up the NMI panel and then moving the mouse before entering `c' can result in the mouse responding incorrectly for a few seconds.
- The NMI key sequence doesn't work when the system is at a high interrupt level.
- Mouse speed varies widely between different makes of mice.

Notes Specific to Release 3.0

These notes were included with the Release 3.0 version of the Event Status Driver.

NAME

Event Status Driver - Keyboard and mouse event driver status and configuration

SYNOPSIS

Keyboard and Mouse Event Status and Configuration
#include <drivers/event_status_driver.h>

DESCRIPTION

The Event Status driver is an interface allowing a program to set the user preference values for the mouse, keyboard, and screen devices.

The Event Status driver can be opened by many programs in order to make calls which will affect the way the mouse, keyboard, and display work.

The following event status calls are available:

Open and Close

NXEventHandle **NXOpenEventStatus(**void**)**

This call opens the event status driver, and returns a handle to be used in future references to the driver. A value of NULL is returned on error.

void **NXCloseEventStatus(**NXEventHandle *handle***)**

This call closes the event status driver, and invalidates the *handle* access token.

General Information/Status Requests

NXEventSystemInfoType **NXEventSystemInfo(**NXEventHandle *handle*,
int *flavor*, int **evs_info*, unsigned int **evs_info_cnt***)**

This call provides an extensible mechanism for querying the event system. The data types and supported queries are defined in <*bsd/dev/ev_types.h*>, which is included by <*drivers/event_status_driver.h*>. The flavor or type of query is passed in *flavor*. A pointer to an array or struct of ints is passed in *evs_info*. The length of the data structure pointed to by *evs_info*, in terms of ints, is passed in *evs_info_count*. The number of ints actually copied into *evs_info* is returned in *evs_info_count*. The call returns a NULL pointer on failure, and *evs_info* cast to a generic information structure on success.

The call is currently implemented as a wrapper around an **ioctl()**, but can be converted to a MiG generated call for future implementations. The slightly odd design of the parameters is due to restrictions imposed by MiG.

The current implementation supports one flavor of query, NX_EVS_DEVICE_INFO. For this call, *evs_info* should be set to the address of a variable of type NXEventSystemDevice. The *evs_info_cnt* variable should be set to NX_EVS_DEVICE_INFO_COUNT. On return, the NXEventSystemDevice variable will contain a list of devices, including types and vendor IDs, which are connected to the event system.

Keyboard Functions

void **NXSetKeyRepeatInterval(**NXEventHandle *handle*, double *seconds***)**

This call sets the key repeat interval for a key on the keyboard which is held down. The double arg to the call gives the number of seconds between successive repeats of the key. If the current key repeat is set to 0.5, for instance, then a key that is held down will repeat twice per second.

double **NXKeyRepeatInterval(**NXEventHandle *handle***)**

This call returns the current key repeat interval in seconds.

void **NXSetKeyRepeatThreshold(**NXEventHandle *handle*, double *threshold***)**

This call sets the initial key repeat interval to the value *threshold*, in seconds. From when a key is first depressed, the initial key repeat interval elapses. If the key goes up during that interval, no additional events are generated. After the initial key repeat interval has elapsed, the first repeated key event is generated (a key-down event with the repeated flag set). Another such event is generated each key repeat interval from then on. The intent of making the initial key repeat interval separate from the key repeat interval is to allow an extra long time before the first repeat, and thus avoid having every key repeat, while still allowing keys to repeat very quickly after they begin repeating.

double **NXKeyRepeatThreshold(**NXEventHandle *handle***)**

This call returns the current initial key repeat interval in seconds.

NX_KeyMapping * **NXSetKeyMapping(**NXEventHandle *handle*, NXKeyMapping **keymap***)**

This call sets the key mapping used by the event system to the one given in the structure pointed to

by (NXKeyMapping *)*keymap*. Key mappings are created by the Keyboard.app demonstration program. The call returns the keymapping, or NULL on failure. Possible causes of failure include invalid, inappropriate, or obsolete key mappings.

int **NXKeyMappingLength(**NXEventHandle *handle***)**

This call returns the length of the current key mapping being used by the event system. This can be used to find out how large an area to allocate for use with the **NXGetKeyMapping** call.

NX_KeyMapping * **NXGetKeyMapping(**NXEventHandle *handle*, NXKeyMapping **keymap***)*

This call takes (NXKeymapping *)*keymap* where the size and pointer given in *keymap* define an area into which the key mapping currently used by the event system are copied. The portion of the key mapping which will fit in the size given in the NXKeyMapping structure pointed to by *keymap* is copied; the remainder of the mapping is thrown away. The number of bytes actually placed in the area is returned in the size component of the NXKeyMapping structure. The call returns the keymapping, or NULL on failure.

void **NXResetKeyboard(**NXEventHandle *handle***)**

This call resets all the user preference items for the keyboard to their initial states. The initial key repeat interval, key repeat interval, and key mapping are all reset to their startup values.

Mouse Functions

void **NXSetClickTime(**NXEventHandle *handle*, double *time***)**

This call sets the click time threshold to the value *time*. The click time threshold is the maximum number of seconds that may elapse between two mouse-down events and still have them be considered a double-click.

double **NXClickTime(**NXEventHandle *handle***)**

This call returns the current click time threshold in seconds.

void **NXSetClickSpace(**NXEventHandle *handle*, NXSize **area***)*

This call sets the current click space threshold in x and y to the values provided in *area*. The click space threshold is the maximum number of pixels apart that two mouse-down events may be and still be considered a double-click. Thus, for a subsequent mouse-down event to be considered a double-click, it must occur within the number of seconds given by the click time threshold since the first mouse-down, and the location where it occurs must be within the distance given by the click space threshold from the original mouse-down. The click space threshold must be met in both x and y.

void **NXGetClickSpace(**NXEventHandle *handle*, NXSize **area***)*

This call returns the current click space threshold in the structure pointed to by its *area* argument.

void **NXSetMouseScaling(**NXEventHandle *handle*, NXMouseScaling **scaling***)*

This call sets the current mouse scaling to the values contained in the NXMouseScaling structure pointed to by its *scaling* arg. When the event system receives a mouse motion event, it looks at the amount of mouse motion that has occurred in its last timing interval (roughly 0.014 seconds). That amount of motion, in pixels, is then compared to the first entry in the scaleThresholds component of the current mouse scaling. If the amount of mouse motion is greater than that threshold, it keeps going to the next entry in the thresholds array. Once the correct entry of the thresholds array is found, the event driver looks in the corresponding element of the scale factors array, found in the scaleFactors component of the structure. It then multiplies the mouse motion by that scale factor. The *scaling* data structure may have up to NX_MAXMOUSESCALINGS entries in its scaleThresholds and scaleFactors arrays.

Mouse Scaling Example

For example, let the numScaleLevels component be 2, and the scaleThresholds array contain 3 and 6, and the scaleFactors array contain 2 and 4. Then, if the mouse moved by a cartesian distance of 4 pixels, it would get to the first element of the thresholds array (3), since it was less than second element (6). This would cause the driver to multiply the mouse motion by the corresponding scale factor, which in this case is 2. Thus the cursor would move twice as far as the mouse. Thus, the affect of mouse scaling is to cause the cursor to move nonlinearally faster than the mouse as both are speeded up. This is very useful on the large screen of a NeXT machine.

void **NXGetMouseScaling(**NXEventHandle *handle*, NXMouseScaling **scaling***)*

This call returns the current mouse scaling thresholds and factors in the NXMouseScaling structure pointed to by *scaling*. The number of entries in these arrays is given by the numScaleLevels component of the structure.

extern void **NXEnableMouseButton(**NXEventHandle *handle*, NXMouseButton *button***)**

This call configures the mouse buttons to be tied (NX_OneButton), to use the left button for menus (NX_LeftButton) and the right button for click events, or to use the right button (NX_RightButton) for menus and the left button for click events. NXMouseButton is a C *enum* with the values NX_OneButton, NX_LeftButton, and NX_RightButton.

extern NXMouseButton **NXMouseButtonEnabled(**NXEventHandle *handle***)**

This call returns the current mouse button configuration.

void **NXResetMouse(**NXEventHandle *handle***)**

This call resets all of the user preference items for the mouse to their default states. The click time threshold, click space threshold, mouse scaling thresholds and factors, the mouse handedness, and the autodim time are all reset. The reset operation undims the display, resets the AutoDim threshold to 1800 seconds, the mouse button mode to NX_OneButton, and restores the startup

mouse scaling, click spacing and timing.

Screen Functions

void **NXSetAutoDimThreshold(**NXEventHandle *handle*, double *threshold***)**

This call sets the autodim period to *threshold* seconds. The autodim period is the number of seconds from the last generated user event until the screen automatically dims to one-fourth of its original value. This is done to prolong the life of the screen phosphor.

double **NXAutoDimThreshold(**NXEventHandle *handle***)**

This call returns the current autodim period in seconds. This is equivalent to the value set by **NXSetAutoDimThreshold()**.

double **NXAutoDimTime(**NXEventHandle *handle***)**

This call returns the current autodim time in seconds. This is the number of seconds until screen dimming will take place. It is determined by the autodim period. The value will be less than or equal to zero if the screen is currently dimmed.

void **NXSetAutoDimState(**NXEventHandle *handle,* BOOL *state***)**

This call sets the current state of auto-dimming. A *state* of YES forces the display to be dimmed. A *state* of NO undims the display and sets the time until autodim to the current autodim threshold.

BOOL **NXAutoDimState(**NXEventHandle *handle***)**

This call returns the current state of auto-dimming. A return value of YES indicates that the display is dimmed. A return value of NO indicates that the display is not dimmed.

void **NXSetScreenBrightness(**NXEventHandle *handle*, double *brightness***)**

This call sets the brightness level of the MegaPixel display to *brightness*, a value between 0.0 and 1.0. This level is immediately reflected on the screen, and is saved in the parameter RAM. Brightness values can range from 0.0 to 1.0 inclusive; if a value outside of this range is passed, it will be clipped to the legal values.

double **NXScreenBrightness(**NXEventHandle *handle***)**

This call returns the current screen brightness level as a double in the range 0.0 to 1.0.

void **NXSetAutoDimBrightness(**NXEventHandle *handle*, double *brightness***)**

This call sets the brightness level to be used when the screen is dimmed. *brightness* should be a value in the range 0.0 to 1.0.

double **NXAutoDimBrightness(**NXEventHandle *handle***)**

This call returns the brightness level to be used when the screen is dimmed, as a double in the range 0.0 to 1.0.

Volume Functions

void **NXSetCurrentVolume(**NXEventHandle *handle*, double *volume***)**

This call sets the attenuation level of the sound output to *level*. This level is immediately reflected in any ongoing sound output, and is saved in the parameter RAM. A volume value of 1.0 results in the maximum volume. Volume levels can range from 0.0 to 1.0 inclusive; if a value outside of this range is passed, it will be clipped to the legal values. The same volume is applied to both left and right sound channels.

double **NXCurrentVolume(**NXEventHandle *handle***)**

This call returns the current volume level in the range 0.0 to 1.0.

Wait Cursor Functions

void **NXSetWaitCursorThreshold(**NXEventHandle *handle*, double *threshold***)**

This call sets the wait cursor threshold (in seconds) to *threshold*. When it has been determined that an application is busy, this is the time that will elapse before putting up the wait cursor.

double **NXWaitCursorThreshold(**NXEventHandle *handle***)**

This call returns the current wait cursor threshold in seconds.

void **NXSetWaitCursorSustain(**NXEventHandle *handle*, double *sustain***)**

This call sets the wait cursor sustain time in seconds. This is the minimum amount of time the wait cursor will remain up.

double **NXWaitCursorSustain(**NXEventHandle *handle***)**

This call returns the current wait cursor sustain time in seconds.

void void **NXSetWaitCursorFrameInterval(**NXEventHandle *handle*, double *rate***)**

This call sets the wait cursor animation frame interval in seconds. This is the amount of time between successive wait cursor frames. A value of zero prevents the wait cursor animation from running.

double **NXWaitCursorFrameInterval(**NXEventHandle *handle***)**

This call returns the current wait cursor frame interval in seconds.

Known Problems

- Mouse button actions and keystrokes can be lost on NeXT computers. This happens less frequently for the NeXT ADB keyboard than for older NeXT keyboards.