DPSAddFD(), DPSRemoveFD()

SUMMARY Monitor a file descriptor

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSAddFD(**int *fd*, DPSFDProc *handler*, void ***userData*, int *priority***)**
void **DPSRemoveFD(**int *fd***)**

DESCRIPTION **DPSAddFD()** registers the function *handler* to be called each time there is activity with the file specified by file descriptor *fd*. The function is called provided the following are true:

- The file descriptor *fd* must be valid and open; typically *fd* is generated through a call to **open()**. There needn't be any data waiting to be read on *fd*.
- *priority*, an integer from 0 to 30, must be equal to or greater than the application's current priority threshold. See **DPSAddTimedEntry()** for a further explanation.

DPSFDProc, *handler*'s defined type, takes the form

```
void *handler(int fd, void *userData)
```

where *fd* is the file descriptor that prompted the function call and *userData* is the same pointer that was passed as the third argument to **DPSAddFD()**. The *userData* pointer is provided as a convenience, allowing you to pass arbitrary data to *handler*.

DPSRemoveFD() removes the specified file descriptor from the list of those that the application will check.

Typically, **DPSAddFD()** is used to listen to a socket or pipe; it's rarely used to monitor a common file.

SEE ALSO **DPSAddPort()**, **DPSAddTimedEntry()**

DPSAddNotifyPortProc(), DPSRemoveNotifyPortProc()

SUMMARY Set the handler function for the notify port

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSAddNotifyPortProc(**DPSPortProc *handler*, void ***userData***)**
void **DPSRemoveNotifyPortProc(**DPSPortProc *handler***)**

DESCRIPTION **DPSAddNotifyPortProc()** registers *handler* as the function that's called when a message arrives on the notify port, the unique port, created through the **task_notify()** Mach function, on which notifications (such as port death) are sent. You don't need to create the notify port yourself; **DPSAddNotifyPortProc()** creates it for you if it doesn't already exist.

DPSPortProc, *handler*'s defined type, takes the form

```
void *handler(msg_header_t *msg, void *userData)
```

where *msg* is a pointer to the message that was received at the port and *userData* is the same pointer that was passed as the second argument to **DPSAddNotifyPortProc()**. The *userData* pointer is

provided as a convenience, allowing you to pass arbitrary data to *handler*.

The notify port can have only one handler at a time; adding a handler removes the current one. You can remove the port's handler without specifying a new one with the **DPSRemoveNotifyPortProc()** function. The function's argument must match the notify port's current handler.

SEE ALSO **DPSAddPort()**, **DPSAddTimedEntry()**

DPSAddPort(), DPSRemovePort()

SUMMARY Monitor a Mach port

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSAddPort(**port_t *port*, DPSPortProc *handler*, int *maxMsgSize*, void **userData*, int *priority***)**
void **DPSRemovePort(**port_t *port***)**

DESCRIPTION **DPSAddPort()** registers the function *handler* to be called each time your application asks for an event or peeks at the event queue. The function is called provided the following are true:

- The Mach port *port* must be valid and it must hold a message waiting to be read.
- *priority*, an integer from 0 to 30, must be equal to or greater than the application's current priority threshold. See **DPSAddTimedEntry()** for a further explanation.

DPSPortProc, *handler*'s defined type, takes the form

```
void *handler(msg_header_t *msg, void *userData)
```

where *msg* is a pointer to the message that was received at the port and *userData* is the same pointer that was passed as the fourth argument to **DPSAddPort()**. The *userData* pointer is provided as a convenience, allowing you to pass arbitrary data to *handler*.

If, within *handler*, you want to call **msg_receive()** to receive further messages at the port, you must first call **DPSRemovePort()** to remove the port from the system's port set. (This is because your application can't receive messages from a port that's in a port set.) After your application is finished receiving messages directly from the port, it can call **DPSAddPort()** to have the system continue to monitor the port.

The contents of the message buffer *msg*, as received by *handler*, are invalid after the function returns. If you need to save any of the information that you find.

The *maxMsgSize* argument is an integer that gives the size, in bytes, of the largest message you expect to receive.

DPSRemovePort() removes the specified Mach port from the list of those that the application will check.

SEE ALSO **DPSAddFD()**, **DPSAddTimedEntry()**

DPSAddTimedEntry(), DPSRemoveTimedEntry()

SUMMARY Create a timed entry

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS DPSTimedEntry **DPSAddTimedEntry(**double *period*, DPSTimedEntryProc *handler*, void **userData*, int *priority***)**
void **DPSRemoveTimedEntry(**DPSTimedEntry *tag***)**

DESCRIPTION **DPSAddTimedEntry()** registers *handler* as a ªtimed entry,º a function that's called repeatedly at a given time interval. *period* determines the number of seconds between calls to the timed entry. Whenever an application based on the Application Kit attempts to retrieve events from the event queue, it also checks (depending on *priority*) to determine whether any timed entries are due to be called. *userData* is a pointer that you can use to pass some data to the timed entry.

The function registered as *handler* has the form:

```
void *handler(DPSTimedEntry tag, double now, void *userData)
```

where *tag* is the timed entry identifier returned by **DPSAddTimedEntry()**, *now* is the number of seconds since some arbitrary point in the past, and *userData* is the pointer **DPSAddTimedEntry()** received when this timed entry was installed.

An application's priority threshold can be set explicitly as an integer from 0 to 31 through a call to **DPSGetEvent()** or **DPSPeekEvent()**. It's against this threshold that *priority* is measured (note that *priority* can be no greater than 30Ðthe additional threshold level, 31, is provided to disallow all inter-event function calls). However, if you're using the Application Kit, you should access the event queue through Application class methods such as **getNextEvent:**. Although some of these methods let you set the priority threshold explicity, you typically invoke the methods that set it automatically. Such methods use only three priority levels:

Constant	Meaning
NX_BASETHRESHOLD	Normal execution
NX_RUNMODALTHRESHOLD	An attention panel is being run
NX_MODALRESPTHRESHOLD	A modal event loop is being run

When applicable, you should use these constants as the value for *priority*. For example, if you want *handler* to be called during normal execution, but not if an attention panel or a modal loop is running, then you would set *priority* to NX_BASETHRESHOLD.

DPSRemoveTimedEntry() removes a previously registered timed entry.

RETURN **DPSAddTimedEntry()** returns a number identifying the timed entry or -1 to indicate an error.

DPSAsynchronousWaitContext()

SUMMARY Procede asynchronously while PostScript code is executed

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSAsynchronousWaitContext(**DPSContext *context*, DPSPingProc *handler*, void **userData***)**

DESCRIPTION This function is similar to the more familiar **DPSWaitContext()** functions, except that rather than wait for all PostScript code to execute, it returns immediately, allowing your application to procede while the PostScript code is executed in the background. The DPSPingProc function *handler* is called (with *context* and *userData* as its two arguments) when all the PostScript code has been executed. The DPSPingProc function takes the form

```
void *handler(DPSContext context, void *userData);
```

Warning: Be careful when you use this function; you mustn't send more PostScript code while waiting for the handler to be called. In general, it's best to not make any demands on the Application Kit or the Client Library if you're waiting for an asynchronous handler to return.

DPSCreateContext(), DPSCreateContextWithTimeoutFromZone(), DPSCreateNonsecureContext(), DPSCreateStreamContext()

SUMMARY Create a PostScript execution context

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS DPSContext **DPSCreateContext(**const char **hostName*, const char **serverName*, DPSTextProc *textProc*, DPSErrorProc *errorProc***)**
DPSContext **DPSCreateContextWithTimeoutFromZone(**const char **hostName*, const char **serverName*, DPSTextProc *textProc*, DPSErrorProc *errorProc*, int *timeout*, NXZone **zone***)**
DPSContext **DPSCreateNonsecureContext(**const char **hostName*, const char **serverName*, DPSTextProc *textProc*, DPSErrorProc *errorProc*, int *timeout*, NXZone **zone***)**
DPSContext **DPSCreateStreamContext(**NXStream **stream*, int *debugging*, DPSProgramEncoding *progEnc*, DPSNameEncoding *nameEnc*, DPSErrorProc *errorProc***)**

DESCRIPTION **DPSCreateContext()** establishes a connection with the Window Server and creates a PostScript execution context in it. The new context becomes the current context. The first argument, *hostName*, identifies the machine that's running the Window Server; the second argument, *serverName*, identifies the Window Server that's running on that machine. With these two arguments and the help of the Mach network server **nmserver**, the Mach port for the Window Server can be identified. If *hostName* is NULL, the network server on the local machine is queried for the Window Server's port. If *serverName* is NULL, a default name for the Window Server is used.

The last two arguments, *textProc* and *errorProc*, refer to call-back functions (defined in the Client Library specification) that handle text returned from the Window Server and errors generated on either side of the connection.

For an application that's based on the Application Kit, you could create an additional context by making this call:

```
DPSContext  c;

c = DPSCreateContext(NXGetDefaultValue([NXApp appName], "NXHost"),
                     NXGetDefaultValue([NXApp appName], "NXPSName"),
                     NULL,
                     NULL);
```

This example queries the application's default values for the indentity of the host machine and the Window Server. By doing this, the new context is created in the correct Window Server even if that Server is not on the same machine as the application process.

The context that **DPSCreateContext()** creates allocates memory from the default allocation zone. Also, when there's difficulty creating the context, **DPSCreateContext()** waits up to 60 seconds before raising an exception. If you want to change either of these parameters, use **DPSCreateContextWithTimeoutFromZone()**. Its two additional arguments let you specify the zone for the context to use when allocating context-specific data and a timeout value in milliseconds.

DPSCreateNonsecureContext() creates a ªnonsecureº context in which you can use PostScript operators that are normally disallowed. The most significant of these are operators that let you write files.

DPSCreateStreamContext() is similar to **DPSCreateContext()**, except that the new context is actually a connection from the client application to a stream. This connection becomes the current context. PostScript code that the application generates is sent to the stream rather than to the Window Server. The first argument, *stream*, is a pointer to an NXStream structure, as created by **NXOpenFile()** or **NXMapFile()**. The *debugging* argument is intended for debugging purposes but is not currently implemented. *progEnc* and *nameEnc* specify the type of program and user-name encodings to be used for output to the stream. The last argument, *errorProc*, identifies the procedure that's called when errors are generated.

Few programmers will need to call either of these functions directly: The Application Kit manages contexts for programs based on the Kit. For example, when an application is launched, its Application object calls **DPSCreateContext()** to create a context in the Window Server. Similarly, to print a View the Kit calls **DPSCreateStreamContext()** to temporarily redirect PostScript code from the View to a stream.

RETURN Each of these functions returns the newly created DPSContext structure.

EXCEPTIONS **DPSCreateContext()** and **DPSCreateContextWithTimeoutFromZone()** raise a dps_err_outOfMemory exception if they encounter difficulty allocating ports or other resources for the new context. They raise a dps_err_cantConnect exception if they can't return a context within the timeout period.

DPSCreateContextWithTimeoutFromZone() → See DPSCreateContext()

DPSCreateNonsecureContext() → See DPSCreateContext()

DPSCreateStreamContext() → See DPSCreateContext()

DPSDefineUserObject(), DPSUndefineUserObject()

SUMMARY Create a user object

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSDefineUserObject(**int *index***)**
void **DPSUndefineUserObject(**int *index***)**

DESCRIPTION **DPSDefineUserObject()** associates *index* with the PostScript object that's on the top of the operand stack, thereby creating a user object (as defined by the PostScript language). If *index* is 0, the object is assigned the next available index number. The function returns the new index, which can then be passed to a **pswrap**-generated function that takes a user object.

Warning: To avoid coming into conflict with user objects defined by the Client Library or Application Kit, use **DPSDefineUserObject()** rather than the PostScript operator **defineuserobject** or the single-operator functions **DPSdefineuserobject()** and **PSdefineuserobject()**.

DPSUndefineUserObject() removes the association between *index* and the PostScript object it refers to, thus destroying the user object. By destroying a user object that's no longer needed, you can let the garbage collector reclaim the previously associated PostScript object.

RETURN **DPSDefineUserObject()**, if successful in assigning an index, returns the index that the object was assigned. If unsuccessful, it returns 0.

DPSDiscardEvents() → See DPSGetEvent()

DPSDoUserPath(), DPSDoUserPathWithMatrix()

SUMMARY Send an encoded PostScript path to the Window Server

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSDoUserPath(**void **coords*, int *numCoords*, DPSNumberFormat *numType*, unsigned char **ops*, int *numOps*, void **bbox*, int *action***)**
void **DPSDoUserPathWithMatrix(**void **coords*, int *numCoords*, DPSNumberFormat *numType*, unsigned char **ops*, int *numOps*, void **bbox*, int *action*, float *matrix*[6]**)**

DESCRIPTION **DPSDoUserPath()** and **DPSDoUserPathWithMatrix()** send an encoded user path to the Window Server and then execute, upon that path, the operator specified by *action*. The use of these functions, rather than the analogous step-by-step path construction, is encouraged; rendering an encoded path is much more efficient than executing the individual PostScript operators that would otherwise be needed.

An encoded user path consists of an array of coordinate values, a sequence of PostScript operators, and a bounding box specification. The values in the coordinate array are used as operands to the operators; the operands are distributed to the operators in the order that they're given. The resulting path must fit within the bounding box.

The coordinates, operators, and bounding box are given by the functions' first five arguments:

- The array of coordinate values is given by *coords*.
- *numCoords* is the number of elements in *coords*.
- *numType* specifies the data type of the coordinates, as described below. All the values in *coords* must be of the same type.
- *ops* is the sequence of PostScript operators, represented by constants as listed below.
- The bounding box is defined by the four coordinate values that you pass as an array in the *bbox* argument. These are passed as operands to the **setbbox** operator. (If you don't supply a **setbbox** as part of the *ops* sequence, one is inserted for you.)

The following integer constants represent the data types that you can pass as the *numType* argument:

Constant	Meaning
dps_float	single-precision floating-point number
dps_long	32-bit integer
dps_short	8-bit integer

You can also specify 16- and 32-bit fixed-point real numbers. For 16-bit fixed-point numbers, use **dps_short** plus the number of bits in the fractional portion. For 32-bit fixed-point numbers, use **dps_long** plus the number of bits in the fractional portion.

These constants are provided for *ops*:

dps_setbbox
dps_moveto
dps_rmoveto
dps_lineto
dps_rlineto
dps_curveto
dps_rcurveto

dps_arc
dps_arcn
dps_arct
dps_closepath
dps_ucache

Once the user path has been constructed, the operator specified by *action* is executed. The value of *action* is an index into Display PostScript's encoded system names; the following constants, provided as a convenience, represent the most commonly used actions:

dps_uappend
dps_ufill
dps_ueofill
dps_ustroke
dps_ustrokepath
dps_inufill
dps_inueofill
dps_inustroke
dps_def
dps_put

DPSDoUserPathWithMatrix()'s *matrix* argument represents the transformation matrix operand used by the **ustroke**, **inustroke**, and **ustrokepath** operators. If *matrix* is NULL, the argument is ignored.

The following program fragment demonstrates the use of **DPSDoUserPath()** as it creates and strokes a user path (an isosceles triangle) within a bounding rectangle whose lower left corner is located at (0, 0) and whose width and height are 200.

```
short   coords[6] = {0, 0, 200, 0, 100, 200};
char    ops[4] = {dps moveto, dps lineto,dps lineto,
                  dps_closepath};
short   bbox[4] = {0, 0, 200, 200};

DPSDoUserPath(coords, 6, dps_short, ops, 4, bbox, dps_ustroke);
```

Note: If an application calls **DPSDoUserPath()** with large values (~10,000-20,000) of *numCoords* and/or *numOps*, it may generate a Display PostScript error.

DPSDoUserPathWithMatrix() → See DPSDoUserPath()

DPSFlush(), DPSSendEOF()

SUMMARY Send PostScript data to the Window Server

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSFlush()**
void **DPSSendEOF(**DPSContext *context***)**

DESCRIPTION **DPSFlush()** flushes the application's output buffer, forcing any buffered PostScript code or data to the Window Server.

DPSSendEOF() sends a PostScript end-of-file marker to the given context. The connection to the context isn't closed or disturbed in any way by this function.

DPSGetEvent(), DPSPeekEvent(), DPSDiscardEvents()

SUMMARY Access events from the Window Server

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSGetEvent(**DPSContext *context*, NXEvent **anEvent*, int *mask*, double *timeout*, int *threshold***)**
int **DPSPeekEvent(**DPSContext *context*, NXEvent **anEvent*, int *mask*, double *timeout*, int *threshold***)**
void **DPSDiscardEvents(**DPSContext *context*, int *mask***)**

DESCRIPTION **DPSGetEvent()** and **DPSPeekEvent()** are macros that access event records in an application's event queue. These routines are provided primarily for programs that don't use the Application Kit. An application based on the Kit should use the corresponding Application class methods (such as **getNextEvent:** and **peekNextEvent:into:**) or the function **NXGetOrPeekEvent()** so that it can be journaled. **DPSDiscardEvents()** removes all event records of a specified type from the queue.

DPSGetEvent() and **DPSPeekEvent()** differ only in how they treat the accessed event record. **DPSGetEvent()** removes the record from the queue after making its data available to the application; **DPSPeekEvent()** leaves the record in the queue.

DPSGetEvent() and **DPSPeekEvent()** take the same parameters. The first, *context*, represents a PostScript execution context within the Window Server. Virtually all applications have only one execution context, which can be returned using **DPSGetCurrentContext()**. Applications having more than one execution context can use the constant DPS_ALLCONTEXTS to access events from all contexts belonging to them.

The second argument, *anEvent*, is a pointer to an event record. If **DPSGetEvent()** or **DPSPeekEvent()** is successful in accessing an event record, the record's data is copied into the storage referred to by *anEvent*.

mask determines the types of events sought. See the section ªTypes and Constantsº for a list of the constants that represent the event type masks. To check for more than one type of event, you combine individual constants using the bitwise OR operator.

If an event matching the event mask isn't available in the queue, **DPSGetEvent()** or **DPSPeekEvent()** waits until one arrives or until *timeout* seconds have elapsed, whichever occurs first. The value of *timeout* can be in the range of 0.0 to NX_FOREVER. If *timeout* is 0.0, the routine returns an event only if one is waiting in the queue when the routine asks for it. If *timeout* is NX_ FOREVER, the routine waits until an appropriate event arrives before returning.

The last argument, *threshold*, is an integer in the range 0 through 31 that determines which other services may be provided during a call to **DPSGetEvent()** or **DPSPeekEvent()**.

Requests for services are registered by the functions **DPSAddTimedEntry()**, **DPSAddPort()**, and **DPSAddFD()**. Each of these functions takes an argument specifying a priority level. If this level is equal to or greater than *threshold*, the service is provided before **DPSGetEvent()** or **DPSPeekEvent()** returns.

DPSDiscardEvents()'s two parameters, *context* and *mask*, are the same as those for **DPSGetEvent()** and **DPSPeekEvent()**. **DPSDiscardEvents()** removes from the application's event queue those records whose event types match *mask* and whose context matches *context*.

RETURN **DPSGetEvent()** and **DPSPeekEvent()** return 1 if they are successful in accessing an event record and 0 if they aren't.

SEE ALSO **DPSAddFD()**, **DPSAddPort()**, **DPSAddTimedEntry()**, **DPSPostEvent()**, **NXGetOrPeekEvent()**

DPSInterruptContext()

Warning: This function is unimplemented in the NEXTSTEP version of the Client Library.

DPSNameFromTypeAndIndex()

SUMMARY Access the system and user name tables

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS const char ***DPSNameFromTypeAndIndex(**short *type*, int *index***)**

DESCRIPTION **DPSNameFromTypeAndIndex()** returns the text associated with *index* from the system or user name table. If *type* is -1, the text is returned from the system name table; if *type* is 0, it's returned from the user name table.

The name tables are used primarily by the Client Library and **pswrap**; few programmers will access them directly.

RETURN This function returns a read-only character string.

DPSPeekEvent() → See DPSGetEvent()

DPSPostEvent()

SUMMARY Create an event

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSPostEvent(**NXEvent **anEvent*, int *atStart***)**

DESCRIPTION **DPSPostEvent()** lets you add an event record to your application's event queue without involving the Window Server. *anEvent* is a pointer to the event record to be added. *atStart* specifies where the new record will be placed in relation to any other records in the queue. If *atStart* is TRUE, the event is posted in front of all others and so will be the next one your application receives. If *atStart* is FALSE, the event is posted behind all others and so won't be returned until events that precede it are processed.

You can free, reuse, or otherwise mangle *anEvent* after you've posted it without fear of corrupting the posted record. **DPSEvent()** copies the record it receives and posts the copy.

Note that event records you post using **DPSPostEvent()** aren't filtered by an event filter function set with **DPSSetEventFunc()**.

RETURN **DPSPostEvent()** returns 0 if successful in posting the event record; it returns -1 if unsuccessful in posting the record because the event queue is full.

SEE ALSO **DPSSetEventFunc()**

DPSPrintError(), DPSPrintErrorToStream()

SUMMARY Print error messages

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSPrintError(**FILE **fp*, const DPSBinObjSeq *error***)**
void **DPSPrintErrorToStream(**NXStream **stream*, const DPSBinObjSeq *error***)**

DESCRIPTION **DPSPrintError()** and **DPSPrintErrorToStream()** format and print error messages received from a PostScript execution context in the Window Server. The error message is extracted from the binary object sequence *error*. **DPSPrintError()** prints the error message to the file identified by *fp*; **DPSPrintErrorToStream()** prints the error message to *stream*.

You rarely need to call these functions directly. However, if you reset the error handler for a PostScript execution context, the new handler you install could use one of these functions to process errors that it receives.

DPSPrintErrorToStream() → See DPSPrintError()

DPSRemoveFD() → See DPSAddFD()

DPSRemovePort() → See DPSAddPort()

DPSRemoveTimedEntry() → See DPSAddTimedEntry()

DPSResetContext()

Warning: This function is unimplemented in the NEXTSTEP version of the Client Library.

DPSSetDeadKeysEnabled()

SUMMARY Allow dead key processing for a context's events

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSSetDeadKeysEnabled(**DPSContext *context*, int *flag***)**

DESCRIPTION **DPSSetDeadKeysEnabled()** turns dead key processing on or off for *context*. If flag is 0, dead key processing is turned off; otherwise, it's turned on (the default).

Dead key processing is a technique for extending the range of characters that can be entered from the keyboard. In NEXTSTEP, it provides one way for users to enter accented characters. For example, a user can type Alternate-e followed by the letter ªeºº to produce the letter ªÝºº. The first keyboard input, Alternate-e, seems to have no effectÐit's the ªdead keyºº. However, it signals Client Library routines that it and the following character should be analyzed as a pair. If, within NEXTSTEP, the pair of characters has been associated with a third character, a keyboard event record representing the third character is placed in the application's event queue, and the first two event records are discarded. If there is no such association between the two characters, the two

event records are added to the event queue.

See the *NeXT User's Reference* manual for a listing of the keys that produce accent characters.

DPSSetEventFunc()

SUMMARY Set the function that filters events

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS DPSEventFilterFunc **DPSSetEventFunc(**DPSContext *context*, DPSEventFilterFunc *func***)**

DESCRIPTION **DPSSetEventFunc()** establishes the function *func* as the function to be called when an event record is returned from the PostScript context *context* in the Window Server. The registered function is called before the event record is put in the event queue. If the registered function returns 0, the record is discarded. If the registered function returns 1, the record is passed on for further processing.

Only event records coming from the Window Server are filtered by the registered function. Records that you post to the event queue using **DPSPostEvent()** aren't affected.

A DPSEventFilterFunc function takes the following form:

```
int *func(NXEvent *anEvent)
```

RETURN **DPSSetEventFunc()** returns a pointer to the previously registered event function. This lets you chain together the current and previous event functions.

SEE ALSO **DPSPostEvent()**

DPSSetTracking()

SUMMARY Coalesce mouse events

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSSetTracking(**int *flag***)**

DESCRIPTION **DPSSetTracking()** turns mouse event-coalescing on or off for the current context. If *flag* is 0, coalescing is turned off; otherwise, it's turned on (the default).

Event coalescing is an optimization that's useful when tracking the mouse. When the mouse is moved, numerous events flow into the event queue. To reduce the number of events awaiting removal by the application, adjacent mouse-moved events are replaced by the most recent event of the contiguous group. The same is done for left and right mouse-dragged events, with the addition that a mouse-up event replaces mouse-dragged events that come before it in the queue.

RETURN **DPSSetTracking()** returns the previous state of the event-coalescing switch.

DPSStartWaitCursorTimer()

SUMMARY Initiate a count down for the wait cursor

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS void **DPSStartWaitCursorTimer()**

DESCRIPTION **DPSStartWaitCursorTimer()** triggers the mechanism that displays a wait cursor when an application is busy and can't respond to user input. In most cases, wait cursor support is automatic: You'll only need to call this function if your application starts a time-consuming operation that's not initiated by a user-generated event.

Client Library routines and the Window Server cooperate to display the wait cursor whenever more than a preset amount of time elapses between the time an application takes an event record from the event queue and the time the application is again ready to consume events. However, when an application starts an operation that isn't initiated by an eventÐsuch as one caused by receiving a Mach message or by processing data from a file (see **DPSAddPort()** and **DPSAddFD()**)Ðthe wait cursor mechanism is bypassed. To ensure proper wait cursor behavior in these cases, call **DPSStartWaitCursorTimer()** before beginning the time-consuming operation.

SEE ALSO **DPSAddFD()**, **DPSAddPort()**, **setwaitcursorenabled**

DPSSynchronizeContext()

SUMMARY Synchronize a context with your application

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSSynchronizeContext(**DPSContext *context*, int *flag***)**

DESCRIPTION **DPSSynchronizeContext()** causes **DPSWaitContext()** to be called after each **pswrap** function is called, thus synchronizing the PostScript context with your application.

DPSTraceContext(), DPSTraceEvents()

SUMMARY Trace data and events

DECLARED IN dpsclient/dpsNeXT.h

SYNOPSIS int **DPSTraceContext(**DPSContext *context*, int *flag***)**
void **DPSTraceEvents(**DPSContext *context*, int *flag***)**

DESCRIPTION **DPSTraceContext()** and **DPSTraceEvents()** control the tracing of data and events between a PostScript execution context (or contexts) in the Window Server and an application process.

The first argument for both functions, *context*, specifies the context to be traced. An application's single context can be returned with **DPSGetCurrentContext()**. Applications having more than one execution context can use the constant DPS_ALLCONTEXTS to trace all contexts belonging to them.

The second argument, *flag*, determines whether tracing is enabled.

When data tracing is enabled (**DPSTraceContext()**), a copy of the PostScript code generated by an application and the values returned to it by the Window Server is sent to UNIX standard error. Values returned to the application are marked by the prepended string:

```
% value returned ==>
```

When event tracing is enabled (**DPSTraceEvents()**), information about each event that the application receives is sent to UNIX standard error. For example, for a left mouse-down event the listing might look like this:

```
Receiving: LMouseDown at: 343.0,69.0 time: 1271899
            flags: 0x0 win: 6 ctxt: 76128 data: 1111,1
```

The listing displays the fields of the event record: type, location, time, flags, local window number, PostScript execution context, and data. The contents of the data field listing depends on the event type; for instance, in the preceding example the event number and the click count were displayed.

For applications based on the Application Kit, there are two preferable methods for turning on data tracing: You can use the NXShowPS command-line switch when you launch an application from Terminal. Alternatively, when you run the application under GDB, you can use the **showps** and **shownops** commands to control tracing output. Similarly, there are more convenient ways to turn on event tracing: You can use the **NXTraceEvents** command-line switch when you launch an application from Terminal. Alternatively, when you run the application under GDB, you can use the **traceevents** and **tracenoevents** commands to control event-tracing output.

Only one tracing context can be created for the supplied *context*. If you attempt to create additional tracing contexts for a context that's already being traced, no new context is created and **DPSTraceContext()** returns -1.

RETURN **DPSTraceContext()** returns 0 if successful in creating a tracing context, or -1 if not.

NX_EVENTCODEMASK()

SUMMARY Generate an event mask for an event type

DECLARED IN dpsclient/event.h

SYNOPSIS int **NX_EVENTCODEMASK(**int *type***)**

DESCRIPTION This handy utility macro returns an event mask that corresponds to the given (single) event type.