NXAllocErrorData(), NXResetErrorData()

SUMMARY Manage the error data buffer

DECLARED IN objc/error.h

SYNOPSIS void **NXAllocErrorData(**int *size*, void ***data***)
void **NXResetErrorData(**void**)**

DESCRIPTION These functions handle the error buffer, which is used to pass error data to an error handler. When an error occurs, **NX_RAISE()** is called with two arguments that point to an arbitrary amount of data about the error. If an error handler can't respond to the error, the error code and associated data are passed to the next higher-level handler.

NXAllocErrorData() allocates *size* amount of space in the error buffer, increasing the size of the buffer if necessary. The *data* argument points to a pointer to the data in the buffer. To empty and free the buffer, call **NXResetErrorData()**. If you're using the Application Kit, the buffer is freed for you upon each pass through the event loop.

SEE ALSO **NX_RAISE()**, **NXDefaultTopLevelErrorHandler()** (Application Kit)

NXAtEOS() → See NXSeek()

NXChangeBuffer() → See NXStreamCreateFromZone()

NXClose()

SUMMARY Close a stream

DECLARED IN streams/streams.h

SYNOPSIS void **NXClose(**NXStream ****stream***)**

DESCRIPTION This function closes the stream given as its argument. If the stream had been opened for writing, it's flushed first. (The NXStream structure is defined in the header file **stream/streams.h**.)

If the stream had been a file stream, the storage used by the stream is freed, but the file descriptor isn't closed. See the UNIX manual page on **close()** for information about closing a file descriptor. If the stream had been opened in memory, the internal buffer is truncated to the size of the data in it. (Calling **NXClose()** on a memory stream is equivalent to **NXCloseMemory()** with the constant NX_TRUNCATEBUFFER.)

EXCEPTIONS **NXClose()** raises an NX_illegalStream exception if the stream passed in is invalid.

SEE ALSO NXCloseMemory()

NXCloseMemory() → See NXOpenMemory()

NXCloseTypedStream() → See NXOpenTypedStream()

NXCompareHashTables() → See NXCreateHashTable()

NXCopyHashTable() → See NXCreateHashTable()

NXCopyStringBuffer() → See NXUniqueString()

NXCopyStringBufferFromZone() → See NXUniqueString()

NXCountHashTable() → See NXHashInsert()

NXCreateChildZone() → See NXCreateZone()

NXCreateHashTable(), NXCreateHashTableFromZone(), NXFreeHashTable(), NXEmptyHashTable(), NXResetHashTable(), NXCopyHashTable(), NXCompareHashTables(), NXPtrHash(), NXStrHash(), NXPtrIsEqual(), NXStrIsEqual(), NXNoEffectFree(), NXReallyFree()

SUMMARY Create and free a hash table

DECLARED IN objc/hashtable.h

SYNOPSIS

NXHashTable ***NXCreateHashTable(**NXHashTablePrototype *prototype*,
unsigned *capacity*, const void **info***)*
NXHashTable ***NXCreateHashTableFromZone(**NXHashTablePrototype *prototype*,
unsigned *capacity*, const void **info*, NXZone **zone***)*
void **NXFreeHashTable(**NXHashTable **table***)*
void **NXEmptyHashTable(**NXHashTable **table***)*
void **NXResetHashTable(**NXHashTable **table***)*
NXHashTable ***NXCopyHashTable(**NXHashTable **table***)*
BOOL **NXCompareHashTables(**NXHashTable **table1*, NXHashTable **table2***)*
unsigned **NXPtrHash(**const void **info*, const void **data***)*
unsigned **NXStrHash(**const void **info*, const void **data***)*
int **NXPtrIsEqual(**const void **info*, const void **data1*, const void **data2***)*
int **NXStrIsEqual(**const void **info*, const void **data1*, const void **data2***)*
void **NXNoEffectFree(**const void **info*, void **data***)*
void **NXReallyFree(**const void **info*, void **data***)*

DESCRIPTION These functions set up, copy, and free a hash table. A hash table provides an efficient means of manipulating elements of an unordered set of data. A data element is stored by computing a hash function—or hashing—on the element to be stored. The value of the hashing function, sometimes called the key, is used to determine the location at which to store the data. The functions described under **NXHashInsert()** insert, remove, and search for a data element; they also count the number of elements and iterate over all elements in a hash table.

To create a hash table, call **NXCreateHashTable()** or **NXCreateHashTableFromZone()**. These functions differ only in that the first one creates the hash table in the default zone, as returned by **NXDefaultMallocZone()**, and the second lets you specify a zone. Only **NXCreateHashTable()** will be discussed below.

The first argument to **NXCreateHashTable()** is a NXHashTablePrototype structure, which is defined in **objc/hashtable.h** and shown below. This structure requires you to specify three functions, a hashing function, a comparison function that determines whether two data elements are equal, and a freeing function that frees a given data element in the table:

```
typedef struct {
    unsigned  (*hash)(const void *info, const void *data);
    int       (*isEqual)(const void *info, const void *data1,
                         const void *data2);
    void      (*free)(const void *info, void *data);
    int         style;
} NXHashTablePrototype;
```

The hashing function must be defined such that if two data elements are equal, as defined by the comparison function, the values produced by hashing on these elements must also be equal. Also, data elements must remain invariant if the value of the hashing function depends on them; for example, if the hashing function operates directly on the characters of a string, that string can't change. The comparison function must return true if and only if the two data elements being compared are equal. The third function specifies how a data element is to be freed. The *style* field is reserved for future use; currently, it should be passed in as 0.

As shown, the third argument for **NXCreateHashTable()**, *info*, is passed as the first argument to the hashing, comparison, and freeing functions. You can use *info* to modify or add to the effects produced by these functions. For example, the comparison function can be modified to return a certain value if the elements being compared are similar to each other but not exactly equal.

For convenience, functions for hashing pointers, integers, and strings and for comparing them have already been defined; two different freeing functions are also provided. **NXPtrHash()** hashes the address bits of *data* and returns a key for storing the data. **NXPtrIsEqual()** returns nonzero if *data1* is equal to *data2* and 0 if they're not equal. These functions can be used for pointers or for data of type **int**. Similarly, **NXStrHash()** returns a key for the string passed in as *data*, and **NXStrIsEqual()** checks whether two strings are equal. **NXReallyFree()** frees the *data* element passed in, allowing its key to be reused. **NXNoEffectFree()**, as its name implies, has no effect.

The *info* argument for all six of these functions isn't used. If you want to hash data other than pointers or strings, or if you want to use the *info* argument, you need to write your own hashing, comparison, and freeing functions.

In addition to the hashing, comparison, and freeing functions, four different prototypes have been predefined. The prototype for pointers (which can also be used for data of type **int**) and the one for strings both use the functions described above:

```
const NXHashTablePrototype NXPtrPrototype = {
    NXPtrHash, NXPtrIsEqual, NXNoEffectFree, 0
};

const NXHashTablePrototype NXStrPrototype = {
    NXStrHash, NXStrIsEqual, NXNoEffectFree, 0
};
```

The following example shows how to use NXPtrPrototype to create a hash table for storing a set of pointers or data of type **int**:

```
NXHashTable  *myHashTable;
myHashTable = NXCreateHashTable(NXPtrPrototype, 0, NULL);
```

Note that you pass the NXPtrPrototype structure as an argument, not a pointer to it. **NXCreateHashTable()** returns a pointer to an NXHashTable structure, which is defined in the header file **objc/hashtable.h**.

The other two prototypes create a hash table for storing a set of structures; the first element of each structure will be used as the key. NXPtrStructKeyPrototype expects the first element to be a pointer, and NXStrStructKeyPrototype expects a string. The free function for both these prototypes is **NXReallyFree()**.

NXCreateHashTable()'s second argument, *capacity,* is only a hint; you can just pass 0 to create a minimally sized table. As more space is needed, it will be automatically and efficiently allocated.

NXFreeHashTable() frees each element of the specified hash table and the table itself. **NXResetHashTable()** frees each element but doesn't deallocate the table. This is useful for retaining the table's capacity. **NXEmptyHashTable()** sets the number of elements in the table to 0 but doesn't deallocate the table or the data in it.

NXCopyHashTable() returns a pointer to a copy of the hash table passed in. **NXCompareHashTables()** returns YES if the two hash tables supplied as arguments are equal. That is, each element of *table1* is in *table2*, and the two tables are the same size.

RETURN **NXCreateHashTable()**, **NXCreateHashTableFromZone()**, and **NXCopyHashTable()** return pointers to the new hash tables they create.

NXCompareHashTables() returns YES if the two hash tables supplied as arguments are equal.

NXPtrHash() returns a key for storing a pointer in a hash table; **NXStrHash()** returns a key for storing a string.

NXPtrIsEqual() and **NXStrIsEqual()** return nonzero if the two data elements passed in are equal, and 0 if they're not.

SEE ALSO NXHashInsert()

NXCreateHashTableFromZone() → See NXCreateHashTable()

NXCreateZone(), NXCreateChildZone(), NXMergeZone(), NXDefaultMallocZone(), NXZoneFromPtr(), NXDestroyZone()

SUMMARY Manage memory zones

DECLARED IN objc/zone.h

SYNOPSIS

NXZone *__NXCreateZone(__size_t *startSize*, size_t *granularity*, int *canFree*__)__
NXZone *__NXCreateChildZone(__NXZone **parentZone*, size_t *startSize*,
size_t *granularity*, int *canFree*__)__
void __NXMergeZone(__NXZone **zone*__)__
NXZone *__NXDefaultMallocZone(__void__)__
NXZone *__NXZoneFromPtr(__void **ptr*__)__
void __NXDestroyZone(__NXZone **zone*__)__

DESCRIPTION These functions set up and manage the memory zones that are used to improve locality of reference. A zone is a region of memory from which functions like **NXZoneMalloc()** can allocate storage. A pointer to a zone is passed to the allocation function, which returns memory from the specified zone. Keeping related data structures together in the same zone reduces the amount of paging activity that otherwise would be required.

NXCreateZone() creates a new zone of *startSize* bytes that will grow and shrink by *granularity* bytes; it returns a pointer to the new zone. The zone will grow as needed as memory is allocated from it, and will shrink as memory is freed. Each time the function is called, it creates and returns a new zone.

Since the point of using zones is to keep data structures together on the same page, small multiples of **vm_page_size** (declared in **mach/mach_init.h**) are a good choice for both

startSize and *granularity.* If these parameters are too large, the benefits of zone allocation can be defeated.

The parameter *canFree* determines whether memory, once allocated, can be freed within the zone. If *canFree* is NO, memory can't be freed and allocation from the zone will be as fast as possible; but you will need to destroy the zone to reclaim the memory.

NXCreateChildZone() creates a new zone that obtains memory from an existing zone, *parentZone*. It returns a pointer to the new zone, or NX_NOZONE if you attempt to create a child zone from a zone which is itself a child. Typically, child zones are used to ensure that a group of data structures are packed together within a larger zone; successive allocations within the child zone are contiguous. The zone is created with a *startSize* sufficient for what it will contain; it can be smaller than a page size. After the allocations are complete, **NXMergeZone()** is called to merge the child zone back into its parent. All memory that was allocated and initialized within the child then resides within the parent zone.

NXDefaultMallocZone() returns the default zone, which is created automatically at startup. This is the zone used by the standard C **malloc()** function.

NXZoneFromPtr() returns the zone for the *ptr* block of memory, or NX_NOZONE if the block was not allocated from a zone. The pointer must be one that was returned by a prior call to an allocation function.

The macro **NXDestroyZone()** destroys a zone; all the memory from the zone is reclaimed.

RETURN

NXCreateZone() and **NXCreateChildZone()** return a pointer to a new zone. **NXDefaultMallocZone()** returns a pointer to the default zone, and **NXZoneFromPtr()** returns the zone for the ptr block of memory. A return of NX_NOZONE indicates that the zone couldn't be created or doesn't exist.

SEE ALSO

NXZoneMalloc()

NXDefaultExceptionRaiser(), NXSetExceptionRaiser(), NXGetExceptionRaiser()

SUMMARY Set and return an exception raiser

DECLARED IN objc/error.h

SYNOPSIS
void **NXDefaultExceptionRaiser(**int *code*, const void **data1*, const void **data2***)**
void **NXSetExceptionRaiser(**NXExceptionRaiser **procedure***)**
NXExceptionRaiser ***NXGetExceptionRaiser(**void**)**

DESCRIPTION These functions set and return the procedure that's called when exceptions are raised using **NX_RAISE()**. By default, the **NXDefaultExceptionRaiser()** will be invoked by **NX_RAISE()**; this function is also what **NXGetExceptionRaiser()** returns unless you've declared your own exception raiser by using **NXSetExceptionRaiser()**, as described below.

NXDefaultExceptionRaiser() forwards the exception condition indicated by *code* and any information about the exception pointed to by *data1* and *data2* to the next error handler. Error handlers exist in a nested hierarchy, which is created by using any number of nested NX_DURING...NX_ENDHANDLER constructs and by defining a top-level error handler.

If the error has occurred outside of the domain of any handler, **NXDefaultExceptionRaiser()** invokes an uncaught exception handling function. For more information on the Application Kit's default uncaught exception handling function or to define your own, see the description of **NXSetUncaughtExceptionHandler()**. If the uncaught exception handling function can't be found, **NXDefaultExceptionRaiser()** exits.

To override the default exception raiser, call **NXSetExceptionRaiser()** and give it a pointer to the exception raising function you want to use. This function must be of type NXExceptionRaiser (that is, the same type as **NXDefaultExceptionRaiser()**), which is defined in the header file **objc/error.h** as follows:

```
typedef void NXExceptionRaiser(int code, const void *data1,
    const void *data2);
```

In other words, the function *procedure* must take three arguments of the types shown above, and it must return **void**. Once you've called **NXSetExceptionRaiser()**, subsequent calls to **NXGetExceptionRaiser()** will return a pointer to *procedure*; also, subsequent calls to **NX_RAISE()** will invoke *procedure*.

SEE ALSO **NX_RAISE()**, **NXSetUncaughtExceptionRaiser()**

NXDefaultMallocZone() → See NXCreateZone()

NXDefaultRead() → See NXStreamCreateFromZone()

NXDefaultWrite() → See NXStreamCreateFromZone()

NXDestroyZone() → See NXCreateZone()

NXEmptyHashTable() → See NXCreateHashTable()

NXEndOfTypedStream()

SUMMARY Determine whether there's more data to be read

DECLARED IN objc/typedstream.h

SYNOPSIS BOOL **NXEndOfTypedStream(**NXTypedStream *_stream_**)**

DESCRIPTION This function indicates whether more data is available to be read from the typed stream passed in as an argument. It should be called only on a typed stream opened for reading. (The NXTypedStream type is declared in the header file **objc/typedstream.h**. The structure itself is private since you never need access to its members.)

RETURN **NXEndOfTypedStream()** returns TRUE if the read operation has reached the end of the stream and no more data is available to be read; returns FALSE otherwise.

EXCEPTIONS **NXEndOfTypedStream()** raises a TYPEDSTREAM_CALLER_ERROR with the message "expecting a reading stream" if the stream passed in wasn't opened for reading.

SEE ALSO **NXOpenTypedStream()**

NXFilePathSearch()

SUMMARY Search for and read a file

DECLARED IN defaults/defaults.h

SYNOPSIS int **NXFilePathSearch(**const char *_envVarName_, const char *_defaultPath_, int _leftToRight_, const char *_filename_, int (*_funcPtr_)(), void *_funcArg_**)**

DESCRIPTION **NXFilePathSearch()** searches a colon-separated list of directories for one or more files named _filename_. The directory list is obtained from the environmental variable, _envVarName_, if it's available. If not, _defaultPath_ is used. If _leftToRight_ is true, the list of directories is searched from left to right; otherwise, it's searched right to left.

In each directory, if the file _filename_ can be accessed, the function specified by _funcPtr_ is called. The function is passed two arguments, the path to the file and _funcArg_, which can contain arbitrary data for the function to use.

RETURN If the function specified by _funcPtr_ is called and returns 0 or a negative value, **NXFilePathSearch()** returns the same value. If the function returns a positive value, **NXFilePathSearch()** continues searching through the directory list for other occurrences of _filename_. If it searches through the entire directory list, it returns 0. If it can't find a list of directories to search, it returns –1.

NXFill() → See NXStreamCreate()

NXFlush()

SUMMARY Flush a stream

DECLARED IN streams/streams.h

SYNOPSIS int **NXFlush(**NXStream *_stream_**)**

DESCRIPTION This function flushes the buffer associated with the stream passed in as an argument. **NXFlush()** is called by **NXClose()**, so you don't have to flush the buffer before closing a stream with **NXClose()**. In some cases, you might not want to close the stream but you might want to ensure that data is actually written to the stream's destination rather than remaining in the buffer.

RETURN **NXFlush()** returns the number of characters flushed from the buffer and written to the stream.

EXCEPTIONS This function raises an NX_illegalStream exception if the stream passed in is invalid. In addition, it raises an NX_illegalWrite exception if an error occurs while flushing the stream.

NXFlushTypedStream()

SUMMARY Flush a typed stream

DECLARED IN objc/typedstream.h

SYNOPSIS void **NXFlushTypedStream(**NXTypedStream ***TypedStream***)**

DESCRIPTION This function flushes the buffer associated with the typed stream passed in as an argument. **NXFlushTypedStream()** is called by **NXCloseTypedStream()**, so you don't have to flush the buffer before closing a typed stream. (The NXTypedStream type is declared in the header file **objc/typedstream.h**. The structure itself is private since you never need to access its members.)

EXCEPTIONS **NXFlushTypedStream()** raises a TYPEDSTREAM_CALLER_ERROR with the message "expecting a writing stream" if the typed stream wasn't opened for writing.

SEE ALSO **NXOpenTypedStream()**

NXFreeHashTable() → See NXCreateHashTable()

NXFreeObjectBuffer() → See NXReadObjectFromBuffer()

NXGetc() → See NXPutc()

NXGetDefaultValue() → See NXRegisterDefaults()

NXGetMemoryBuffer() → See NXOpenMemory()

NXGetTempFilename()

SUMMARY Create a temporary file name

DECLARED IN defaults/defaults.h

SYNOPSIS char ***NXGetTempFilename(**char **name*, int *pos***)**

DESCRIPTION This function creates a unique file name by altering the *name* argument it is passed. **NXGetTempFilename()** replaces the six characters starting at the *pos*th position within *name* with digits it generates; it then checks whether the file name is unique. If it is, the file name is returned; if not, different digits are tried until a unique name is found. **NXGetTempFilename()** is similar to the standard C function **mktemp()**, except that it can leave suffixes intact since you specify the location of the characters that get replaced.

RETURN **NXGetTempFilename()** returns the unique file name it generates.

NXGetTypedStreamZone(), NXSetTypedStreamZone()

SUMMARY Set zones for streams

DECLARED IN objc/typedstream.h

SYNOPSIS NXZone ***NXGetTypedStreamZone(**NXTypedStream **stream***)**
void **NXSetTypedStreamZone(**NXTypedStream **stream*, NXZone **zone***)**

DESCRIPTION These functions let you associate a zone with a typed stream. Zones improve application performance by optimizing locality of reference. See the description under **NXCreateZone()** for more on allocating and freeing zones.

If no zone is set for a typed stream, its zone is the default zone. Use these functions to associate zones with the typed streams used to unarchive objects in your application. You can, for example, use these functions to be sure that objects that interact are all unarchived in the same zone.

Use **NXSetTypedStreamZone()** to set the zone used for unarchiving objects from a typed stream. Use **NXGetTypedStreamZone()** to access the zone associated with a particular typed stream.

RETURN **NXGetTypedStreamZone()** returns the zone set for *stream*.
NXSetTypedStreamZone() sets *zone* as the zone for *stream*.

NXGetUncaughtExceptionHandler() → See NXSetUncaughtExceptionHandler()

NXHashGet() → See NXHashInsert()

NXHashInsert(), NXHashInsertIfAbsent(), NXHashMember(), NXHashGet(), NXHashRemove(), NXCountHashTable(), NXInitHashState(), NXNextHashState()

SUMMARY Manipulate the elements of a hash table

DECLARED IN objc/hashtable.h

SYNOPSIS
void ***NXHashInsert(**NXHashTable **table*, const void **data***)**
void ***NXHashInsertIfAbsent(**NXHashTable **table*, const void **data***)**
int **NXHashMember(**NXHashTable **table*, const void **data***)**
void ***NXHashGet(**NXHashTable **table*, const void **data***)**
void ***NXHashRemove(**NXHashTable **table*, const void **data***)**
unsigned **NXCountHashTable(**NXHashTable **table***)**
NXHashState **NXInitHashState(**NXHashTable **table***)**
int **NXNextHashState(**NXHashTable **table*, NXHashState **state*, void ***data***)**

DESCRIPTION These functions manipulate the elements of a hash table that was created using **NXCreateHashTable()**. **NXCreateHashTable()**, which is described earlier in this

chapter, returns a pointer to the NXHashTable structure it creates. You pass a pointer to this structure (which is defined in the header file **objc/hashtable.h**) for each of the functions described here.

NXHashInsert() inserts *data* into the hash table specified by *table*. It checks whether *data* is already in the table by using the function referred to by the *isEqual* member of the NXHashTablePrototype; this prototype is defined when the table is created. (See the description of **NXCreateHashTable()** for more information about defining the *isEqual* function.) If *data* is already in the table, the new data is inserted anyway and a pointer to the old data is returned. If *data* isn't already in the table, it's inserted and NULL is returned.

NXHashInsertIfAbsent() inserts *data* only if it isn't already in the table and then returns a pointer to *data*. If *data* is already in the table, as determined using the function referred to by *isEqual*, a pointer to the existing data is returned.

NXHashMember() checks whether *data* is in the hash table specified by *table*. If so, it returns a nonzero value; if not, it returns 0. **NXHashGet()** returns a pointer to *data* if it's in the table; if not, it returns NULL. You can use these functions if you have a pointer to the data that might be stored in the table. You can also use them if data is stored in the table as a structure containing the key for that data and if you have that key. (In a hash table, the key determines where data is stored.) For example, suppose my hash table contains data of type MyStruct and that you have a key:

```
typedef struct {
    MyKey  key;
    . . .
    } MyStruct;

MyStruct pseudo;
pseudo.key = yourKey;
```

You can then use your key on my hash table with either function:

```
int        foundIt;
foundIt = NXHashMember(myTable, &pseudo);

MyStruct  *storedData;
storedData = NXHashGet(myTable, &pseudo);
```

NXHashRemove() removes and returns a pointer to *data* unless it can't find *data* in the table, in which case it returns NULL.

NXCountHashTable() returns the number of elements in the hash table specified by *table*.

NXInitHashState() and **NXNextHashState()** iterate through the elements of a hash table. **NXInitHashState()** returns an NXHashState structure to start the iteration process; this

structure is then passed to **NXNextHashState()**, which visits each element of the hash table and finally returns 0. (NXHashState is defined in the header file **objc/hashtable.h**; you shouldn't use members of this structure as they may change in the future.) The following example counts the elements in the hash table **table**:

```
unsigned count = 0;
MyData  *data;
NXHashState state = NXInitHashState(table);

while (NXNextHashState(table, &state, &data))
    count++;
```

As it progresses through the table, **NXNextHashState()** reads each element of the table into the location specified by its third argument.

RETURN **NXHashInsert()** returns NULL if the given data isn't already in the table. Otherwise, it returns a pointer to the existing data.

NXHashInsertIfAbsent() returns a pointer to the given data if it isn't already in the table. Otherwise, a pointer to the existing data is returned.

NXHashMember() returns a nonzero value if it finds the given data in the hash table specified; if not, it returns 0.

NXHashGet() returns a pointer to the given data if it's in the table; if not, it returns NULL.

NXHashRemove() returns a pointer to the data it removes unless it can't find the data, in which case it returns NULL.

NXCountHashTable() returns the number of elements in the hash table.

NXInitHashState() returns an NXHashState for use with **NXNextHashState()**.

NXNextHashState() returns 0 when it has visited every element of the hash table.

SEE ALSO NXCreateHashTable()

NXHashInsertIfAbsent() → See NXHashInsert()

NXHashMember() → See NXHashInsert()

NXHashRemove() → See NXHashInsert()

NXInitHashState() → See NXHashInsert()

NXIsAlNum() → See NXIsAlpha()

NXIsAlpha(), NXIsAlNum(), NXIsCntrl(), NXIsDigit(), NXIsGraph(), NXIsLower(), NXIsPrint(), NXIsPunct(), NXIsSpace(), NXIsUpper(), NXIsXDigit(), NXIsAscii()

SUMMARY Classify NEXTSTEP-encoded values

DECLARED IN appkit/NXCType.h

SYNOPSIS
int **NXIsAlpha(**unsigned int *c***)**
int **NXIsAlNum(**unsigned int *c***)**
int **NXIsUpper(**unsigned int *c***)**
int **NXIsLower(**unsigned int *c***)**
int **NXIsDigit(**unsigned int *c***)**
int **NXIsXDigit(**unsigned int *c***)**
int **NXIsSpace(**unsigned int *c***)**
int **NXIsPunct(**unsigned int *c***)**
int **NXIsPrint(**unsigned int *c***)**
int **NXIsGraph(**unsigned int *c***)**
int **NXIsCntrl(**unsigned int *c***)**
int **NXIsAscii(**unsigned int *c***)**

DESCRIPTION These functions classify NEXTSTEP-encoded integer values. They return a nonzero value if the tested value belongs to the indicated class of characters or 0 if it does not.

These functions are similar to the standard C library routines for testing ASCII-encoded integer values (see the **ctype**(3) UNIX manual page), except that they act on the extended character set defined by NEXTSTEP encoding. For example, both **isalpha()** and **NXIsAlpha()** classify the character “a” as a letter; however, only **NXIsAlpha()** classifies “â” as a letter. The functions make these tests:

Function	Tests whether *c* is:
NXIsAlpha(*c*)	a letter
NXIsUpper(*c*)	an uppercase letter
NXIsLower(*c*)	a lowercase letter
NXIsDigit(*c*)	a digit
NXIsXDigit(*c*)	a hexadecimal digit
NXIsAlNum(*c*)	an alphanumeric character
NXIsSpace(*c*)	a space, tab, carriage return, newline, vertical tab, or formfeed
NXIsPunct(*c*)	a punctuation character (neither control nor alphanumeric)
NXIsPrint(*c*)	a printing character
NXIsGraph(*c*)	a printing character; like **NXIsPrint()** except false for space
NXIsCntrl(*c*)	a control character (0x00 through 0x1F, 0x7F, 0x80, 0xFE, 0xFF)
NXIsAscii(*c*)	an ASCII character (code less than 0x7F)

RETURN Each of these functions returns a nonzero value if the tested value belongs to the indicated class of characters or 0 if it does not.

SEE ALSO NXToAscii()

NXIsAscii() → See NXIsAlpha()

NXIsCntrl() → See NXIsAlpha()

NXIsDigit() → See NXIsAlpha()

NXIsGraph() → See NXIsAlpha()

NXIsLower() → See NXIsAlpha()

NXIsPrint() → See NXIsAlpha()

NXIsPunct() → See NXIsAlpha()

NXIsSpace() → See NXIsAlpha()

NXIsUpper() → See NXIsAlpha()

NXIsXDigit() → See NXIsAlpha()

NXLoadLocalizedStringFromTableInBundle() → See NXLocalizedString()

NXLocalizedString(), NXLocalizedStringFromTable(), NXLocalizedStringFromTableInBundle(), NXLoadLocalizedStringFromTableInBundle()

SUMMARY Get localized versions of strings

DECLARED IN objc/NXBundle.h

SYNOPSIS

const char ***NXLocalizedString(**const char **key*, const char **value*, *comment***)**
const char ***NXLocalizedStringFromTable(**const char **table*, const char **key*, const char **value*, *comment***)**
const char ***NXLocalizedStringFromTableInBundle(**const char **table*, NXBundle **bundle*, const char **key*, const char **value*, *comment***)**
const char ***NXLoadLocalizedStringFromTableInBundle(**const char **table*, NXBundle **bundle*, const char **key*, const char **value***)**

DESCRIPTION These three macros and one function select a localized string to display to the user. They each look up the *key* string in a table and return a matching string in a language of the user's preference. For example, if the key is "Cancel" and the user's preferred language is French, the string returned might be "Annuler"; if the user's preferred language is German, the same key might designate "Abbrechen". Users choose their preferred languages in the Preferences application.

To localize your application—to permit it to be run in more than one language—you must (1) keep the compiled code free of any user-visible strings and (2) provide resource files containing those strings in all the languages you're willing to support. Language-specific resources are kept in *Language***.lproj** subdirectories of a *bundle* directory that can be managed by an NXBundle object. Most applications keep ".lproj" subdirectories in the file package that contains the application executable. This file package is a directory named after the application and assigned a ".app" extension. When it contains resource files and ".lproj" subdirectories, it's also known as the *main bundle*. An application can be organized into additional bundle directories, each with its own set of subdirectories, inside the main bundle. (See the description of the NXBundle class for more on bundle directories.)

Each ".lproj" subdirectory of a bundle bears the name of a language—such as **English.lproj**, **French.lproj**, or **German.lproj**—and stores resources specific to that language. Every resource file is repeated (with the identical name) in every subdirectory of the bundle. In addition to strings that are displayed to the user, localized resources include images, sounds, and nib files produced by Interface Builder.

One kind of resource in a ".lproj" subdirectory is a string table—identified by a ".strings" extension on the file name. Entries in a string table look like this,

```
[ /* comment */ ]
"key" [ = "value" ] ;
```

where the square brackets indicate that the comment and value are optional. The key is a string that's used to identify the entry; it must be unique within a file. The value is the localized string that's matched to the key. If the key and value strings are identical, the value string can be omitted. The comment is typically an explanation that would aid translators preparing correct versions of the string in other languages.

For example, an **English.lproj** subdirectory might contain a **my.strings** file with this entry:

```
/* unable to open a file; %s is the file name */
"open failure"="Can't open %s";
```

In **French.lproj**, the **my.strings** file might have this entry:

```
/* unable to open a file; %s is the file name */
"open failure"="Ouverture de %s impossible";
```

And in **German.lproj**, the entry could look like this:

```
/* unable to open a file; %s is the file name */
"open failure"="%s kann nicht geöffnet werden";
```

The **NXLoadLocalizedStringFromTableInBundle()** function searches for a localized version of the string designated by *key*. It looks only in the bundle directory managed by the *bundle* object and in the string table named *table*, which may or may not include the ".strings" extension. If *bundle* is **nil**, it looks in the main bundle; if *table* is NULL, it looks for a file named **Localizable.strings**.

The search starts with the ".lproj" subdirectory of the user's most preferred language and continues down the ordered list of language preferences until the *table* file is found. (If *table* occurs in every subdirectory, it should be found for the user's preferred language, provided the application is localized for that language.) If *table* can't be found in any ".lproj" subdirectory, the function looks for it in the bundle directory itself.

If a *key* entry is found in the string table, the function returns the matching value string (the string in the entry after the equal sign). If a value string is absent from the entry, it returns the key string. If the string table can't be found, or if the table lacks an entry for *key*, it returns the default *value* passed to the function as an argument. If *value* is NULL, it returns *key*.

The three macros are defined on the **NXLoadLocalizedStringFromTableInBundle()** function and do just what it does. However, they're preferred to the function since, in combination with the **genstrings** utility, they can aid in constructing string tables. **genstrings** searches for each occurrence of the macros in source code and constructs string table entries from the *key*, *value*, and *comment* arguments it finds. The *comment* argument can simply be information for translators who might render localized versions of the entry; it's discarded by the preprocessor and is not passed to the function. **genstrings** writes the entries into the *table* file, creating the file and adding the ".strings" extension if necessary. In the case of **NXLocalizedString()**, which doesn't have a *table* argument, it writes the results to the standard output For example, from this code,

```
char *s;
s = NXLocalizedStringFromTable("my", "open failure", "Can't open %s",
unable to open a file; %s is the file name);
```

genstrings would construct the string table entry illustrated earlier and put it in the **my.strings** file. The **genstrings** utility is more fully documented on-line, in the file **Localization.rtfd** under the **/NextLibrary/Documentation/NextDev/Concepts** directory.

The **NXLocalizedStringFromTableInBundle()** macro works just like the **NXLoadLocalizedStringFromTableInBundle()** function, except that it provides source material for **genstrings**. The **NXLocalizedStringFromTable()** macro looks for the *key* string in the *table* file in the main bundle. The **NXLocalizedString()** macro, the simplest of the three to use, looks for the *key* string in the string table named **Localizable.string** in the main bundle.

RETURN The function and all three macros return a localized string designated by *key*, or *value* if the string can't be found, or *key* if the string can't be found and *value* is NULL.

NXLocalizedStringFromTable() → See NXLocalizedString()

NXLocalizedStringFromTableInBundle() → See NXLocalizedString()

NXMallocCheck(), NXNameZone(), NXZonePtrInfo()

SUMMARY Aid in debugging memory allocation

DECLARED IN objc/zone.h

SYNOPSIS int **NXMallocCheck(**void**)**
void **NXNameZone(**NXZone *_zone_, const char *_name_**)**
void **NXZonePtrInfo(**void *_ptr_**)**

DESCRIPTION These functions assist in debugging memory allocation problems. **NXMallocCheck()** verifies all internal memory-allocation information, and returns 0 if there are no inconsistencies or errors. This function is used by **malloc_debug()**. **NXNameZone()** assigns *name* to *zone*. **NXZonePtrInfo()** prints various information about the *ptr* memory block to the standard output. The information includes the name of the zone, if one was assigned by **NXNameZone()**.

SEE ALSO **NXZoneMalloc()**, **NXCreateZone()**

NXMapFile() → See NXOpenMemory ()

NXMergeZone() → See NXCreateZone()

NXNameZone() → See NXMallocCheck()

NXNextHashState() → See NXHashInsert()

NXNoEffectFree() → See NXCreateHashTable()

NXOpenFile(), NXOpenPort()

SUMMARY Open a file stream or a Mach port stream

DECLARED IN streams/streams.h

SYNOPSIS
NXStream ***NXOpenFile(**int *fd*, int *mode***)**
NXStream ***NXOpenPort(**port_t *port*, int *mode***)**

DESCRIPTION These functions connect a stream to a file or a Mach port. (The NXStream structure is defined in the header file **streams/streams.h**.)

NXOpenFile() opens a stream on the file specified by the file descriptor argument, *fd*, which can refer to a pipe or a socket. (If the file is stored on disk, use **NXMapFile()**; this function is described below under **NXOpenMemory()**.) The *mode* argument should be one of the three constants NX_READONLY, NX_WRITEONLY, or NX_READWRITE to specify how the stream will be used. The mode should be the same as the one used when obtaining the file descriptor. (The system call **open()**, which returns a file descriptor, takes 0_RDONLY, 0_WRONLY, or 0_RDWR to indicate whether the file will be used for reading, writing, or both. For more information on this function, see its UNIX manual page.)

You can use **NXOpenFile()** to connect to **stdin**, **stdout**, and **stderr** by obtaining their file descriptors using the standard C library function **fileno()**. (For more information on this function, see its UNIX manual page.)

NXOpenPort() opens a stream associated with the Mach port specified by *port*. The *mode* must be either NX_READONLY or NX_WRITEONLY. The port must already be allocated using the Mach function **port_allocate()**. See the *NEXTSTEP Operating System Software* manual for more information about using this function.

Once the file or Mach port stream is open, you can read from or write to it. See the descriptions of **NXRead()** and **NXPutc()** for more information about the functions available for reading or writing to a stream.

When you're finished with the stream, close it with **NXClose()**. If you've written to the stream, the data will be automatically saved in the file. After calling **NXClose()** on a file stream, you still need to close the file descriptor. To do this, use the system call **close()**, giving it the file descriptor as an argument. (For more information about **close()**, see its UNIX manual page.)

RETURN Both functions return a pointer to the stream they open or NULL if an error occurred while trying to open the stream.

SEE ALSO **NXOpenMemory()**, **NXRead()**, **NXPutc()**, **NXClose()**

NXOpenMemory(), NXMapFile(), NXSaveToFile(), NXGetMemoryBuffer(), NXCloseMemory()

SUMMARY Manipulate a memory stream

DECLARED IN streams/streams.h

SYNOPSIS
NXStream ***NXOpenMemory(**const char **address*, int *size*, int *mode***)**
NXStream ***NXMapFile(**const char **pathName*, int *mode***)**
int **NXSaveToFile(**NXStream **stream*, const char **name***)**
void **NXGetMemoryBuffer(**NXStream **stream*, char ***streambuf*, int **len*, int **maxlen***)**
void **NXCloseMemory(**NXStream **stream*, int *option***)**

DESCRIPTION These functions open, save, and close streams on memory. (The NXStream structure is defined in the header file **streams/streams.h**.)

NXOpenMemory() returns a pointer to the memory stream it opens. Its argument *mode* specifies whether the stream will be used for reading or writing. If NX_WRITEONLY is specified, the first two arguments should be NULL and 0 to allow the amount of memory available to be automatically adjusted as more data is written. Any other value for *address* should be the starting address of memory allocated with **vm_allocate()**. If NX_READONLY is specified, a memory stream will be set up for reading the data beginning at the location specified by the first argument; the second argument indicates how much data will be read. To use the stream for both writing and reading, you can either use NULL and 0 or specify the location and amount of data to be read; again, *address* should be the starting address of memory allocated with **vm_allocate()**.

NXMapFile() maps a file into memory and then opens a memory stream. A related function, **NXOpenFile()**, connects a stream to a file specified with a file descriptor. (This function is described earlier in this chapter.) Memory mapping allows efficient random and multiple access to the data in the file, so **NXMapFile()** should be used whenever the file is stored on disk. When you call **NXMapFile()**, give it the pathname for the file and indicate whether you will be writing, reading, or both, by using one of the *mode* constants described above. If you use the stream only for reading, just close the memory stream when you're

finished. If you write to the memory-mapped stream, you need to call **NXSaveToFile()**, as described below, to save the data. If you try to map a file that doesn't exist, this function returns a NULL stream.

Once the memory stream is open, you can read from or write to it. See the descriptions of **NXRead()** and **NXPutc()** for more information about reading or writing to a stream.

Before you close a memory stream, you can save data written to the stream in a file. To do this, call **NXSaveToFile()**, giving it the stream and a pathname as arguments. **NXSaveToFile()** writes the contents of the memory stream into the file, creating it if necessary. After saving the data, close the stream using **NXCloseMemory()**.

NXGetMemoryBuffer() returns the memory buffer (*streambuf*) and its current and maximum lengths (*len* and *maxlen*).

When you're finished with a memory stream, close it by calling **NXCloseMemory()**. If you've used the stream for writing, more memory may have been made available than was actually used; the constant NX_TRUNCATEBUFFER indicates that any unused pages of memory should be freed. (Calling **NXClose()** with a memory stream is equivalent to calling **NXCloseMemory()** and specifying NX_TRUNCATEBUFFER.) NX_SAVEBUFFER doesn't free the memory that had been made available. **NXCloseMemory()** doesn't free the internal buffer: Use **NXGetMemoryBuffer()** to get the internal buffer and use **vm_deallocate()** to free it.

RETURN **NXOpenMemory()** and **NXMapFile()** return a pointer to the stream they open or NULL if the stream couldn't be opened.

NXSaveToFile() returns −1 if an error occurred while opening or writing to the file and 0 otherwise.

EXCEPTIONS The functions in this group that take a stream as an argument raise an NX_illegalStream exception if the stream is invalid. This exception is also raised if these functions are used on a stream that isn't a memory stream.

SEE ALSO **NXRead()**, **NXPutc()**, **NXOpenFile()**

NXOpenPort() → See NXOpenFile()

NXOpenTypedStream(), NXCloseTypedStream(), NXOpenTypedStreamForFile()

SUMMARY Open or close a typed stream

DECLARED IN objc/typedstream.h

SYNOPSIS
NXTypedStream ***NXOpenTypedStream(**NXStream **stream*, int *mode***)**
void **NXCloseTypedStream(**NXTypedStream **stream***)**
NXTypedStream ***NXOpenTypedStreamForFile(**const char **filename*, int *mode***)**

DESCRIPTION These functions open, save the contents of, and close a typed stream. A typed stream should be used for archiving—that is, for saving Objective C objects for later use, typically in a file. (The NXTypedStream type is declared in the header file **objc/typedstream.h**. The structure itself is private since you never need to access its members.)

The first argument for **NXOpenTypedStream()** is an already opened NXStream structure. See the descriptions of **NXOpenMemory()**, **NXOpenFile()**, and **NXOpenPort()** earlier in this chapter for more information about opening a stream. The second argument to **NXOpenTypedStream()** must be NX_READONLY or NX_WRITEONLY to specify how the typed stream will be used.

Once the typed stream is open, you can write to or read from it. See the descriptions of **NXReadType()**, **NXReadObject()**, and **NXReadPoint()** later in this chapter for more information about reading and writing. When you're finished with the typed stream, you must first close the typed stream using **NXCloseTypedStream()** and then close the NXStream structure. See the descriptions of **NXClose()** and **NXCloseMemory()** for more information about closing a stream.

To open a typed stream on a file, use **NXOpenTypedStreamForFile()**. This function opens a memory stream and an associated typed stream. If *mode* is NX_READONLY, the typed stream is initialized with the contents of the file specified by *filename*; if the named file doesn't exist or doesn't contain a typed stream, the function returns NULL . A subsequent call to **NXCloseTypedStream()** will close the NXTypedStream and NXStream structures and free the buffer that had been used. If *mode* is NX_WRITEONLY, a typed stream on memory is opened, ready for writing. When you finish writing, calling **NXCloseTypedStream()** will flush the typed stream, save its contents in the file specified by *filename*, close both the NXTypedStream and the NXStream structures, and free the buffer used.

Note: The *filename* argument to **NXOpenTypedStreamForFile()** is stored as a pointer. If the file is opened in NX_WRITEONLY mode, the referenced file isn't actually opened for writing until **NXCloseTypedStream()** is called. Thus if the string pointed to by *filename* changes between these two function calls, the data will be written to the file of the new name. **NXCloseTypedStream()** will raise an exception if *filename* can't be opened for writing.

RETURN **NXOpenTypedStream()** and **NXOpenTypedStreamForFile()** return a pointer to the typed stream they open or NULL if the stream couldn't be opened.

EXCEPTIONS **NXOpenTypedStream()** and **NXOpenTypedStreamForFile()** raise a TYPEDSTREAM_CALLER_ERROR exception with the message "NXOpenTypedStream: invalid mode" if the mode is anything other than NX_READONLY or NX_WRITEONLY.

NXOpenTypedStream() raises a TYPEDSTREAM_CALLER_ERROR exception with the message "NXOpenTypedStream: null stream" if an invalid NXStream structure is passed.

SEE ALSO **NXOpenMemory()**, **NXOpenFile()**, **NXClose()**, **NXCloseMemory()**, **NXReadType()**, **NXReadObject()**, **NXReadPoint()**

NXOpenTypedStreamForFile() → See NXOpenTypedStream()

NXPrintf() → See NXPutc()

NXPtrHash() → See NXCreateHashTable()

NXPtrIsEqual() → See NXCreateHashTable()

NXPutc(), NXGetc(), NXUngetc(), NXScanf(), NXPrintf(), NXVScanf(), NXVPrintf()

SUMMARY Read or write formatted data to or from a stream

DECLARED IN streams/streams.h

SYNOPSIS
int **NXPutc(**NXStream *_stream_, char _c_**)**
int **NXGetc(**NXStream *_stream_**)**
void **NXUngetc(**NXStream *_stream_**)**
int **NXScanf(**NXStream *_stream_, const char *_format_, ...**)**
void **NXPrintf(**NXStream *_stream_, const char *_format_, ...**)**
int **NXVScanf(**NXStream *_stream_, const char *_format_, va_list _argList_**)**
void **NXVPrintf(**NXStream *_stream_, const char *_format_, va_list _argList_**)**

DESCRIPTION These functions and macros read and write data to and from a stream that has already been opened. (See the descriptions of **NXOpenMemory()** and **NXOpenFile()** for more information about opening a stream.) After writing to a stream, you may need to call **NXFlush()** to flush data from the buffer associated with the stream. (See the description of **NXFlush()** earlier in this chapter.)

The macros for writing and reading single characters at a time are similar to the corresponding standard C functions: **NXPutc()** and **NXGetc()** work like **putc()** and **getc()**. **NXPutc()** appends a character to the stream. Its second argument specifies the character to be written to the stream. **NXGetc()** retrieves the next character from the stream. To reread a character, call **NXUngetc()**. This function puts the last character read back onto the stream. **NXUngetc()** doesn't take a character as an argument as **ungetc()** does. **NXUngetc()** can only be called once between any two calls to **NXGetc()** (or any other reading function).

The other four functions convert strings of data as they're written to or read from a stream. **NXPrintf()** and **NXScanf()** take a character string that specifies the format of the data to be written or read as an argument. **NXPrintf()** interprets its variables according to the format string and writes them to the stream. Similarly, **NXScanf()** reads characters from the stream, interprets them as specified in the format string, and stores them in the variables indicated by the last set of arguments. The conversion characters in the format string for both functions are the same as those used for the standard C library functions, **printf()** and **scanf()**. For detailed information on these characters and how conversions are performed, see the UNIX manual pages for **printf()** and **scanf()**.

Two related functions, **NXVPrintf ()** and **NXVScanf()**, are exactly the same as **NXPrintf()** and **NXScanf()**, except that instead of being called with a variable number of arguments, they are called with a **va_list** argument list, which is defined in the header file **stdarg.h**. This header file also defines a set of macros for advancing through a **va_list**.

RETURN **NXPutc()** and **NXGetc()** return the character written or read. **NXScanf()** and **NXVScanf()** return EOF if all data was successfully read; otherwise, they return the number of successfully read data items.

SEE ALSO **NXOpenMemory()**, **NXOpenFile()**, **NXFlush()**, **NXRead()**

NXRead(), NXWrite()

SUMMARY Read from or write to a stream

DECLARED IN streams/streams.h

SYNOPSIS int **NXRead(**NXStream ***stream*, void ***buf*, int *count***)**
int **NXWrite(**NXStream ***stream*, const void ***buf*, int *count***)**

DESCRIPTION These macros read and write multiple bytes of data to a stream that has already been opened. (See the descriptions of **NXOpenMemory()** and **NXOpenFile()** for more information about opening a stream.) After writing to a stream, you may need to call **NXFlush()** to flush data from the buffer associated with the stream. (See the description of **NXFlush()** earlier in this chapter.)

To read data from a stream, call **NXRead()**:

```
NXRect        myRect;
NXRead(stream, &myRect, sizeof(NXRect));
```

NXRead() reads the number of bytes specified by its third argument from the given stream and places the data in the location specified by the second argument.

In the following example, an NXRect structure is written to a stream.

```
NXRect  myRect;

NXSetRect(&myRect, 0.0, 0.0, 100.0, 200.0);
NXWrite(stream, &myRect, sizeof(NXRect));
```

The second and third arguments for **NXWrite()** give the location and amount of data (measured in bytes) to be written to the stream.

RETURN These macros return the number of bytes written or read. If an error occurs while writing or reading, not all the data will be written or read.

SEE ALSO NXFlush()

NXReadArray(), NXWriteArray()

SUMMARY Read or write arrays from or to a typed stream

DECLARED IN objc/typedstream.h

SYNOPSIS
void **NXReadArray(**NXTypedStream *_stream_, const char *_dataType_, int _count_, void *_data_**)**
void **NXWriteArray(**NXTypedStream *_stream_, const char *_dataType_, int _count_, const void *_data_)

DESCRIPTION These functions read and write arrays from and to a typed stream. They can be used within **read:** or **write:** methods for archiving purposes. See the description of **NXReadObject()** in this chapter for more about these methods. Functions are also available for reading and writing other data types; they're listed below in the "See Also" section.

Before using a typed stream for reading and writing, it must be opened; see the description of **NXOpenTypedStream()** for details on opening a typed stream. (The NXTypedStream type is declared in the header file **objc/typedstream.h**. The structure itself is private since you never need access to its members.)

NXReadArray() and **NXWriteArray()** read and write an array of *count* elements of type *dataType* from or to *stream*. **NXReadArray()** reads the array from the typed stream into the location specified by *data*, which must have been previously allocated. **NXWriteArray()** writes the array specified by *data* to the typed stream. Both functions use the characters listed under the description of **NXReadType()** for *dataType*.

The following is an example of an integer array being written. To read the same array, **NXReadArray()** would be called with the same first three arguments as **NXWriteArray()**; the fourth argument would be a pointer to memory for the array.

```
int  aa[4];

aa[0] = 0; aa[1] = 11; aa[2] = 22; aa[3] = 33;
NXWriteArray(typedStream, "i", 4, aa);
```

EXCEPTIONS Both functions check whether the typed stream has been opened for reading or for writing and raise a TYPEDSTREAM_FILE_INCONSISTENCY exception if the mode isn't correct. For example, if **NXReadArray()** is called and the stream was opened for writing, the exception is raised.

NXReadArray() raises a TYPEDSTREAM_FILE_INCONSISTENCY exception if the data to be read is not of the expected type.

SEE ALSO **NXOpenTypedStream()**, **NXReadType()**, **NXReadObject()**, and **NXReadPoint()**

NXReadDefault() → See NXRegisterDefaults()

NXReadObject(), NXWriteObject(), NXWriteObjectReference(), NXWriteRootObject()

SUMMARY Read or write Objective C objects from or to a typed stream

DECLARED IN objc/typedstream.h

SYNOPSIS id **NXReadObject(**NXTypedStream *_stream_**)**
void **NXWriteObject(**NXTypedStream *_stream_, id _object_**)**
void **NXWriteObjectReference(**NXTypedStream *_stream_, id _object_**)**
void **NXWriteRootObject(**NXTypedStream *_stream_, id _rootObject_**)**

DESCRIPTION These functions initiate the archiving and unarchiving processes for Objective C objects. They read and write the object passed in from or to *stream*. When an object is archived with these functions, its class is automatically written as well. In addition, the data type of each of its instance variables is archived along with the value of the variable. These functions also ensure that objects are written only once.

Before you use a typed stream for reading and writing, it must be opened; see the description of **NXOpenTypedStream()** for details on opening a typed stream. (The

NXTypedStream type is declared in the header file **objc/typedstream.h**. The structure itself is private since you never need to access its members.)

NXReadObject() begins the unarchival process by allocating memory for a new object of the correct class. It then sends the object a **read:** message to initialize its instance variables from the typed stream. **read:** messages should only be generated through **NXReadObject()**; they shouldn't be sent directly to objects. Application Kit objects already have **read:** methods, but you need to implement **read:** methods for any classes you create that add instance variables:

```
- read:(NXTypedStream *)typedStream
{
    [super read:typedStream];
    . . ./* code for reading instance variables declared in
            this class */
}
```

The message to **super** ensures that inherited instance variables will be unarchived. The body of the **read:** method unarchives the object's instance variables, using the appropriate function for that data type. The functions available for unarchiving include **NXReadTypes()**, **NXReadPoint()**, and **NXReadArray()**, as well as **NXReadObject()**. See the descriptions of these functions in this chapter for information about how to use them. A **read:** method can also check the version of the class being unarchived. See the description of **NXTypedStreamClassVersion()** for more information about how to do this.

After **NXReadObject()** unarchives an object, it sends the object **awake** and **finishUnarchiving** messages. You can implement an **awake** method to initialize the object to a usable state. The **finishUnarchiving** method allows you to replace the just-unarchived object with another one. If you implement a **finishUnarchiving** method, it should free the unarchived object and return the replacement object.

NXWriteObject() writes *object* to *stream* by sending the object a **write:** message. As is the case with **read:** methods, **write:** methods shouldn't be sent directly to objects, and they need to be implemented for classes that add instance variables. They also need to begin with a message to **super**. The functions available for archiving instance variables parallel those for unarchiving; they include **NXWriteTypes()**, **NXWritePoint()**, and **NXWriteArray()**, all of which are described elsewhere in this chapter. If the object being archived has **id** instance variables (including those that are statically typed to a class), they're archived as described below.

In some cases, an object's **id** instance variables contain inherent properties of the object to which they belong, or they might be necessary for the object to be usable. For example, a View's subview list is an intrinsic part of that View, just as a ButtonCell is needed for a Button to work properly. For these kinds of instance variables, the object—the View or the

Button in the examples mentioned—uses **NXWriteObject()** within its **write:** method. (Actually, Button objects inherit Control's **write:** method, which archives the **cell** instance variable.) The function **NXWriteTypes()** can also be used to archive **id** instance variables, by specifying the **id** data type format character.

In other cases, an object's **id** instance variables refer to other objects that act at the discretion of the object, such as its target or delegate, or that aren't inherently part of the object. A View's **superview** and **window** instance variables aren't considered intrinsic to the View since you might want to hook up the View to another superview or to a different Window. For these kinds of instance variables, the object calls **NXWriteObjectReference()** within its **write:** method. When archiving a data structure that includes objects that have called **NXWriteObjectReference()**, **NXWriteRootObject()** must be used instead of **NXWriteObject()**.

NXWriteObjectReference() specifies that a pointer to **nil** should be written for the object passed in, unless that object is an intrinsic part of some member of the data structure being archived. If the object is intrinsic, it will be archived and, after unarchiving, the pointer will point to the object. **NXWriteRootObject()** makes two passes through the data structure being written. The first time, it defines the limits of the data to be written by including instance variables intrinsic to the data structure and by making a note of which objects have been written with **NXWriteObjectReference()**. On the second pass, **NXWriteRootObject()** archives the data structure.

As an example, consider a View that has a Button as one subview and a TextField, which is the target of the Button, as another subview. If you archive the Button, its ButtonCell will be written. The archived ButtonCell's **target** instance variable will point to **nil**. If you archive the View, however, the Button and the TextField will be archived since they're subviews. The ButtonCell will be archived since it's needed by the Button. The ButtonCell's **target** instance variable will point to the TextField since it's an intrinsic part of the View.

RETURN **NXReadObject()** returns the **id** of the object read.

EXCEPTIONS All functions check whether the typed stream has been opened for reading or for writing and raise a TYPEDSTREAM_CALLER_ERROR exception with an appropriate message if it isn't correct. For example, if **NXReadObject()** is called and the stream was opened for writing, an exception is raised.

If an error occurs while creating an instance of the appropriate class, **NXReadObject()** raises a TYPEDSTREAM_CLASS_ERROR. This function also raises a TYPEDSTREAM_FILE_INCONSISTENCY exception if the data to be read is not of type **id**.

If **NXWriteObject()** is used to archive a data structure that includes objects with calls to **NXWriteObjectReference()**, a TYPEDSTREAM_WRITE_REFERENCE_ERROR exception is raised.

SEE ALSO **NXOpenTypedStream()**, **NXReadArray()**, **NXReadType()**, **NXReadPoint()** (Application Kit), and **NXTypedStreamClassVersion()**

NXReadObjectFromBuffer(), NXReadObjectFromBufferWithZone(), NXWriteRootObjectToBuffer(), NXFreeObjectBuffer()

SUMMARY Read and write an object to a typed-stream memory buffer

DECLARED IN objc/typedstream.h

SYNOPSIS
id **NXReadObjectFromBuffer(**const char ***buffer*, int *length***)**
id **NXReadObjectFromBufferWithZone(**const char ***buffer*, int *length*, NXZone ***zone***)**
char ***NXWriteRootObjectToBuffer(**id *object*, int ***length***)**
void **NXFreeObjectBuffer(**char ***buffer*, int *length***)**

DESCRIPTION These functions allow you to easily read and write an object to a typed stream on memory. They're particularly useful for archiving an object, writing it to the pasteboard, and then unarchiving it from the pasteboard.

NXWriteRootObjectToBuffer() opens a stream on memory (using **NXOpenMemory()**) and a corresponding typed stream. It then writes the object given as its argument by calling **NXWriteRootObject()** and closes the typed stream. (See the description of **NXWriteRootObject()** under **NXReadObject()** above for more information about how the object is written.) **NXWriteRootObjectToBuffer()** also closes the memory stream but retains the buffer, which is truncated to the size of the object.
NXWriteRootObjectToBuffer() returns the size of the object (in the location specified by *length*) and a pointer to the buffer itself.

NXReadObjectFromBuffer() calls **NXReadObjectFromBufferWithZone()** with the default zone as its *zone* argument.

NXReadObjectFromBufferWithZone() opens a stream on memory and a corresponding typed stream with its zone set by the **NXSetTypedStreamZone()** function. The *buffer* and *length* arguments passed in should be taken from a previous call to **NXWriteRootObjectToBuffer()**. **NXReadObject()** is called to read the object from the

buffer into the zone, after which the streams are closed.
NXReadObjectFromBufferWithZone() saves the memory buffer and returns the object it reads in the zone specified. Unless you're going to reread the buffer, you should free it using the **NXFreeObjectBuffer()** function.

NXFreeObjectBuffer() frees the buffer specified by *buffer*, which should be *length* bytes long. These arguments should be taken from a previous call to **NXWriteRootObjectToBuffer()**.

RETURN **NXReadObjectFromBuffer()** returns the object it reads from the buffer.

NXWriteRootObjectToBuffer() returns a pointer to the buffer it creates.

EXCEPTIONS **NXReadObjectFromBuffer()** and **NXReadObjectFromBufferWithZone()** raise a TYPEDSTREAM _FILE_INCONSISTENCY exception if the data to be read from the buffer is not of type **id**.

SEE ALSO **NXOpenMemory()**, **NXReadObject()**, and **NXOpenTypedStream()**

NXReadObjectFromBufferWithZone() → See NXReadObjectFromBuffer()

NXReadType(), NXWriteType(), NXReadTypes(), NXWriteTypes()

SUMMARY Read or write arbitrary data to a typed stream

DECLARED IN objc/typedstream.h

SYNOPSIS
void **NXReadType(**NXTypedStream ***stream*, const char ***type*, void ***data***)**
void **NXWriteType(**NXTypedStream ***stream*, const char ***type*, const void ***data***)**
void **NXReadTypes(**NXTypedStream ***stream*, const char ***types*, ...**)**
void **NXWriteTypes(**NXTypedStream ***stream*, const char ***types*, ...**)**

DESCRIPTION These functions read and write strings of data from and to a typed stream. They can be used within **read:** or **write:** methods for archiving purposes. See the description of **NXReadObject()** in this chapter for more about these methods. Functions are also

available for reading and writing certain data types; they're listed below in the "See Also" section.

These functions are similar to the **NXPrintf()** and **NXScanf()** functions for streams (and to the **printf()** and **scanf()** standard C functions). Before using a typed stream for reading and writing, it must be opened; see the description of **NXOpenTypedStream()** for details on opening a typed stream. (The NXTypedStream type is declared in the header file **objc/typedstream.h**. The structure itself is private since you never need to access its members.)

These four functions take as arguments a pointer to a typed stream, a character string indicating the format of the data to be read or written, and the address of the data. The data types and format string characters listed below are supported.

Type	Code	TypeCode
int	i	**char**c
unsigned int	I	**unsigned char**C
short	s	**char** **
unsigned short	S	NXAtom%
long	l	**id**@
unsigned long	L	Class#
float	f	SEL:
double	d	structure{<*types*>}
ignored	!	array[<*count*><*types*>]

For example, "[15d]" means that each stored element is an array of fifteen **doubles**, and "{csi*@}" means that each stored element is a structure containing a **char**, a **short**, an **int**, a character pointer, and an object.

Most of these codes are identical to ones that would be returned by the **@encode()** compiler directive. However, there are some differences:

- A structure description can contain only encoded type information between the braces. It can't include a full type name or structure name.
- The '%' descriptor specifies a unique string pointer. When the pointer is unarchived, the **NXUniqueString()** function is called to make sure that it's also unique within the new context.
- The '!' descriptor marks data that won't be archived. Each occurrence of '!' instructs the archiver to skip data the size of an **int**.
- A few **@encode()** descriptors—such as the ones for pointers, bitfields, and undefined types—should not be used. Use only the codes shown in the table above.

NXReadType() and **NXWriteType()** read and write the data specified by *data* as the single data type specified by *type*. The functions **NXReadTypes()** and **NXWriteTypes()** read and write multiple types of data; the types should be listed in *types* using the appropriate format characters shown above, and matching data should be provided in *data*. This example shows three different data types being written to an already open typed stream:

```
float   aa = 3.0;
int     bb = 5;
char   *cc = "foo";

NXWriteTypes(typedStream, "fi*", &aa, &bb, &cc);
```

If **NXWriteType()** had been used, three lines of code would have been necessary, one for each data type. Both functions take pointers to the data to be written, unlike **printf()**.

To read these three pieces of data from the NXTypedStream, **NXReadTypes()** would be called with the same arguments as shown above for **NXWriteTypes()**:

```
NXReadTypes(typedStream, "fi*", &aa, &bb, &cc);
```

Note: **NXWriteType()**/**NXReadType()** and **NXWriteTypes()**/**NXReadTypes()** must be used symmetrically. That is, if you write data values using a series of **NXWriteType()** function calls, you must read those values using a corresponsing series of **NXReadType()** function calls. A similar stricture applies for **NXWriteTypes()** and **NXReadTypes()**.

Note: Use **NXWriteType()** and **NXReadType()** to archive structures; for example, use the following code to write a structure of four floats:

```
NXWriteType(s, "{4f}", &data)
```

then use the corresponding code to read the structure:

```
NXReadType(s, "{4f}", &data)
```

Use the **NXWriteArray()** and **NXReadArray()** functions to write and read arrays. Avoid using the **NXWriteTypes()** and **NXReadTypes()** functions for structures and arrays; these functions can archive arrays and structures incorrectly and cause errors at runtime.

EXCEPTIONS

All four functions check whether the typed stream has been opened for reading or for writing and raise a TYPEDSTREAM_FILE_INCONSISTENCY exception if the type isn't correct. For example, if **NXReadType()** or **NXReadTypes()** is called and the stream was opened for writing, the exception is raised.

The functions for reading raise a TYPEDSTREAM_FILE_INCONSISTENCY exception if the data to be read is not of the expected type.

SEE ALSO **NXOpenTypedStream()**, **NXReadObject()**, and **NXReadPoint()**

NXReadTypes() → See NXReadType()

NXReallyFree() → See NXCreateHashTable()

NXRegisterDefaults(), NXGetDefaultValue(), NXReadDefault(), NXSetDefault(), NXWriteDefault(), NXWriteDefaults(), NXUpdateDefault(), NXUpdateDefaults(), NXRemoveDefault(), NXSetDefaultsUser()

SUMMARY Set or read default values

DECLARED IN defaults/defaults.h

SYNOPSIS
int **NXRegisterDefaults(**const char *_owner_, const NXDefaultsVector _vector_**)**
const char ***NXGetDefaultValue(**const char *_owner_, const char *_name_**)**
const char ***NXReadDefault(**const char *_owner_, const char *_name_**)**
int **NXSetDefault(**const char *_owner_, const char *_name_, const char *_value_**)**
int **NXWriteDefault(**const char *_owner_, const char *_name_, const char *_value_**)**
int **NXWriteDefaults(**const char *_owner_, NXDefaultsVector _vector_**)**
const char ***NXUpdateDefault(**const char *_owner_, const char *_name_**)**
void **NXUpdateDefaults(**void**)**
int **NXRemoveDefault(**const char *_owner_, const char *_name_**)**
const char ***NXSetDefaultsUser(**const char *_newUser_**)**

DESCRIPTION These functions give you access to a system of *default parameters* through which you can allow users to customize your application to meet their needs. For example, you can allow users to determine what units of measurement your application will display or how often documents will be automatically saved. The parameters get the name *default* since they're commonly used to determine an application's default state at startup or the way it will act by default.

Parameter values can be set from the command line or from a user's *defaults database*. Since values are stored specific to a particular user, you can use the parameters to record user preferences or to capture the application's state in one session so that it can be carried over to the next session. You can invent whatever parameters your application needs. Some parameters are defined in NEXTSTEP software—for example, many record choices the user makes in the Preferences application. See Appendix B, "Default Parameters," for a listing of system-defined parameters that you can read or set.

Parameters are set on the command line by preceding the parameter name with a hyphen and following it with a value. For example, the following instruction would launch the Edit application on the host machine named "earth" and assign that name as the value of the NXHost parameter:

```
localhost> /NextApps/Edit.app/Edit -NXHost earth
```

Listing a parameter on the command line doesn't put it in the defaults database. To put a parameter in the defaults database, you must use the functions described below.

A defaults database is created automatically for each user. It's named **.NeXTdefaults** and is located in the **.NeXT** directory in the user's home directory. Each parameter in the database is made up of three components:

- An owner, which is either the name of a specific application or "GLOBAL"
- The name of the parameter
- The value associated with the parameter

Each component is specified as a character string.

At run time, the parameters your application will use are placed in an internal cache. By using this cache, you avoid having to open the user's defaults database each time that you need access to a parameter. The cache is a list of parameters containing the same three components for each parameter as the database: the owner, the parameter name, and the associated value.

To register parameters in the cache, call **NXRegisterDefaults()** and give it two arguments: a character string specifying the owner and an NXDefaultsVector array. This array is a list of structures, each containing a parameter name and a value. (NXDefaultsVector is defined in the header file **defaults/defaults.h**.) Every application should register default parameters early in the program, before any of the values are needed.

Note: You should use the full market name of your product as the owner of the parameters you create. This will avoid conflicts with existing parameters. Noncommercial applications might use the name of the program and the author or institution.

A good place to call **NXRegisterDefaults()** is in the **initialize** method of the class that will use the parameters. The following example registers the values in **ArbDefaults** for the owner "Arboretum" (note that NULL is used to signal the end of the NXDefaultsVector array):

```
+ initialize
{
    static NXDefaultsVector ArbDefaults = {
        {"NXMeasurementUnit", NULL},
        {"AutoPropagate", "YES"},
        {NULL}
    };

    NXRegisterDefaults("Arboretum", ArbDefaults);

    return self;
}
```

If the defaults database doesn't exist when **NXRegisterDefaults()** is called, it's automatically created and placed in the **.NeXT** directory; the directory is created if necessary.

NXRegisterDefaults() creates a cache that contains a value for each of the parameters listed in the NXDefaultsVector array. For each parameter, a value is determined by first looking to see if it was defined on the command line (if the application was launched that way); if not, the user's defaults database (**.NeXTdefaults**) is searched. If **NXRegisterDefaults()** finds a parameter and owner in the database that match those passed to it as arguments, the corresponding value from the database is placed in the cache. If no parameter-owner match is found, **NXRegisterDefaults()** searches the database's global parameters—those owned by "GLOBAL"—for a matching parameter, and, if it finds one, places the corresponding value in the cache. If a match still isn't found, the parameter-value pair listed in the NXDefaultsVector array is used. (A value can be specified in the array as NULL.)

To summarize, this is the precedence ordering used to obtain a value for a given parameter for the cache:

1. The command line
2. The user's defaults database (**.NeXTdefaults**), with a matching owner
3. The user's defaults database, with the owner listed as "GLOBAL"
4. The NXDefaultsVector array passed to **NXRegisterDefaults()**

To read a parameter value, you'll most often call **NXGetDefaultValue()**. This function takes an owner and name of a parameter as arguments and returns a character pointer to the value for that parameter. **NXGetDefaultValue()** first looks in the cache for a matching

owner-parameter item. If **NXGetDefaultValue()** doesn't find a match in the cache (which would be the case only if the parameter wasn't in the NXDefaultsVector array passed to **NXRegisterDefaults()**), it searches the user's defaults database (**.NeXTdefaults**) for the owner and parameter. If still no match is found, it searches for a matching global parameter, first in the cache and then in the database. If the value is found in the database rather than the cache, **NXGetDefaultValue()** registers that value for subsequent use.

Occasionally, you may want to search only the database for a parameter value and ignore the cache. For example, you might want to get a parameter value that another application may have changed after the cache was created. In these rare cases, call **NXReadDefault()**, which takes an owner and parameter name as arguments and looks in the database for an exact match. It doesn't look for a global parameter unless "GLOBAL" is specified as the owner. If a match is found, a character pointer to the value is returned; if no value is found, NULL is returned. After obtaining a value from the database with **NXReadDefault()**, you may want to write it into the cache with **NXSetDefault()**.

NXSetDefault() takes as arguments an owner, the name of a parameter, and a value for that parameter. The parameter and its value are placed in the cache, but they aren't written into the user's defaults database (**.NeXTdefaults**).

NXWriteDefault() writes the owner, parameter, and value specified as its arguments into the user's defaults database and places them in the cache. Similarly, **NXWriteDefaults()** writes a vector of parameters into the database and registers it. Both **NXWriteDefault()** and **NXWriteDefaults()** return the number of successfully written values. To maximize efficiency, you should use one call to **NXWriteDefaults()** rather than several calls to **NXWriteDefault()** to write multiple values. This will save the time required to open and close the database each time a value is written.

Since other applications (and the user) can write to the database, at various points the database and the internal cache might not agree on the value of a given parameter. You can update the cache with any changes that have been made to the database since the cache was created by calling **NXUpdateDefault()** or **NXUpdateDefaults()**. Both functions compare the cache and the database. If a value is found in the database that is newer than the corresponding value in the internal cache, the new value is written into the cache.

NXUpdateDefault() updates the value for the single parameter and owner given as its arguments. **NXUpdateDefaults()**, which takes no arguments, updates the entire cache. It checks every parameter in the cache, determines whether a newer value exists in the database, and puts any newer values it finds in the cache.

NXRemoveDefault() removes the specified owner-parameter pair from both the user database and the internal cache.

Ordinarily, the functions described above use the database belonging to the user who started the application. **NXSetDefaultsUser()** changes which defaults database is used by subsequent calls to these functions. **NXSetDefaultsUser()** accepts the name of a user whose database you wish to use; it returns a pointer to the name of the user whose defaults database was previously set for access by these functions. All entries in the internal cache are purged; use **NXGetDefaultValue()** or **NXRegisterDefaults()** to get the new user's defaults for your application. When **NXSetDefaultsUser()** is called, the user who started the application must have appropriate access (read, write, or both) to the defaults database of the new user. This function is generally called in applications intended for use by a superuser who needs to update defaults databases for a number of users.

RETURN **NXRegisterDefaults()** returns 0 if the database couldn't be opened; otherwise it returns 1.

NXGetDefaultValue() returns a character pointer to the requested parameter value, or 0 if the database couldn't be opened.

NXReadDefault() returns a character pointer to the parameter value; if a value is not found, NULL is returned.

NXSetDefault() returns 1 if it successfully set a parameter value, and 0 if not.

NXWriteDefault() returns 1 unless an error occurs while writing the parameter value, in which case it returns 0.

NXWriteDefaults() returns the number of successfully written parameter values.

NXUpdateDefault() returns the new value, or NULL if the value did not need to be updated.

NXRemoveDefault() returns 1, or 0 if the parameter couldn't be removed.

NXSetDefaultsUser() returns the login name of the user whose defaults database was being used before the function was called.

NXRegisterPrintfProc()

SUMMARY Register a procedure for formatting data written to a stream

DECLARED IN streams/streams.h

SYNOPSIS void **NXRegisterPrintfProc(**char *formatChar*, NXPrintfProc **proc*, void **procData***)*

DESCRIPTION **NXRegisterPrintfProc** registers *formatChar*, a format character that corresponds to **proc*, which is a pointer to a function of type NXPrintfProc. The type definition for an NXPrintfProc function is:

```
typedef   void   NXPrintfProc(NXStream *stream, void *item,
                                  void *procData)
```

formatChar can be any of the characters “vVwWyYzZ”; other characters are reserved for use by NeXT. *procData* represents client data that will be blindly passed along to the function.

After calling **NXRegisterPrintfProc()**, *formatChar* can be used in a format string for the **NXPrintf()** or **NXVPrintf()** functions. When these functions encounter *formatChar* in a format string, *proc* will be called to format the corresponding argument passed to **NXPrintf()**. For example:

```
tabOver(NXStream stream, void *item, void *data)
{
    . . .
}

NXRegisterPrintfProc('v', &tabOver, NULL);
    . . .
NXPrintf(myStream, "%v", itemOne);
```

This code registers “v” as the formatting character for **tabOver()**; with the NULL argument, no client data will be passed to **tabOver()**. **NXPrintf()** then passes the variable **itemOne** to **tabOver()** for formatting, which formats the item and places it in **myStream**.

SEE ALSO NXPutc()

NXRemoveDefault() → See NXRegisterDefaults()

NXResetErrorData() → See NXAllocErrorData()

NXResetHashTable() → See NXCreateHashTable()

NXSaveToFile() → See NXOpenMemory()

NXScanf() → See NXPutc()

NXSeek(), NXTell(), NXAtEOS()

SUMMARY Set or report current position in a stream

DECLARED IN streams/streams.h

SYNOPSIS
void **NXSeek(**NXStream *_stream_, long _offset_, int _ptrName_**)**
long **NXTell(**NXStream *_stream_**)**
BOOL **NXAtEOS(**NXStream *_stream_**)**

DESCRIPTION These functions set or report the current position in the stream given as an argument. This position determines which data will be read next or where the next data will be written since the functions for reading and writing to a stream start from the current position.

NXSeek() sets the position *offset* number of bytes from the place indicated by *ptrName*, which can be NX_FROMSTART, NX_FROMCURRENT, or NX_FROMEND.

NXTell() returns the current position of the buffer. This information can then be used in a call to **NXSeek()**.

The macro **NXAtEOS()** evaluates to TRUE if the end of a stream has been reached. Since streams opened for writing don't have an end, this macro should only be used with streams opened for reading.

Since position within a Mach port stream is undefined, **NXSeek()** and **NXTell()** shouldn't be called on a Mach port stream. These functions also shouldn't be used on a typed stream. The NX_CANSEEK flag (defined in the header file **streams/streams.h**) can be used to determine if a given stream is seekable.

RETURN **NXTell()** returns the current position of the buffer.

NXAtEOS() evaluates to TRUE if the end of the stream has been detected and to FALSE otherwise.

EXCEPTIONS **NXSeek()** and **NXTell()** raise an NX_illegalStream exception if the stream passed in is invalid.

NXSeek() raises an NX_illegalSeek exception if *offset* is less than 0 or greater than the length of a reading stream. This exception will also be raised if *ptrName* is anything other than the three constants listed above.

NXSetDefault() → See NXRegisterDefaults()

NXSetDefaultsUser() → See NXRegisterDefaults()

NXSetExceptionRaiser() → See NXDefaultExceptionRaiser()

NXSetTypedStreamZone() → See NXGetTypedStreamZone()

NXSetUncaughtExceptionHandler(), NXGetUncaughtExceptionHandler()

SUMMARY Handle uncaught exceptions

DECLARED IN objc/error.h

SYNOPSIS void **NXSetUncaughtExceptionHandler(**NXUncaughtExceptionHandler **proc***)**
NXUncaughtExceptionHandler ***NXGetUncaughtExceptionHandler(**void**)**

DESCRIPTION These macros provides a means of handling exceptions that are raised outside of an NX_DURING...NX_ENDHANDLER construct. You can use the Application object's default procedure, or you can define your own handler using **NXSetUncaughtExceptionHandler()**.

If *proc* is NULL or if you never call **NXSetUncaughtExceptionHandler()**, your program will use the Application object's default procedure. This function writes an uncaught exception message to **stderr** if the application was launched from a terminal. If the application was launched by the Workspace Manager, the message is written using **syslog()** with the priority set to LOG_ERR; this message will normally appear in the Workspace Manager's console window. The default uncaught exception handler then calls the function pointed to by **NXTopLevelErrorHandler()** and passes it any data about the exception supplied by **NX_RAISE()**, which was called when the exception occurred. (See the description of **NX_RAISE()**.) If you haven't defined your own top-level error handler, the program exits.

To create your own handler, you define an exception handling function and give the name of that function as an argument to **NXSetUncaughtExceptionHandler()**. Subsequent calls

to **NXGetUncaughtExceptionHandler()** will return a pointer to the function. These two macros are defined in the header file **objc/error.h**.

SEE ALSO **NX_RAISE()**, **NXDefaultTopLevelErrorHandler()**

NXStreamCreateFromZone(), NXStreamCreate(), NXStreamDestroy(), NXDefaultRead(), NXDefaultWrite(), NXFill(), NXChangeBuffer()

SUMMARY Support a user-defined stream

DECLARED IN streams/streamsimpl.h

SYNOPSIS
NXStream ***NXStreamCreateFromZone(**int *mode*, int *createBuf*, NXZone **zone***)**
NXStream ***NXStreamCreate(**int *mode*, int *createBuf***)**
void **NXStreamDestroy(**NXStream **stream***)**
int **NXDefaultRead(**NXStream **stream*, void **buf*, int *count***)**
int **NXDefaultWrite(**NXStream **stream*, const void **buf*, int *count***)**
int **NXFill(**NXStream **stream***)**
void **NXChangeBuffer(**NXStream **stream***)**

DESCRIPTION These functions need only be used if you implement your own version of a stream. If you're using a memory stream, a stream on a file, a stream on a Mach port, or a typed stream, you don't need the functions described here. Instead, you can just use the functions already defined for these types of streams; see the *NEXTSTEP Programming Interface Summaries* manual for a list of these functions.

The first argument to **NXStreamCreateFromZone()**, *mode*, indicates whether the stream to be created will be used for reading or writing or both. It should be one of the following constants: NX_READONLY, NX_WRITEONLY, or NX_READWRITE. The argument *createBuf* specifies whether the stream should be buffered. If it is TRUE, a buffer is created of size NX_DEFAULTBUFSIZE, as defined in the header file **streams/streamsimpl.h**. The argument *zone* specifies the memory zone where you allocate memory for the new stream; see **NXCreateZone()** for more on allocating zones of memory. When implementing your own version of a stream, you may want to provide a function to open such a stream; this function will probably call **NXStreamCreateFromZone()**, as **NXOpenMemory()**, **NXOpenPort()**, and **NXOpenFile()** do.

NXStreamCreate() calls **NXStreamCreateFromZone()** with the default zone as its *zone* argument.

NXStreamDestroy() destroys the stream given as its argument, deallocating the space it had used. If a buffer had been created for *stream*, its storage is also freed. To avoid losing data, a stream should be flushed using **NXFlush()** before it's destroyed. When implementing your own version of a stream, you may want to provide a function to close such a stream; this function will probably call **NXStreamDestroy()**, as **NXClose()** and **NXCloseMemory()** do.

NXDefaultRead() and **NXDefaultWrite()** read and write multiple bytes of data on a stream. **NXDefaultRead()** reads the next *count* number of bytes from *stream*, starting at the position specified by the buffer pointer *buf*. **NXDefaultWrite()** writes *count* number of bytes to *stream*, starting at the position specified by *buf*. These functions return the number of bytes read or written. When implementing your own version of a stream, you can use these functions with your stream unless you want to perform specialized buffer management. If you implement your own versions of these functions for reading and writing bytes, they should return the number of bytes read or written.

When reading from a buffered stream, **NXFill()** can be called to fill the buffer with the next data to be read. Check whether **buf_left** is equal to 0 to determine whether all the data currently in the buffer has been read. (See the header file **streams/streams.h** for more information about **buf_left**, which is part of an NXStream structure.)

NXChangeBuffer() switches the mode of a stream between reading and writing. If the argument *stream* had been defined for reading, this function changes it to a stream that can be written to; if *stream* had been defined for writing, it becomes a stream for reading. In both cases, the pointer that points to either the next piece of data to be read from the buffer or the next location to which data will be written is realigned appropriately. Also, NX_READFLAG and NX_WRITEFLAG are updated to reflect the new mode of the stream.

RETURN

NXStreamCreate() returns a pointer to the stream it creates.

NXDefaultRead() and **NXDefaultWrite()** return the number of bytes read or written.

NXFill() returns the number of characters read into the buffer.

EXCEPTIONS

All functions that take a stream as an argument raise an NX_illegalStream exception if the stream passed in is invalid.

NXFill() raises an NX_illegalRead exception if an error occurs while filling.

NXChangeBuffer() raises an NX_illegalStream exception if NX_READFLAG and NX_WRITEFLAG have not been set to match the NX_CANREAD and NX_CANWRITE flags.

SEE ALSO **NXOpenFile()**, **NXOpenMemory()**, **NXClose()**, **NXFlush()**, **NXRead()**

NXStreamDestroy() → See NXStreamCreateFromZone()

NXStrHash() → See NXCreateHashTable()

NXStrIsEqual() → See NXCreateHashTable()

NXTell() → See NXSeek()

NXToAscii(), NXToLower(), NXToUpper()

SUMMARY Convert NEXTSTEP-encoded characters

DECLARED IN NXCType.h

SYNOPSIS
unsigned char ***NXToAscii(**unsigned int *c***)**
int **NXToLower(**unsigned int *c***)**
int **NXToUpper(**unsigned int *c***)**

DESCRIPTION These functions convert characters encoded in the extended character set defined by NEXTSTEP encoding. They are similar to the standard C library functions **toascii()**, **tolower()**, and **toupper()** (see the **ctype**(3) UNIX manual page), which operate on characters in the ASCII character set.

NXToLower() converts an uppercase letter to its lowercase equivalent, and **NXToUpper()** converts a lowercase letter to its uppercase equivalent. If there's no opposite case equivalent—or if the character is already of the desired case—these functions return the supplied argument unchanged.

NXToAscii() converts its argument to a value that lies within the standard ASCII character set. The lower 128 positions in NEXTSTEP encoding constitute the ASCII character set,

so no conversion is required for codes in this range. For the upper 128 character codes—the extended characters—**NXToAscii()** makes these conversions:

Extended Character	Converts to
Agrave, Aacute, Acircumflex, Atilde, Adieresis, Aring	A
Ccedilla	C
Egrave, Eacute, Ecircumflex, Edieresis	E
Igrave, Iacute, Icircumflex, Idieresis	I
Ntilde	N
Ograve, Oacute, Ocircumflex, Otilde, Odieresis, Oslash	O
Ugrave, Uacute, Ucircumflex, Udieresis	U
Yacute	Y
eth, Eth	TH
Thorn, thorn	th
fi	fi
fl	fl
agrave, aacute, acircumflex, atilde, adieresis, aring	a
ccedilla	c
egrave, eacute, ecircumflex, edieresis	e
AE	AE
igrave, iacute, icircumflex, idieresis	i
ntilde	n
Lslash	L
OE	OE
ograve, oacute, ocircumflex, otilde, odieresis, oslash	o
ae	ae
ugrave, uacute, ucircumflex, udieresis	u
dotlessi	i
yacute, ydieresis	y
lslash	l
oe	oe
germandbls	ss
multiply	x
divide	/
exclamdown	!
quotesingle	'
quotedblleft, guillemotleft, quotedblright, guillemotright, quotedblbase	\
quotesinglbase	'
guilsinglleft	<
guilsinglright	>
periodcentered	.

Extended Character	Converts to
brokenbar	|
bullet	*
ellipsis	...
questiondown	?
onesuperior	1
twosuperior	2
threesuperior	3
emdash	-
plusminus	+-
onequarter	1/4
onehalf	1/2
threequarters	3/4
ordfeminine	a
ordmasculine	o
mu, copyright, cent, sterling, fraction, yen, florin, section, currency, registered, endash, dagger, daggerdbl, paragraph, perthousand, logicalnot, grave, acute, circumflex, tilde, macron, breve, dotaccent, dieresis, ring, cedilla, hungarumlaut, ogonek, caron	_

RETURN **NXToAscii()** returns by reference a valid ASCII character. **NXToLower()** or **NXToUpper()** returns an integer value that represents the converted character.

SEE ALSO NXIsAlpha()

NXToLower() → See NXToAscii()

NXToUpper() → See NXToAscii()

NXTypedStreamClassVersion()

SUMMARY Get the class version number of an archived instance

DECLARED IN objc/typedstream.h

SYNOPSIS int **NXTypedStreamClassVersion(**NXTypedStream *_stream_, const char *_className_**)**

DESCRIPTION This function returns the class version number of an archived object. Class versioning is useful if you create a class, archive an instance of it, then change the class—by adding instance variables to it, for example. This function is used in a class's **read:** method to select the appropriate code for initializing the instance being unarchived. This function should be called only on a typed stream opened for reading with **NXReadObject()**.

NXTypedStreamClassVersion() can be called in your **read:** method after sending a **[super read:***stream***]** message and before performing version-specific initialization. Calling this function doesn't change the position of the read pointer in *stream*. If you need to know the version of an object's superclass (or any class in its inheritence hierarchy), call this function using the name of that class as *className*.

For **NXTypedStreamClassVersion()** to return a nonzero value, you should change the class version to a new value whenever you change the class definition. The Object class provides two methods for handling class versioning. Object's **setVersion:** class method can be used in a subclass's **initialize** class method to set a new class version when you change the instance variables. Object's **version** class method returns the current version of your class.

The **NXWriteObject()** function automatically archives the class version when it is archiving an object. The default version number is 0. Thus if you have previously archived instances of a class without setting the version, you can set the version of the altered class to any integer value other than 0, then use this function to detect old and new instances of the class.

In the following code example, MyClass's **initialize** method sets the class version using Object's **setVersion:** method:

```
@implementation MyClass:MySuperClass
+ initialize
{
    if (self == [MyClass class]) {
          [MyClass setVersion:MYCLASS_CURRENT_VERSION];
    }
    return self;
}
```

Note that this code tests to see that initialize is being invoked by the implementing class, not a subclass. This is useful to assure that subclasses don't inherit the version number (or other class-specific details).

In the next example, MyClass's **read:** method uses version numbers to unarchive old and new instances differently:

```
- read:(NXTypedStream *)typedStream
{
    [super read:typedStream];
    if (NXTypedStreamClassVersion(typedStream, "MyClass") ==
        [MyClass version] {
        /* read code for current version */
            . . .
    }
    else {
        /* read code for old version */
        . . .
    }
}
```

See the description of **NXReadObject()** earlier in this chapter for more information about archiving. The NXTypedStream type is declared in the header file **objc/typedstream.h**. The structure itself is private since you never need access to its members.

SEE ALSO NXReadObject()

NXUngetc() → See NXPutc()

NXUniqueString(), NXUniqueStringWithLength(), NXUniqueStringNoCopy(), NXCopyStringBuffer(), NXCopyStringBufferFromZone()

SUMMARY Manipulate a string buffer

DECLARED IN objc/hashtable.h

SYNOPSIS NXAtom **NXUniqueString(**const char *_buffer_**)**
NXAtom **NXUniqueStringWithLength(**const char *_buffer_, int _length_**)**
NXAtom **NXUniqueStringNoCopy(**const char *_buffer_**)**
char ***NXCopyStringBuffer(**const char *_buffer_**)**
char ***NXCopyStringBufferFromZone(**const char *_buffer_, NXZone *_zone_**)**

DESCRIPTION The first three functions in this group create unique strings, which are allocated once and then can be shared. The fourth and fifth functions allocate memory for and return a copy of the given string.

Unique strings are identified by the type NXAtom, which indicates that they can be compared using == rather than **strcmp()**. NXAtom strings shouldn't be deallocated or modified; the Mach function **vm_protect()** is used to ensure that the strings are read-only. (The type NXAtom is defined in the **objc/hashtable.h** header file.)

NXUniqueString(), **NXUniqueStringWithLength()**, and **NXUniqueStringNoCopy()** maintain a hash table of unique strings. Each function checks if the string passed in is already in the table and if so, returns it. Because a hash table is used, the average search time is constant regardless of how many unique strings exist. If *buffer* doesn't exist in the hash table, **NXUniqueString()** and **NXUniqueStringWithLength()** return a pointer to a copy of it as an NXAtom; **NXUniqueStringNoCopy()** inserts the string in the hash table but doesn't make a copy of it. For efficiency, all unique strings are stored in the same area of virtual memory.

NXUniqueString() assumes *buffer* is null-terminated; if it's NULL, **NXUniqueString()** returns NULL. **NXUniqueStringWithLength()** assumes that *buffer* is a non-NULL string of at least *length* non-NULL characters.

NXCopyStringBuffer() allocates memory from the default memory zone for a copy of *buffer*. Then *buffer*, which should be null-terminated, is copied using **strcpy()**. **NXCopyStringBufferFromZone()** is identical to **NXCopyStringBuffer()** except that memory is allocated from the specified zone.

RETURN **NXUniqueString()** and **NXUniqueStringWithLength()** return a pointer to a copy of *buffer* as an NXAtom.

NXUniqueStringNoCopy() returns a pointer to the string passed in.

NXCopyStringBuffer() and **NXCopyStringBufferFromZone()** return a pointer to a copy of *buffer*.

NXUniqueStringNoCopy() → See NXUniqueString()

NXUniqueStringWithLength() → See NXUniqueString()

NXUpdateDefault() → See NXRegisterDefaults()

NXUpdateDefaults() → See NXRegisterDefaults()

NXVPrintf() → See NXPutc()

NXVScanf() → See NXPutc()

NXWrite() → See NXRead()

NXWriteArray() → See NXReadArray()

NXWriteDefault() → See NXRegisterDefaults()

NXWriteDefaults() → See NXRegisterDefaults()

NXWriteObject() → See NXReadObject()

NXWriteObjectReference() → See NXReadObject()

NXWriteRootObject() → See NXReadObject()

NXWriteRootObjectToBuffer() → See NXReadObjectFromBuffer()

NXWriteType() → See NXReadType()

NXWriteTypes() → See NXReadType()

NXZoneCalloc() → See NXZoneMalloc()

NXZoneFromPtr() → See NXCreateZone()

NXZoneFree() → See NXZoneMalloc()

NXZoneMalloc(), NXZoneCalloc(), NXZoneRealloc(), NXZoneFree()

SUMMARY Allocate and free memory within a zone

DECLARED IN objc/zone.h

SYNOPSIS
void ***NXZoneMalloc(**NXZone **zone*, size_t *size***)**
void ***NXZoneCalloc(**NXZone **zone*, size_t *numElems*, size_t *numBytes***)**
void ***NXZoneRealloc(**NXZone **zone*, void **ptr*, size_t *size***)**
void **NXZoneFree(**NXZone **zone*, void **ptr***)**

DESCRIPTION These functions allocate and free memory within a particular region, or *zone*. They're similar to the standard C library functions **malloc()**, **calloc()**, **realloc()**, and **free()**, but allow more control over memory placement. By placing data structures that are likely to be used in conjunction with each other in the same zone, you can ensure better locality of reference. This can significantly improve performance on a paged virtual memory system. When related data structures are grouped close together, consecutive references are less likely to result in memory paging activity.

To use these functions, you must first obtain a pointer to a zone, generally by creating a new zone using **NXCreateZone()**. The zone pointer is passed as the first argument to each of these functions. Memory is allocated from the zone specified.

NXZoneMalloc() allocates *size* bytes from *zone*, and returns a pointer to the allocated memory. **NXZoneCalloc()** allocates enough memory from *zone* for *numElems* elements, each with a size of *numBytes* bytes, and returns a pointer to the allocated memory. Both allocate memory that's aligned to accommodate any C data type. Like **calloc()**, **NXZoneCalloc()** sets the allocated memory to 0 throughout; **NXZoneMalloc()**, like **malloc()**, does not.

NXZoneRealloc() changes the size of the block of memory pointed to by *ptr* to *size* bytes. It may allocate new memory to replace the old. If so, it moves the contents of the old memory block to the new block, up to a maximum of *size* bytes.

NXZoneFree() returns memory to the zone from which it was allocated. The standard C function **free()** does the same, but spends time finding which zone the memory belongs to.

For both **NXZoneRealloc()** and **NXZoneFree()**, *ptr* must be a pointer to a memory block that was returned by **NXZoneMalloc()**, **NXZoneCalloc()**, **NXZoneRealloc()**, or their standard C counterparts. The *zone* must be the one from which the *ptr* memory block was allocated; if it's not, the results are unpredictable, and possibly disastrous.

NXZoneMalloc(), **NXZoneRealloc()**, and **NXZoneFree()** are implemented as macros.

RETURN If successful, **NXZoneMalloc()**, **NXZoneCalloc()**, and **NXZoneRealloc()** return a pointer to the memory allocated (or reallocated). If unsuccessful, they return NULL.

SEE ALSO **NXCreateZone()**, – **allocFromZone:** (Object class)

NXZonePtrInfo() → See NXMallocCheck()

NXZoneRealloc() → See NXZoneMalloc()

NX_ADDRESS()

SUMMARY Get a pointer to the objects stored in a List

DECLARED IN objc/List.h

SYNOPSIS id ***NX_ADDRESS(**List **aList***)**

DESCRIPTION This macro takes a List object, *aList*, as its argument and returns a pointer to the first **id** stored in the List. With this pointer, you get direct access to the contents of the List and can avoid the overhead of messaging. **NX_ADDRESS()** therefore provides an alternative to List's **objectAt:** method for situations where somewhat greater performance is required. In general, however, the method is the preferred way of accessing the List.

RETURN This macro returns a pointer to the contents of a List object.

SEE ALSO List class

NX_ENDHANDLER → See NX_DURING

NX_DURING, NX_HANDLER, NX_ENDHANDLER

SUMMARY Mark exception handling domains and handlers

DECLARED IN objc/error.h

SYNOPSIS
NX_DURING
NX_HANDLER
NX_ENDHANDLER

DESCRIPTION These macros are used to delimit portions of code that are under the control of the NEXTSTEP exception handling system. Code that lies between the NX_DURING and NX_HANDLER macros is said to lie in an exception-handling domain. Code that lies between NX_HANDLER and NX_ENDHANDLER is said to be within the exception

handler. A call to **NX_RAISE()** within the exception-handling domain transfers program execution to the first line of code in the exception handler. See **ExceptHandling.rtfd** in **/NextLibrary/Documentation/NextDev/Concepts** for more information.

SEE ALSO NX_RAISE()

NX_HANDLER → See NX_DURING

NX_RAISE(), NX_RERAISE(), NX_VALRETURN(), NX_VOIDRETURN

SUMMARY Raise an exception

DECLARED IN objc/error.h

SYNOPSIS void **NX_RAISE(**int *code*, const void **data1*, const void **data2***)**
NX_RERAISE(void**)**
NX_VALRETURN(*val***)**
NX_VOIDRETURN

DESCRIPTION These macros initiate the error handling mechanism by alerting the appropriate error handler that an error has occurred. Error handlers exist in a nested hierarchy, which is created by using any number of nested NX_DURING...NX_ENDHANDLER constructs and by defining a top-level error handler.

The three arguments for **NX_RAISE()** provide information about the error condition. The first argument is a constant that acts as a label for the error. (Error codes used by the Application Kit are defined in the header file **appkit/errors.h**.) The next two arguments point to arbitrary data about the error. Within an NX_DURING...NX_ENDHANDLER construct, this data is stored in a local variable called **NXLocalHandler** (which is of type NXHandler, defined in the header file **objc/error.h**). (See the description of **NXAllocErrorData()** for more information about managing the storage of error data.) **NX_RAISE()** calls the function pointed to by **NXGetExceptionRaiser()**; see this function's description earlier in this chapter.

By default, an error handler should call **NX_RERAISE()** when it encounters an error that it can't handle, as shown below. **NX_RERAISE()** has the same functionality as **NX_RAISE()**, but it's called with no arguments. Since **NX_RERAISE()** implies a

previous call to **NX_RAISE()**, the error data will already be stored in the local handler, eliminating the need for arguments.

```
NX_DURING
    /* code that may cause an error */
NX_HANDLER
    switch (NXLocalHandler.code)
    case
        NX_someErrorCode:
            /* code to execute for this type of error */
    default: NX_RERAISE();
NX_ENDHANDLER
```

NX_VALRETURN() and **NX_VOIDRETURN** can be used to exit a method or function from within the block of code between NX_DURING and NX_HANDLER labels. The only legal ways of exiting this block are falling out the bottom or using one of these macros. **NX_VALRETURN()** causes its method (or function) to return *val*, while **NX_VOIDRETURN** can be used to return from a method (or function) that has no return value. Use these macros only within an NX_DURING...NX_HANDLER construct.

SEE ALSO **NXAllocErrorData()**, **NXSetUncaughtExceptionHandler()**, **NXDefaultTopLevelErrorHandler()** (Application Kit), **NXRegisterErrorReporter()** (Application Kit), **NXDefaultExceptionRaiser()**

NX_RERAISE() → See NX_RAISE()

NX_VALRETURN() → See NX_RAISE()

NX_VOIDRETURN → See NX_RAISE()