10 *Support Objects and Functions*

This information, although fundamentally correct, has not been updated for release 3.0. For up-to-date information on:

Section	**New Documentation**
Defaults	/NextLibrary/Documentation/NextDev/GeneralRef/ApB_Defaults
Exception Handling	/NextLibrary/Documentation/NextDev/Concepts/ExceptionHandling.rtf

For up-to-date information on the various classes mentioned below (Font, Storage, and so on), see the appropriate section of the *NeXTSTEP General Reference* manual (available online in /NextLibrary/Documentation/NextDev/GeneralRef).

In addition to a program structure for applications that use the NeXT window system and a variety of preprogrammed user-interface objects, the Application Kit offers a number of other program support facilities.

Some are implemented as class definitions and some as standard C functions and macros. All are designed to work well with the Kit's program structure and user-interface objects. They include:

- A set of functions for writing and reading data to streams.
- A set of functions that allow you to save data structures, including objects, in an archive file and load them from the file into an application.
- A system for specifying program defaults.
- A Pasteboard object that supports cut, copy, and paste operations.
- A Font and a FontManager object that help applications get information about a specific font and serve as a vehicle for setting the font in the Window Server.
- List, Storage, and HashTable objects that act as general memory allocators. The StreamTable object is a specialized storage class.
- A mechanism for handling errors.

Other Application Kit objects depend on these support facilities. For example, the Text class uses a variety of Font objects and the Pasteboard for cut and paste operations. Program support facilities aren't confined to a behind-the-scenes role, however. You can make direct use of them in your program.

Streams

A stream is a sequence of data into or out of a program. It acts as a channel, connecting an application with a source of data or a destination for data. If you use a stream (rather than a file, for example) for the input or output of data, you can read or write data without regard to its source or destination. For example, suppose you

designed an object that writes data to a stream. You can save that data in a file, or send it to another process using a Mach port, without changing the object. You only need to call the function that connects a stream to a file or a Mach port and then pass the connected stream to the object.

The Application Kit writes instances of its classes to a special kind of stream, a typed stream. Typed streams are particularly useful for writing and reading objects and other complex data structures. See ªArchiving to a Typed Streamº later in this chapter for more information.

When using a stream, you can select memory, a file, or a Mach port as the source or destination by calling the appropriate function to create the stream. You can also implement the functions needed to connect a stream to a different source or destination, such as a Text object, thereby creating your own type of stream.

Memory and file streams allow two-way data flowÐthat is, you can use the same stream for both writing and reading. In addition, you can set the position of the next input or output operation on these streams. For example, you can read the first few bytes of data from a memory stream, skip to the middle to read some more, and then write data at the end of the stream.

The next section discusses how to write data to and read it from a stream. Then the steps needed to connect the stream to memory, a file, or a Mach port are presented.

Writing and Reading

The functions that write to or read from a stream can be grouped into three categories, depending on whether they:

- Write or read single characters at a time,
- Write or read a specified number of bytes of data, or
- Convert data according to a format string as it's read or written.

These functions are modeled after the standard C library functions for input and output. If you are familiar with those standard C functions, you know in a general way what the corresponding NeXT-defined functions do.

The NeXT functions for writing and reading take a pointer to a stream as an argument. These functions can be used with a stream connected to any source or destination. In the examples shown below, this stream has already been connected and is referred to as **stream**. See the section ªConnecting Streams to a Source or Destinationº below for details on connecting a stream.

Writing and Reading Characters

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:

```
NXPutc(stream, 'c');
```

The second argument specifies the character to be written to the stream. **NXGetc()** retrieves the next character from the stream:

```
unsigned char  aCharacter;
aCharacter = NXGetc(stream);
```

The **unsigned char** type should be used for portability.

To reread a character, call **NXUngetc()**. This function puts the last character read back onto the stream:

```
unsigned char  aCharacter;

NXUngetc(stream);
aCharacter = NXGetc(stream);
```

Note that **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).

Writing and Reading Bytes of Data

The functions **NXWrite()** and **NXRead()** write multiple bytes of data to and read them from a stream. 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. 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.

Writing and Reading Formatted Data

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 arguments 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 locations 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()**. The examples below illustrate the use of some of these conversion characters. For detailed information on these characters and how conversions are performed, see the UNIX manual pages for **printf()** and **scanf()**.

The following writes data of the form ªPlease send 500 bucks before Fridayº to a stream:

```
int    amt = 500;
char  *day = "Friday";

NXPrintf(stream, "Please send %d bucks before %s", amt, day);
```

The call

```
int    numint;
float  numflo;
char   name[15];

NXScanf(stream,"%d%f%s", &numint, &numflo, name);
```

with the stream of data

```
5  19.61  Jacqueline
```

will assign 5 to the variable **numint**, 19.61 to **numflo**, and ªJacquelineº to **name**.

Two related functions, **NXVPrintf ()** and **NXVScanf()**, are exactly the same as **NXPrintf()** and **NXScanf()**, respectively, 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**.

Flushing and Filling

File and Mach port streams are buffered, which means that data is initially written to a buffer rather than to the file or the port itself. If you write more data than the buffer can hold, the buffer is flushed, sending all the data to the destination that the stream is connected to. Also, before a stream is disconnected from its destination, the buffer is flushed to ensure that all data actually gets sent to the destination. Usually you won't need to flush the buffer yourself. However, if you don't want to disconnect the stream but your code depends on knowing that all data has been sent to the stream's destination, call **NXFlush()**:

```
NXFlush(stream);
```

If **NXFlush()** is called with a memory stream, more memory is made available for writing. However, you don't need to call this function with a memory stream since more memory is automatically allocated as needed.

When reading from a file or Mach port stream, data is loaded into a buffer and then read from the buffer. This buffer is automatically filled after you've read all the data in it. To explicitly fill the buffer yourself, call **NXFill()**:

```
NXFill(stream);
```

Calling this function with a memory stream has no effect.

Seeking

Stream functions for writing and reading start at the current position of the stream, so you may need to manipulate the position of the stream:

```
NXSeek(stream, 0, NX_FROMSTART);
```

NXSeek() moves forward by the number of bytes specified by its second argument relative to the position indicated by its last argument, which can be NX_FROMSTART, NX_FROMCURRENT, or NX_FROMEND. In the example, the current position is set to the beginning of the stream. The function **NXTell()** returns an **int** that specifies the current position in the stream given as its argument. This value, which is measured in bytes from the beginning of the stream, can be used in a call to **NXSeek()**.

Position within some streamsÐfor example, Mach port streamsÐis undefined, so these two functions shouldn't be used with such streams. They can be used with memory or file streams, but they should be avoided if the stream will ever be connected to an unseekable source or destination. The NX_CANSEEK flag, defined in the header file **streams/streams.h**, indicates whether a stream is seekable.

Connecting Streams to a Source or Destination

Functions are provided to open a stream on memory, a file, or a Mach port. Opening a stream involves connecting it to a source or destination and specifying whether it will be used for writing or reading (or both). Regardless of what the source and destination are, you can use the functions described above for writing data to and reading it from the stream. When you're finished with the stream, use the appropriate function to close it. The functions for closing a stream disconnect it from its source or destination and release storage used by the stream. (The NXStream structure used below is defined in the header file **streams/streams.h**.)

Connecting to Memory

A memory stream is a temporary buffer for writing or reading data. To open a memory stream, call **NXOpenMemory()**:

```
NXStream  *stream;
stream = NXOpenMemory(NULL, 0, NX_WRITEONLY);
```

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. 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.

When you're finished with a memory stream, close it by calling **NXCloseMemory()**:

```
NXCloseMemory(stream, NX_FREEBUFFER);
```

Usually, you'll use NX_FREEBUFFER as the second argument to free all storage used by the stream, but there are two other constants that can be used. 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.

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:

```
const char  *home = NXHomeDirectory();
NXSaveToFile(stream, home);
```

NXSaveToFile() writes the contents of the memory stream into the file, creating it if necessary. After saving the data, close the stream using **NXCloseMemory()**.

Connecting to a File

Two functions are available to connect a stream to a file. **NXMapFile()** maps a file into memory and then opens a memory stream; **NXOpenFile()** connects a stream to the file. Memory mapping allows efficient random and multiple access of the data in the file, so **NXMapFile()** should be used whenever the file is stored on disk. (The *NeXT Operating System Software* manual discusses memory mapping in more detail.) If you want to connect a stream to a pipe or a socket, use **NXOpenFile()**. This function takes a file descriptor as an argument.

To map a file into memory, call **NXMapFile()**, giving it the pathname for the file and indicating whether you'll be writing, reading, or both:

```
NXStream  *stream;
stream = NXMapFile("aPathname", NX_READONLY);
```

This function opens a memory stream and initializes it with the contents of the file. Then you can use the functions described above for writing and reading. If you use the stream only for reading, just close the memory stream when you're finished. If you write to the stream, you need to explicitly save the data written before closing:

```
NXSaveToFile(stream, "aPathname");
```

```
NXCloseMemory(stream, NX_FREEBUFFER);
```

To open a file stream using a file descriptor, call **NXOpenFile()**, giving it the descriptor and specifying how the stream will be used:

```
NXStream  *stream;
stream = NXOpenFile(fd, NX_WRITEONLY);
```

If the file descriptor was obtained through the system call **open()**, one of three flags (0_WRONLY, 0_RDONLY, or 0_RDWR) was used to open the file for writing, reading, or both. The meaning of this flag must match that used in the call to **NXOpenFile()**. See the UNIX manual page on **open()** for more information about its arguments.

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.) The following example allows you to read from **stdin**.

```
int        fileDescriptor;
NXStream  *stream;

fileDescriptor = fileno(stdin);
stream = NXOpenFile(fileDescriptor, NX_READONLY);
```

After you've finished with the file stream, you need to disconnect it from the file and free the storage used by the stream:

```
NXClose(stream);
```

NXClose() saves any data you wrote to the stream in the file, but it doesn't release the file descriptor. To release the descriptor, use the system call **close()**, giving it the descriptor as an argument.

Connecting to a Mach Port

In Mach, tasks and threads communicate among themselves and with the operating system kernel by sending

messages. These messages must adhere to a certain structure. They're sent to a Mach port using the Mach function **msg_send()**, where they're queued until read by the receiver using **msg_receive()**. Rather than setting up this message structure yourself, you can connect a stream to a Mach port using **NXOpenPort()**. Then when you use **NXWrite()** or **NXRead()** (depending on whether you are sending or receiving data), the data you send or receive will be sent to or dequeued from the port you specify. Mach ports and messages are described in more detail in the *NeXT Operating System Software* manual.

Mach port streams can't be opened for both writing and reading, so you need to connect one stream to a port to send data and another stream to the same port to read that data. The following paragraphs show how to set up such a pair of streams.

To send data to a Mach port, first open a stream using **NXOpenPort()**. A port must be previously allocated using the Mach function **port_allocate()**; see the *NeXT Operating System Software* manual for more information about using this function.

```
port_t     thePort;
NXStream  *outStream;

outStream = NXOpenPort(thePort, NX_WRITEONLY);
```

Since this is the sender's stream, it's opened for writing. Now, using any of the functions for writing described above, write to the opened stream the data you want to send to the port. You should close the stream after you finish writing to it:

```
NXClose(outStream);
```

This ensures that all data is actually sent to the port by flushing the buffer associated with the stream. If you want to keep the stream open, you can flush the buffer using **NXFlush()**:

```
NXFlush(outStream);
```

To read this data, the receiver opens a stream on the same port for reading. This can be done independently of when the sender's stream was opened and when the data was sent.

```
port_t     thePort;
NXStream  *inStream;

inStream = NXOpenPort(thePort, NX_READONLY);
```

The functions for reading data will wait until it's available and then read it into the specified location. After the data has been read, the stream can be closed:

```
NXClose(inStream);
```

Archiving to a Typed Stream

Archiving is the process of preserving a data structure, especially an object, for later use. An archived data structure is usually stored in a file, but it can also be written to memory, copied to the pasteboard, or sent to another application. Archiving involves writing data to a special kind of data stream, called a *typed stream*. During unarchiving, memory is allocated for the data structure, and it's initialized with values read from a typed stream.

Typed streams are an abstraction built on the streams abstraction discussed earlier in this chapter. Because of this relationship, data can be written to and read from a typed stream without regard to what destination or source the stream is connected to. Once you've written the code to archive a data structure, you can use that code to store the data structure in a file, write it to memory, or send it to a Mach port.

Typed streams are used for archiving because they provide some protection against future changes that might affect the ability to unarchive a data structure. When a typed stream is used, the data type is archived along with the data and, in the case of objects, the object's class hierarchy and version are also archived. This additional information is checked when a data structure is unarchived, and an exception is raised if necessary. Typed streams also provide some degree of data portability between machines.

The archiving functions make it easy to write structures consisting of several different data types, including objects. Archiving with a typed stream also ensures that objects are written only once even if several members of a data structure refer to the same object. In addition, when archiving an object, you can limit the scope of what's archived by deciding which objects referred to by **id** instance variables should be archived. Classes defined in the Application Kit and the common classes archive themselves using typed streams. If you include instances of these classes (or subclasses) in a data structure, you'll want to archive it using typed streams.

All functions mentioned in this section are described in the *NeXTstep Reference* manuals.

Archiving a Data Structure

To write data to a typed stream, you use any of several functions. Two of these, **NXWriteType()** and **NXWriteTypes()**, allow you to specify the data type or types being written. Other functions write a specific data type; for example, **NXWritePoint()** writes an NXPoint structure. The archiving functions are listed below and discussed in the following sections. The functions for unarchiving are similar to these, and they're listed and described later.

Function	Data Type
NXWriteType()	Single specified type
NXWriteTypes()	Multiple specified types
NXWriteArray()	An array
NXWritePoint()	An NXPoint structure
NXWriteSize()	An NXSize structure
NXWriteRect()	An NXRect structure
NXWriteObject()	An **id**
NXWriteObjectReference()	An **id**

All these functions take a pointer to a typed stream as their first argument. Since you can write to a typed

stream without knowing what it's connected to, opening a typed stream and writing to it are described separately. The ªOpening and Closing a Typed Streamº section below explains how to obtain a typed stream pointer.

Archiving Arbitrary Data

NXWriteType() and **NXWriteTypes()** write strings of data to a typed stream. (These functions are similar to the **printf()** standard C function, which is described in its UNIX manual page.) They take 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 as arguments. The format string characters and their corresponding data types are listed below. They're described in more detail in *NeXTstep Reference, Volume 2*.

Format Character	Data Type
c	char
s	short
i	int
f	float
d	double
@	id
*	char *
%	NXAtom
:	SEL
#	class
!	int; corresponding data won't be read or written
{*type*}	struct
[*count type*]	array

NXWriteType() writes data as the single data type specified by its format string. **NXWriteTypes()** writes multiple types of data. The types are listed in the format string using the appropriate format characters shown above, and pointers to matching data are listed as the last arguments. This example shows three different data types

being written to a typed stream:

```
float   aFloat = 3.0;
int     anInt = 5;
char   *aCharStar = "foo";

NXWriteTypes(typedStream, "fi*", &aFloat, &anInt, &aCharStar);
```

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()**; this implementation results in the corresponding archiving and unarchiving functions taking the same arguments.

Both functions are particularly useful for writing structures consisting of several kinds of data. For example, this structure

```
typedef struct {
    float  aFloat;
    int    anInt;
    char  *aCharStar;
} MyStruct;
```

would be written as follows:

```
NXWriteType(typedStream, "{fi*}", &MyStruct);
```

Both **NXWriteType()** and **NXWriteTypes()** write objects if the ª@º format character is used, which is equivalent to calling **NXWriteObject()**. The section ªArchiving Objectsº below explains the issues involved in writing objects and the different ways of archiving them.

Archiving Arrays and NXPoint, NXSize, and NXRect Structures

For convenience, several functions are provided to archive specific kinds of data structures. These structures can all be written using **NXWriteType()** or **NXWriteTypes()**, but it's easier to use the specialized functions.

NXWriteArray() writes an array to the typed stream passed as its first argument. You specify the number of elements in the array and their type. The following is an example of an integer array being written.

```
int  myArray[4];

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

Note that **NXWriteArray()** takes an array, not a pointer to an array, as an argument.

NXWritePoint(), **NXWriteSize()**, and **NXWriteRect()** work through **NXReadType()** to write NXPoint, NXSize, or NXRect structures to a typed stream. The following example shows these three data structures being archived.

```
NXPoint  zeroPoint = {0.0, 0.0};
NXSize   rectSize = {100.0, 200.0};
NXRect   aRect = {zeroPoint, rectSize};

NXWritePoint(typedStream, &zeroPoint);
NXWriteSize(typedStream, &rectSize);
NXWriteRect(typedStream, &aRect);
```

Archiving Objects

Archiving an object begins with a call to either **NXWriteRootObject()** or **NXWriteObject()**. These functions take a pointer to a typed stream and an object's **id** as arguments. They send the object a **write:** message, passing it the typed stream. The **write:** method contains the code that writes the values of the object's instance variables to the typed stream. (Note that **NXWriteObject()** is equivalent to using **NXWriteType()** or **NXWriteTypes()** and specifying ª@º in the format string; in the following discussion, **NXWriteObject()** will be used as a proxy for all these equivalent methods of writing objects.)

NXWriteRootObject() and **NXWriteObject()** differ in how they expect the object's **write:** method to handle its **id** instance variables. **NXWriteObject()** expects to be able to archive every object referred to by **id** instance

variables, as well as objects referred to by those objects, and so on. **NXWriteRootObject()** allows you to limit the scope of what's archived by letting some **id** instance variables point to **nil** when they're unarchived. The next sections describe how to set up a **write:** method and when to use these two functions.

A third function, **NXWriteRootObjectToBuffer()**, also begins the process of archiving a given object. This function doesn't take a typed stream as an argument. Instead, it opens a typed stream on memory, writes the object to it, and returns a pointer to the memory buffer. This function is discussed in more detail below under ªOpening and Closing a Typed Stream.º

The write: Method

Any Application Kit class or Common Class that declares instance variables already has a **write:** method that archives those instance variables. You need to supply a **write:** method for any class you create that adds instance variables. However, not every single instance variable needs to be archived. If an instance variable can be initialized by using the values of other instance variables, you don't need to archive its value.

Note that **write:** messages shouldn't be sent directly to objects. They should only be generated by the functions **NXWriteRootObject()** and **NXWriteObject()**.

Every **write:** method should begin with a message to **super**:

```
- write:typedStream {
    [super write:typedStream];
    . . .  /* code for writing instance variables declared in this
              class */
}
```

This ensures that the object's class hierarchy and its inherited instance variables are archived. The body of the **write:** method uses the appropriate functions to archive the instance variables declared in that class. You can use any of the functions listed above in the ªArchiving a Data Structureº section. If the object being archived has **id** instance variables, they're archived as described below.

Archiving id Instance Variables

An object's **id** instance variables can be archived in one of two ways, depending on whether the object referred to by the instance variable is an intrinsic part of the object being archived. If it is intrinsic, use **NXWriteObject()**, **NXWriteType()**, or **NXWriteTypes()**, which are all equivalent. If it's not intrinsic, use **NXWriteObjectReference()**. The following paragraphs explain the differences among these functions.

An object's **id** instance variables may 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. These kinds of instance variables are archived using **NXWriteObject()**. The following shows part of a View's **write:** method:

```
- write:(NXTypedStream *) typedStream {
    [super write:typedStream];
    NXWriteObject(typedStream, subviews);
    . . .  /* code for writing other instance variables */
}
```

If you design a subclass of View that defines instance variables, you'll need to create a **write:** method that archives those instance variables. Since your method will begin with a message to **super**, the subviews list will be archived along with the View. Button objects don't define instance variables, so they inherit Control's **write:** method, which archives the **cell** instance variable.

In some 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. These kinds of instance variables are archived using **NXWriteObjectReference()**.

NXWriteObjectReference() specifies that a pointer to **nil** should be written for the **id** 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 the pointer will point to the archived object.

Archiving an Object with id Instance Variables

When an object that includes any calls to **NXWriteObjectReference()** is archived, **NXWriteRootObject()** must be used to archive the object instead of **NXWriteObject()**. If the object being archived is based on the Application Kit, **NXWriteRootObject()** should be used since several Application Kit classes use **NXWriteObjectReference()**. Using **NXWriteRootObject()** will always give the desired result whether **NXWriteObjectReference()** is called or not. However, **NXWriteObject()** will raise an exception if used to archive an object that calls **NXWriteObjectReference()**.

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 instance variables are written with **NXWriteObjectReference()**. On the second pass, **NXWriteRootObject()** archives the data structure. Because of this two-pass implementation, **write:** methods are performed twice; therefore, **write:** methods shouldn't contain any code that has side effects.

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.

Unarchiving a Data Structure

The functions for unarchiving data are similar to the functions for writing:

Function	Data Type
NXReadType()	Single specified type
NXReadTypes()	Multiple specified types

NXReadArray()	An array
NXReadPoint()	An NXPoint structure
NXReadSize()	An NXSize structure
NXReadRect()	An NXRect structure
NXReadObject()	An **id**

With the exception of **NXReadObject()**, these functions take the same arguments as their counterparts for archiving. Rather than writing the data pointed to by the arguments, however, the unarchiving functions read data from the typed stream into locations specified by the function's arguments. **NXReadObject()** takes only a typed stream as an argument and returns the unarchived object's **id**. The section ªUnarchiving Objectsº below contains more information about reading an object from a typed stream and initializing it.

In the following example, a **float**, an **int**, and a **char** * are read from a typed stream and stored in the locations specified by the last three arguments to **NXReadTypes()**.

```
float  aFloat;
int    anInt;
char  *aCharStar;

NXReadTypes(typedStream, "fi*", &aFloat, &anInt, &aCharStar);
```

All the functions for reading check the type of data on the stream and raise an exception if the type isn't what's expected.

Unarchiving Objects

Unarchiving an object from a typed stream is initiated by a call to **NXReadObject()**. Because an object's class hierarchy is archived with the object, **NXReadObject()** can determine the object's class and allocate enough memory for a new instance of that class. It then initializes the object's instance variables by sending it a **read:** message, which reads values for the instance variables from the typed stream.

The read: Method

read: methods have already been defined for all Application Kit classes and common classes that declare instance variables. You need to supply a **read:** method for any class you create that adds instance variables. As with **write:** methods, **read:** messages shouldn't be sent directly to objects. They should only be generated by **NXReadObject()**.

Every **read:** method should begin with a message to **super**:

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

This ensures that the object's inherited instance variables are unarchived. The body of the **read:** method uses the appropriate functions to unarchive the instance variables declared in that class, in the order in which they were archived in the **write:** method. Any of the functions listed above in the ªUnarchiving a Data Structureº section can be used to read values for instance variables.

Unarchived **id** instance variables are initialized to point either to an object or to **nil**, depending on whether the referenced object was archived. If **NXWriteObjectReference()** was used for the **id** instance variable and if the referenced object isn't an intrinsic part of any member of the structure that was archived, then the instance variable will point to **nil**. Otherwise, it'll point to the object. See ªArchiving Objectsº earlier for more information.

Values for other instance variables may not have been archived because they can be derived from others. Values for these instance variables should be computed in the **read:** method. Other initialization needed can be performed as described in the next section.

If you create a class, archive an instance of it, and later create a new version of that class (for example, you decide to add an instance variable), you can set up your **read:** method to read both versions. When a class is

created, its version should be set using Object's **setVersion:** method:

```
@implementation MyClass:MySuperClass
+ initialize
{
    [MyClass setVersion:MYCLASS_CURRENT_VERSION];
    return self;
}
```

The **read:** method for this class can then check the version being unarchived by using **NXTypedStreamClassVersion()** and, if necessary, use different code for reading an old version:

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

Initializing an Object

Immediately after an object has been read from a typed stream, **NXReadObject()** sends it an **awake** message. This gives the object a chance to perform initialization tasks that can't be done in the **read:** methodÐthat is, those tasks that require the entire object to be unarchived and in a usable state. For example, Window's **awake** method has the Window Server redisplay the window and assign it a window number. If you override any of the Application Kit's **awake** methods, your version should begin by sending an **awake** message to **super**.

After sending an **awake** message, **NXWriteObject()** sends the object a **finishUnarchiving** message. The purpose of this method is to allow 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.

Opening and Closing a Typed Stream

The functions for archiving and unarchiving take an already opened typed stream as an argument. You can use one of three functions to open a typed stream, depending on whether you're archiving to or unarchiving from a file, memory, or some other destination or source:

- **NXOpenTypedStreamForFile()** returns a pointer to a typed stream opened on a specified file.
- **NXWriteRootObjectToBuffer()** opens a typed stream on memory and writes the given object to it using **NXWriteRootObject()**. The corresponding function for unarchiving, **NXReadObjectFromBuffer()**, opens a memory stream and reads an object from it.
- **NXOpenTypedStream()** takes an already opened NXStream structure as an argument and returns a pointer to a typed stream.

Regardless of what the typed stream is opened on, the same archiving code (or unarchiving code) can be used for a given data structure.

A Typed Stream on a File

NXOpenTypedStreamForFile()'s two arguments are the pathname of a file and a constant that indicates whether you'll be archiving or unarchiving:

```
NXTypedStream  *typedStream;
```

```
typedStream = NXOpenTypedStreamForFile("yourPathname", NX_WRITEONLY);
```

This function returns a pointer to a typed stream on memory and makes note of the fact that the stream is associated with a file. If you open the stream for archivingÐby specifying NX_WRITEONLYÐyou can use any of the functions described above in ªArchiving a Data Structure.º When you've finished, call **NXCloseTypedStream()**. This function saves the contents of the typed stream in the file, creating it if necessary, and closes the stream.

If NX_READONLY is specified, the typed stream is initialized with the contents of the file specified. You unarchive data by using any of the functions described above in ªUnarchiving a Data Structure.º Then call **NXCloseTypedStream()** to close the typed stream.

A Typed Stream on Memory

NXWriteRootObjectToBuffer() and **NXReadObjectFromBuffer()** both open a stream on memory. They're particularly useful for archiving an object, writing it to the pasteboard, and then unarchiving it from the pasteboard. See ªThe Pasteboardº later in this chapter for an example of using these functions in conjunction with the pasteboard.

NXWriteRootObjectToBuffer() opens a memory stream, writes the object given as its argument by calling **NXWriteRootObject()**, and then closes the stream. It returns the size of the object written, in the location specified by the second argument, and a pointer to the memory buffer.

```
char  *data;
int    length;

data = NXWriteRootObjectToBuffer(anId, &length);
```

NXReadObjectFromBuffer()'s two arguments should be taken from a previous call to **NXWriteRootObjectToBuffer()**:

```
id  someId;
```

```
someId = NXReadObjectFromBuffer(data, length);
```

NXReadObjectFromBuffer() calls **NXReadObject()** to read the object and then closes the typed stream and returns the object's **id**.

When you finish with the memory buffer, after a call to either **NXWriteRootObjectToBuffer()** or **NXReadObjectFromBuffer()**, free the buffer by calling **NXFreeObjectBuffer()**. This function takes the same arguments as **NXReadObjectFromBuffer()**.

Using an NXStream Structure

In addition to the functions described above for opening a typed stream on file or memory, you can open a typed stream by passing **NXOpenTypedStream()** a pointer to an NXStream structure. NXStream structures can be opened on Mach ports, memory, files, or even objects. To obtain an NXStream pointer, use the functions described in ªStreamsº earlier in this chapter.

NXOpenTypedStream()'s second argument should be NX_READONLY or NX_WRITEONLY to specify whether you'll be archiving or unarchiving. This constant should be the same as that used to open the NXStream structure. When you finish archiving or unarchiving, you need to close the NXStream structure and the typed stream. The section on ªStreamsº describes how to close NXStream structures. Use **NXCloseTypedStream()** to close the typed stream.

The Defaults System

Through the defaults system, you can allow users to customize your application to match their preferences. For example, you can let users express a preference for where the main menu of your application should come up the next time it's launched. This preference will override the default location of the main menu that users can

set with the Preferences application. An application records such preferences by assigning default values to a set of parameters. Each user has a defaults database named **.NextDefaults**, which resides in the **.NeXT** subdirectory in the user's home directory, for storing these default values.

Warning: The **.NeXTDefaults** file should never be accessed directly. Values in it can be read and written using the functions and commands described in this section.

Since the defaults database is a system resource, it isn't owned by any single application. In fact, any application can store values for parameters or get values stored by another application. For example, the ChoosePrinter panel writes to the defaults database to store the name of the printer selected by the user. Another application may want to obtain this printer specification from the database. Applications can use the functions discussed below to read values from and write them to the database. These functions are described in detail in *NeXTstep Reference, Volume 2*.

Creating a Registration Table

The registration table allows an application to efficiently read default values for a set of parameters without having to open and close the **.NextDefaults** database to obtain each value. The table consists of a list of pairs; each pair is composed of a parameter name and a corresponding default value. The registration table is created at run time by opening the database once to read default values for the parameters the application will use. Every application should create its registration table early in the program, before any default values are needed.

To create this table, call **NXRegisterDefaults()** and give it two arguments: a character string specifying the name of an application, or owner, and an NXDefaultsVector structure. Like the registration table, this structure consists of a list of pairs of parameter names and default values. (It's defined in the header file **appkit/defaults.h**.)

The NXDefaultsVector structure serves two purposes. First, it provides a complete list of all parameters that the

application will use. Values for all the parameters specified are placed in the registration table at once, so the database doesn't need to be opened and closed for subsequent uses of the parameters. (However, if the application later asks for values for parameters that aren't registered, the database will be opened, read, and closed again.) Second, the structure allows the programmer to suggest values for the parameters. These values are used if the user hasn't stated a preference for a specific value.

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 **WriteNowDefaults** for the owner **WriteNow**:

```
+ initialize
{
static NXDefaultsVector WriteNowDefaults = {
    {"NXFont", "Helvetica"},
    {"NXFontSize", "12.0"},
    {NULL}
};

NXRegisterDefaults("WriteNow", WriteNowDefaults);

return self;
}
```

NXRegisterDefaults() creates a registration table that contains a value for each of the parameters listed in the NXDefaultsVector structure. (Note that NULL is used to signal the end of the NXDefaultsVector structure.) This value will be the one listed in the structure if there's no value for that parameter in the database, as described below.

A user's database may contain values for parameters stored multiple times, each with a different owner. For example, the NXFont parameter can have the value Ohlfs with a GLOBAL owner, Times for the owner WriteNow, and Courier for the owner Mail. When searching a user's database for the parameters listed in the NXDefaultsVector structure, **NXRegisterDefaults()** ignores values owned by an application different from the one used as its argument. If it finds a parameter and owner that matches those passed to it as arguments, the corresponding value from the user's database rather than the value from the NXDefaultsVector structure is

placed in the registration table. If no parameter-owner match is found, **NXRegisterDefaults()** searches the database's global parametersÐthat is, those owned by GLOBALÐfor a match, and, if it finds one, places the corresponding value in the registration table. (Global parameters are discussed in a later section.) If a parameter isn't found in the user's database, the parameter-value pair listed in the NXDefaultsVector structure is placed in the registration table.

Note: When creating their own parameters, applications should use the full market name of their product as the owner of the parameter to avoid colliding with already existing parameters. Noncommercial applications might use the name of the program and the author or institution.

If the application was launched from the command line, any parameter values specified there will be used, overriding values listed in the database and the NXDefaultsVector structure. See ªThe Command Lineº below for more information.

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

1. The command line
2. The defaults database, with a matching owner
3. The defaults database, with the owner listed as GLOBAL
4. The NXDefaultsVector structure passed to **NXRegisterDefaults()**

Reading Default Values

To get a value for a parameter, you typically call **NXGetDefaultValue()**. This function takes an owner and a parameter as arguments, as shown below, and returns a **char** pointer to the default value for that parameter.

```
char  *myDefaultFont;
myDefaultFont = NXGetDefaultValue("WriteNow", "NXFont");
```

NXRegisterDefaults() should already have been called, so **NXGetDefaultValue()** first looks in the registration table, where usually it will find a matching parameter and value. If **NXGetDefaultValue()** doesn't find a match in the registration table (which would only be the case if you hadn't listed all parameters when you called **NXRegisterDefaults()**), it searches the **.NextDefaults** database for the owner and parameter. If still no match is found, it searches for a matching global parameter, first in the registration table and then in the database. If the value is found in the database rather than the table, **NXRegisterDefaults()** registers that value for subsequent use.

Occasionally, you may want to search only the database for a default value and ignore the command line and the registration table. For example, you might want a value that another application may have changed after the table was created. In these rare cases, call **NXReadDefault()**, which takes an owner and the parameter 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 **char** pointer to the default 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 registration table with **NXSetDefault()**, which is described below.

Writing Default Values

If you have allowed a user to customize an application, you probably want to write new values into the user's **.NextDefaults** database to store these preferences. You probably also want to put the values in the registration table for efficient access by **NXGetDefaultValue()**. In addition, at various points in your program, you may want to update the registration table with any recent changes to the database. The following paragraphs explain the functions that manipulate the contents of the database and the registration table.

NXWriteDefault() writes a default value into both the database and the registration table. It takes an owner, a parameter, and a default value for that parameter as arguments:

```
NXWriteDefault("WriteNow", "NXFont", "Helvetica");
```

In this example, the NXFont parameter and its value Helvetica are written into both the database and the registration table for the owner **WriteNow**.

Similarly, **NXWriteDefaults()** writes a vector of default values for the given owner into the database and registers them.

```
static NXDefaultsVector WriteNowDefaults = {
    {"NXFont", "Times"};
    {"NXFontSize", "12.0"};
    {NULL};
};
NXWriteDefaults("WriteNow", WriteNowDefaults);
```

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.

NXSetDefault() takes an owner, a parameter, and a value for that parameter as arguments:

```
NXSetDefault("WriteNow", "NXFont", "Helvetica");
```

The parameter and its default value are placed in the registration table, but they aren't written into the **.NextDefaults** database.

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

NXUpdateDefault() updates the value for the single parameter and owner given as its arguments:

```
NXUpdateDefault("WriteNow", "NXFont");
```

NXUpdateDefaults(), which takes no arguments, updates the entire registration table. It checks every parameter in the registration table, determines whether a newer value exists in the database, and puts any newer values it finds in the registration table.

NXRemoveDefault() removes a specified parameter for the given owner from the **.NextDefaults** database.

```
NXRemoveDefault("WriteNow", "NXFontSize");
```

Changing the Defaults Database from a Shell Window

In addition to the functions described above, the following three commands can be used in a Terminal or Shell window to read and write default values:

- **dread -l** reads all the values in the defaults database and sends them to **stdout**. Instead of the **-l** option, you can specify a particular owner and a parameter; if no owner is specified, it's assumed to be GLOBAL.
- **dwrite** takes an owner, a parameter, and a value as arguments and writes the value into the defaults database. If the **-g** option is used, the owner is assumed to be GLOBAL. If no arguments are given, input is taken from **stdin**.
- **dremove** removes the parameter named as an argument from the database. If an owner is specified as the first argument, **dremove** removes that owner's parameter-value pair; if the **-g** option is used, the owner is assumed to be GLOBAL. If no arguments are given, input is taken from **stdin**.

All arguments for these commands should be separated by spaces. For more information on using these commands, see their UNIX manual pages.

The Command Line

Without changing the **.NextDefaults** database, you can temporarily override values in the database or supply values for parameters that don't exist in the database. To do this, specify the desired values when launching an application from a Shell or Terminal window, as shown below:

```
Edit -WidthInChars 100 -HeightInChars 120 SomeFile.m &
```

In this example, Edit will be launched, in a window that's 100 characters wide and 120 characters high. When **NXRegisterDefaults()** is called, the command-line values will be placed in the registration table, overriding values specified by the database and the NXDefaultsVector structure. However, these values will not be written into the database.

System and Global Parameters

The Application Kit registers values for system and global parameters. System parameters are used by all applications for such things as determining which printer to use and which font to use in attention panels. Values for system parameters should remain constant across the system, so applications shouldn't overwrite system values. Values for global parameters are used by applications if there's no application-specific value. Global parameters determine the location of the main menu and the font used to display text, for example. Applications are encouraged to declare their own, application-specific, values for global parameters.

The following sections list the system and global parameters and describe their meaning. Parameters owned by the Workspace Manager are also discussed since they sometimes affect multiple applications. The parameters owned by Edit, Shell, and Terminal are described in the *NeXT Development Tools* manual. (All parameter names and their values are character strings; for simplicity, they're shown below without quotation marks.)

System Parameters

Applications obtain values for system parameters by specifying ªSystemº as the owner in one of the functions described above for reading default values. Users can set values for some of these parameters through the Preferences applications. (For more information about Preferences, see *The NeXT User's Reference Manual*.)

The system parameters and their initially registered values are listed below.

Parameter	Initial Value
SystemAlert	Both
UnixExpert	NO
PublicWindowServer	NO
Umask	18
BrowserSpeed	50
Printer	Local_Printer
PrinterHost	NULL
PrinterResolution	400
SystemFont	Helvetica
BoldSystemFont	Helvetica-Bold
ScrollerButtonDelay	0.5
ScrollerButtonPeriod	0.025

Users can set default values for the first five parameters through the Preferences application.

- The SystemAlert parameter allows applications to offer voice as well as panels for system alerts.
- If the UnixExpert parameter is set to NO, all UNIX system files will be hidden.

- The PublicWindowServer parameter determines whether processes that are not descended from the Workspace Manager have host access to the computer.
- Values for the Umask parameter are integers that correspond to the octal values used by the **umask()** system call to set the file-creation mask.
- The BrowserSpeed parameter determines how fast scrolling will be when the user clicks on a browser's scroll button. Values for this parameter can range from 0 to 100.

The next three parameters concern printing.

- Printer specifies the printer that will be used. Valid printers are listed in the ChoosePrinter panel that's opened through the Choose button in the Print panel.
- PrinterHost specifies the host machine of the printer. The default value, NULL, indicates the local printer's host.
- Printing can be performed with a resolution of either 300 or 400 dpi.

The next two pairs of parameters specify the font used by the system to display text and how scroll buttons will respond.

- The parameters SystemFont and BoldSystemFont are stored in global variables NXSystemFont and NXBoldSystemFont, respectively, for easy use by the Application Kit. The Kit uses these variables, for example, to display text in attention panels and in Cells of type NX_TEXTCELL.
- The value for the ScrollerButtonDelay parameter specifies how many seconds the user must hold down the mouse button to make a scroll button repeat. ScrollerButtonPeriod indicates the interval, also in seconds, at which the scrolling action will be repeated if the user continues to hold down the mouse button.

Global Parameters

The Application Kit registers values for global parameters in the Application object's **initialize** method. Users can set values for some of these parameters using the Preferences application. Preferences writes values specified by a user into the defaults database (with owner set to GLOBAL) so they will override the initial values supplied by the Application object.

The global parameters that have been defined on the NeXT computer and their initially registered values are listed below. The following paragraphs discuss the possible values for these parameters and their meaning.

Parameter	Initial Value
NXFont	Helvetica
NXFontSize	12
NXMenuX	-1.0
NXMenuY	1000000.0
NXFixedPitchFont	Ohlfs
NXFixedPitchFontSize	10
NXPaperType	Letter
NXMargins	72 72 90 90
NXAutoLaunch	NO
NXCaseSensitiveBrowser	NULL
NXHost	NULL
NXOpen	NULL
NXOpenTemp	NULL
NXShowAllWindows	NULL

NXShowPS NULL
NXMallocDebug 1

NXPSName NULL

Users can set default values for the first four parameters through the Preferences application.

- Because the initial value for the NXFont parameter is 12-point Helvetica, applications that create documents will use this font by default. However, many applications assign their own values to the NXFont and NXFontSize parameters, and most of these applications also provide a Font panel through which users can change the values.
- NXMenuX and NXMenuY specify the location of the main menu of the application. The initially registered values are off the screen, so applications will probably want to supply their own values.

The next two parameters affect the font of applications that use fixed-width fonts, such as Shell and Terminal.

- The default value is 10-point Ohlfs font. NXFixedPitchFont must be set to a fixed-width font, such as Courier or Ohlfs, rather than a variable-width font, such as Times.

The next pair of parameters concerns printing.

- NXPaperType must be one of the standard paper types for PostScript documents such as Letter, Legal, or A4.
- The NXMargins parameter specifies the printing area on the page; the initial setting is appropriate for letter-size paper.

Values for the next two parameters indicate whether an application was automatically launched at login and whether an application's browser ignores case.

- The Workspace Manager passes YES as the value for NXAutoLaunch if the application was automatically launched when the user logged in. (See the description of the LaunchThese parameter below under ªWorkspace Manager's Parameters.º)

- Applications that create browsers can use the NXCaseSensitiveBrowser parameter to determine whether they should ignore case when alphabetizing the browser's contents.

You can use the command line to specify values for the last six parameters, which are used only at launch time and shouldn't be written to the database. See ªThe Command Lineº above for more information on how to do this.

- The NXHost parameter enables you to run an application on one machine while sending the PostScript code generated to another machine. The host machine will display windows and accept events from the user.
- The NXOpen parameter specifies the name of the file to be opened by the application being launched. If NXOpenTemp is used to specify a file, that file won't be saved when the application quits unless you explicitly tell the application to save it.

NXShowPS, NXShowAllWindows, and NXMallocDebug control the display of debugging output. By default, NXShowPS and NXShowAllWindows are turned off.

- NXShowPS writes to **stderr** both the PostScript code produced by the application and values returned from the PostScript interpreter to the application.
- NXShowAllWindows displays all off-screen windows created by the application. These windows typically contain Bitmap objects used for compositing into on-screen windows.
- The value of NXMallocDebug is passed to the **malloc_debug()** function, which controls the amount of error checking that **malloc()** performs. The UNIX manual page on **malloc()** contains more information about both these functions.

NXPSName is used to establish a connection with the Window Server:

- The Application Kit uses the value of NXPSName to look up the Window Server from the Network Name Server.

Workspace Manager's Parameters

The parameters belonging to the Workspace Manager and their initial values are shown below.

Parameter	Initial Value
LaunchThese	Preferences
IconsSnapTo	YES
BrowserColWidth	120
ApplicationPaths	~/Apps:/LocalApps:/NextApps:/NextDeveloper/Apps: /NextAdmin:/NextDeveloper/Demos
CoreLimit	NULL
BrowserX	265
BrowserY	287
BrowserW	534
BrowserH	296

The first two parameters can be set through panels brought up by the Workspace Manager. They're documented in more detail in *The NeXT User's Reference Manual*.

- LaunchThese indicates which applications are automatically launched when the user enters the workspace.
- IconsSnapTo specifies whether icons displayed in the Icon view are aligned on the grid.

The next three parameters can be set by using the command line at launch time or by writing them into the database from a Shell or Terminal window.

- BrowserColWidth specifies the width of the columns of the Directory Browser.
- The Workspace Manager searches for application programs in the colon-separated directory list in

ApplicationPaths. See ªPathsº in Chapter 2, ªThe NeXT User Interface,º for more information about how the Workspace Manager uses this parameter.

- CoreLimit places a limit on the size of core files for Workspace Manager and its children. If a program dies, the system will write a core file if you have write permission in the working directory of the program and if CoreLimit is larger than the size of the core image. The default value, NULL, means that Workspace Manager inherits its parent's core limit.

The remaining four parameters set the position and size of the Browser.

- BrowserX and BrowserY specify the x- and y-coordinates of the lower left corner of the Directory Browser window. BrowserW and BrowserH specify its width and height. Values for these parameters shouldn't be set directly; they're set simply by moving or resizing the Browser on-screen.

The Pasteboard

The pasteboard is the principal means by which users can move data within and between applications. It supports a cut/copy/paste user-interface paradigm. To the user, there is a single pasteboard that all applications share, providing a unified environment. Also from the user's point of view, there is a single thing in the pasteboard at a given time. Internally, however, the pasteboard may contain more than one representation of its contents. For example, if a user cuts a piece of text from a word processor, that text replaces whatever was previously held in the pasteboard; however, that text may be represented in the Pasteboard object by an ASCII string and a piece of PostScript code at the same time.

Using the Pasteboard

Applications using the pasteboard perform all operations through a single instance of the Pasteboard class. This global Pasteboard object is accessed by sending a **pasteboard** message to the Application object:

```
id  myPboard;
myPboard = [NXApp pasteboard];
```

The Pasteboard object manages all communications with **pbs**, the pasteboard server. All data read from or written to the pasteboard goes through **pbs**. Data for a particular type is transmitted as a single, contiguous buffer of memory. Since data is transmitted using Mach messaging, these buffers are shared among applications, making the communication very efficient for large quantities of data. Essentially, each application that has used the data has a pointer to the same, shared physical memory, even though it may appear in different ranges of their address spaces.

Declaring Data Types

When an application performs a copy (or a cut), it first becomes the owner of the pasteboard by declaring what types of data it will put in the pasteboard. It does this with the **declareTypes:num:owner:** method. The first two arguments for this method are a list of all possible representations for the selection being copied and the number of types in that list. An application can write any of the standard pasteboard data types defined by NeXT. It can also write its own data types for its own use, or for use among a cooperating set of applications. (Data types are named by null-terminated character strings.) The standard data types and their corresponding global variables, which are declared in the Application Kit header file **Pasteboard.h**, are listed below.

Data Type	Global Variable
Plain ASCII text	NXAsciiPboard
Rich Text Format (RTF) version 1.0	NXRTFPboard
Encapsulated PostScript (EPS) version 1.2	NXPostScriptPboard
Tag Image File Format (TIFF) version 5.0	NXTIFFPboard

NeXT sound pasteboard data type	NXSoundPboard

The ASCII and RTF types both describe text. Clients of the pasteboard that handle text should always declare and be able to accept ASCII data. If they can also produce or read RTF, they should declare that type as well.

The pasteboard owner, which is the third argument for **declareTypes:num:owner:**, promises to supply data in all the representations declared. When copying data to the pasteboard, as described below, the owner can choose to delay writing a type until that type is requested, or it can supply all representations at one time. If writing will be delayed, the owner must be an object that won't be freed so that it can be informed when data has been requested. If all representations will be supplied at one time, the owner can be NULL.

Copying Data to and Reading it from the Pasteboard

After declaring the data types, data can be written to the pasteboard with the **writeType:data:length:** method. The first argument specifies the type of the data, and the second points to the data to be written. The length argument specifies the number of bytes of data. This method is called each time a different type is written to the pasteboard.

When an application performs a paste, it first examines the available data types in the pasteboard. The **types** method returns a null-terminated array of character strings describing the available types. If the application finds a data type that's appropriate, it requests the data with the **readType:data:length:** method. If that data representation has not yet been written to the pasteboard, the owner specified in the **declareTypes:num:owner:** method is sent a **provideData:** message with the type requested as an argument. The owner must then write that type of data to the pasteboard. (If the application quits before supplying all declared data types, a **provideData:** message will also be sent. This only works if the application quits using Application's **terminate:** method.)

Applications that do significant calculation to import a certain type may be able to save this work on repeated pastes of the same data by checking the change count. The change count is an integer (returned by the **changeCount** method) that increments every time a set of types for the pasteboard is declared. If the change

count is the same as it was during a previous paste, the same data is being imported.

The following section gives examples of the process of copying data to and reading it from the pasteboard. All functions mentioned below are described in more detail in *NeXTstep Reference, Volume 2*.

Examples of Preparing and Parsing Data

An NXStream or NXTypedStream structure can be helpful in creating and interpreting the buffers of data that the pasteboard deals with. They're similar to the stream model and interface in the UNIX **stdio** library, but they can read and write to memory and Mach ports as well as UNIX file descriptors. Using a stream, the same code can be used to interpret a buffer of a certain data type from the pasteboard as can be used to read a disk file of the same format. (See ªStreamsº in this chapter for more information.) A special kind of data stream, a typed stream, should be used for copying Objective-C objects to the pasteboard. Typed streams are discussed in more detail in ªArchiving to a Typed Streamº earlier in this chapter.

The next two sections contain examples of using a stream and a typed stream with the pasteboard.

Using a Stream

In the following example, a View writes the PostScript representing itself to a stream and then copies it to the pasteboard.

```
- copy:sender
{
    id          pb = [NXApp pasteboard];
    NXStream   *stream;
    char       *data;
    int         length;
```

```
    [pb declareTypes:&NXPostScriptPboard num:1 owner:self];
    stream = NXOpenMemory(NULL, 0, NX_WRITEONLY);
    [self copyPSCodeInside:NULL to:stream];
    NXGetMemoryBuffer(stream, &data, &length, &maxLength);
    [pb writeType:NXPostScriptPboard data:data length:length];
    NXCloseMemory(stream, NX_FREEBUFFER);
    return self;
}
```

The **declareTypes:num:owner:** method readies the pasteboard to receive a single type of data, PostScript. Then a stream that writes to memory is opened using **NXOpenMemory()**. Next the View's PostScript code is written to the stream with the **copyPSCodeInside:to:** method. The contents of the stream are obtained with **NXGetMemoryBuffer()** and are then transferred to the pasteboard through **writeType:data:length:**. Finally, the stream is closed.

This is what the corresponding **paste:** method might look like:

```
- paste:sender
{
    id         pb = [NXApp pasteboard];
    char     **type;
    char      *data;
    int        length;
    NXStream  *stream;

    for(type = [pb types]; *type; type++)
        if(!strcmp(*type, NXPostScriptPboard))
            break;

    if(*type) {
        [pb readType:NXPostScriptPboard data:&data length:&length];
        stream = NXOpenMemory(data, length, NX_READONLY);
        /* parse PostScript data using NXGetc() or NXScanf() */
        . . .
```

```
        NXCloseMemory(stream, NX_FREEBUFFER);
    }
    else
        /* invalid data type - raise an exception */;
        . . .
    return self;
}
```

The **paste:** method first ensures that PostScript code is one of the pasteboard's available data types. Then it reads the PostScript data from the pasteboard with **readType:data:length:** and opens a memory stream on the data using **NXOpenMemory()**. The data can be parsed using **NXGetc()** or **NXScanf()** and pasted in, after which the stream is closed. Data read from the pasteboard is allocated using **vm_allocate()**, so it must be freed using **vm_deallocate()**. **NXCloseMemory()** does this automatically if NX_FREEBUFFER is specified.

Using a Typed Stream

A typed stream should be used to copy an Objective-C object to and read it from the pasteboard. A typed stream writes an object's class hierarchy as well as both the data type and value of the object's instance variables.

The example below writes an object to a typed stream using the function **NXWriteRootObjectToBuffer()** and then puts it on the pasteboard.

```
-copy:sender
{
    const char  *const types[1] = {"PrivateTypes"};
    id           pb = [NXApp pasteboard];
    char        *data;
    int          length;

    [pb declareTypes:types num:1 owner:self];
    data = NXWriteRootObjectToBuffer(SelectionList, &length);
```

```
    [pb writeType:types[0] data:data length:length];
    NXFreeObjectBuffer(data, length);
    return self;
}
```

In this example, the data to be written to the pasteboard exists in SelectionList, which might be a List object, for example. **NXWriteRootObjectToBuffer()** opens a typed stream on memory, writes the object given as its argument, and then closes the stream. It also returns both the size of the object (in the location specified by **length**) and a pointer to the memory buffer itself, which is truncated to the size of the object. The contents of this buffer can then be written to the pasteboard with **writeType:data:length:**. Finally, the typed stream and the data are freed with **NXFreeObjectBuffer()**.

The following method reads the object from the pasteboard:

```
-paste:sender
{
    char  **type;
    id      pb = [NXApp pasteboard];
    char   *data;
    int     length;
    id      PasteList;

    for(type = [pb types];*type;type++) {
        if(!strcmp(*type,"PrivateTypes"))
            break;

    }
    if(*type) {
        [pb readType:*type data:&data length:&length];
        pasteList = NXReadObjectFromBuffer(data, length);
    NXFreeObjectBuffer(data, length);
    }
    /*code for pasting in the data*/
    . . .
```

```
        return self;
    }
```

The **for** loop shown above checks whether the desired data type is in the pasteboard. If so, the corresponding data is read from the pasteboard into the typed stream with the **readType:data:length:** method. **NXReadObjectFromBuffer()** then opens a typed stream, reads the data into **data**, closes the stream, and returns the buffer. Since in this case the buffer won't be reread, it's freed with **NXFreeObjectBuffer()**.

Responding to Cut, Copy, and Paste

Interface Builder provides your application with a main menu containing a standard Edit submenu with Cut, Copy, and Paste commands. These commands are initialized to send the **cut:**, **copy:**, and **paste:** messages to the first responder, and thus through the responder chain. Editable Application Kit classes, like Text, implement the **cut:**, **copy:**, and **paste:** methods, and therefore respond to these menu choices without any explicit connections from the menu items. An application's View subclasses that support cut, copy, and paste should allow themselves to become the first responder, implement **cut:**, **copy:**, and **paste:** methods, and let the standard menu items and the responder chain find these implementations.

Exception Handling

For up-to-date information on exception handling, see the documentation in:

/NextLibrary/Documentation/NextDev/Concepts/ExceptionHandling.rtf