NeXTstep Reference
Volume 1

NeXT Developer's Library

NeXTstep

Draw upon the library of software contained in NeXTstep to develop your applications. Integral to this development environment are the Application Kit and Display PostScript.

Concepts

A presentation of the principles that define NeXTstep, including user interface design, object-oriented programming, event handling, and other fundamentals.

Reference, Volumes 1 and 2

Detailed, comprehensive descriptions of the NeXTstep Application Kit software.

Sound, Music, and Signal Processing

Let your application listen, talk, and sing by using the Sound Kit and the Music Kit. Behind these capabilities is the DSP56001 digital signal processor. Independent of sound and music, scientific applications can take advantage of the speed of the DSP.

Concepts

An examination of the design of the sound and music software, including chapters on the use of the DSP for other, nonaudio uses.

Reference

Detailed, comprehensive descriptions of each piece of the sound, music, and DSP software.

NeXT Development Tools

A description of the tools used in developing a NeXT application, including the Edit application, the compiler and debugger, and some performance tools.

NeXT Operating System Software

A description of NeXT's operating system, Mach. In addition, other low-level software is discussed.

Writing Loadable Kernel Servers

How to write loadable kernel servers, such as device drivers and network protocols.

NeXT Technical Summaries

Brief summaries of reference information related to NeXTstep, sound, music, and Mach, plus a glossary and indexes.

Supplemental Documentation

Information about PostScript, RTF, and other file formats useful to application developers.

NeXTstep Reference
Volume 1

Manual written by Don Larkin, Matt Morse, Jim Inscore, Sam Streeper, and Jackie Neider
Edited by Caroline Rose, Kathy Walrath, Roy West, Helen Casabona, Adrienne Wong, and Jeremy Brest
Book design by Eddie Lee
Illustrations by Jeff Yaksick and Don Donoughe
Production by Adrienne Wong, Jennifer Yu, and Katherine Arthurs
Publications management by Cathy Novak

Reorder Product #N6007B

Contents

Introduction

Introduction

This manual describes the Application Programming Interface (API) for the NeXTstep® development environment. It's part of a collection of manuals called the *NeXT™ Developer's Library*; the illustration on the first page of this manual shows the complete set of manuals in this Library.

In two volumes, this manual provides detailed descriptions of all classes, functions, operators, and other programming elements that make up the API, listed alphabetically within each category for easy reference. Some topics discussed here aren't covered in detail; instead, you're referred to a generally available book on the subject, or to an on-line source of the information (see "Suggested Reading" in the *NeXT Technical Summaries* manual.

For many programmers, only a fraction of the information in this manual will have to be learned; the more sophisticated the application, the more you'll need to understand.

This manual assumes you're familiar with the standard NeXT user interface. Some experience using a NeXT application, such as the WriteNow® word processor, would be helpful.

A version of this manual is stored on-line in the NeXT Digital Library (which is described in the user's manual *NeXT Applications*). The Digital Library also contains Release Notes that provide last-minute information about the latest release of the software.

Using Documented API

The API described in this manual provides all the functionality you need to make full use of the NeXTstep software. If you have questions about using the API, this documentation and the NeXT Technical Support Department can help you use it correctly. If a feature in the API doesn't work as described, it's considered a bug which NeXT will work to fix. If API features change in future releases, these changes will be described in on-line release notes and printed documentation.

Undocumented features are not part of the API. If you use undocumented features, you run several risks. First, your application may be unreliable, because undocumented features won't work the way you expect them to in all cases. Second, NeXT Technical Support can't provide full assistance in fixing problems that arise, other than to recommend that you use documented API. Finally, your application may be incompatible with future releases, since undocumented features can and will change without notice.

How This Manual is Organized

The chapters in this manual are as follows:

- Chapter 1, "Constants and Data Types," lists constants and data types used by the methods, instance variables, and functions described in the remaining chapters. Not listed in this chapter are constants and data types specific to a particular class; these are documented with the associated class in Chapter 2.

- Chapter 2, "Class Specifications," describes the classes defined in the Application Kit as well as those that come with the NeXT implementation of the Objective-C language. Each class specification details the instance variables the class declares, the methods it defines, and any special constants and defined types it uses. There's also a general description of the class and its place in the inheritance hierarchy.

- Chapter 3, "C Functions," describes in detail the C functions provided by NeXT (except for Mach functions). It lists the functions in two groups, NeXTstep functions and run-time functions. Each function's calling sequence, its return value, and any exceptions it raises are given, in addition to a description of what the function does.

- Chapter 4, "PostScript® Operators," describes NeXT's extensions to the Display PostScript® system. It also lists the standard PostScript operators that have different or additional effects in the NeXT implementation.

- Chapter 5, "Data Formats," describes the standard data formats recognized by the pasteboard.

Volume 1 includes the introductory material, all of Chapter 1, and Chapter 2 through the OpenPanel class in the Application Kit. Volume 2 continues Chapter 2, beginning with the PageLayout class; it includes Chapters 3, 4, and 5 and the index.

Conventions

Syntax Notation

Where this manual shows the syntax of a method, function, or other programming element, the use of bold, italic, square brackets [ ], and ellipsis has special significance, as described here.

Bold denotes words or characters that are to be taken literally (typed as they appear). *Italic* denotes words that represent something else or can be varied. For example, the syntax

> **print** *expression*

means that you follow the word **print** with an expression.

Square brackets [ ] mean that the enclosed syntax is optional, except when they're bold **[ ]**, in which case they're to be taken literally. The exceptions are few and will be clear from the context. For example,

pointer [*filename*]

means that you type a pointer with or without a file name after it, but

[*receiver message***]**

means that you specify a receiver and a message enclosed in square brackets.

Ellipsis (...) indicates that the previous syntax element may be repeated. For example:

Syntax	**Allows**
pointer ...	One or more pointers
pointer [, *pointer*] ...	One or more pointers separated by commas
pointer [*filename* ...]	A pointer optionally followed by one or more file names
pointer [, *filename*] ...	A pointer optionally followed by a comma and one or more file names separated by commas

Chapter 1
Constants and Data Types

Chapter 1
Constants and Data Types

This chapter lists many of the constants and data types used in developing NeXTstep applications. This list includes constants and types defined in the **/usr/include** subdirectories **objc**, **dpsclient**, **appkit**, and **streams**. Not included are constants and types defined in the header files for the common classes and Application Kit classes: these are listed with the class descriptions in Chapter 2.

Constants and Data Types are presented in separate sections of this chapter. Each listing includes a reference to the class header file where the constant or type is defined.

Constants

In most cases, the value defined for a constant is arbitrary; you don't need to know the value to use the constant. In cases where a constant provides access to a meaningful value, the definition of that value is included in parentheses next to the constant's name.

Name	**Defined In**
CLS_CLASS	objc/objc-class.h
CLS_META	objc/objc-class.h
CLS_INITIALIZED	objc/objc-class.h
CLS_POSING	objc/objc-class.h
CLS_MAPPED	objc/objc-class.h
DPS_ALLCONTEXTS	dpsclient/dpsNeXT.h
DPS_ARRAY	dpsclient/dpsfriends.h
DPS_BOOL	dpsclient/dpsfriends.h
DPS_DEF_TOKENTYPE	dpsclient/dpsfriends.h
DPS_ERRORBASE	dpsclient/dpsclient.h
DPS_EXEC	dpsclient/dpsfriends.h
DPS_EXT_HEADER_SIZE	dpsclient/dpsfriends.h
DPS_HEADER_SIZE	dpsclient/dpsfriends.h
DPS_HI_IEEE	dpsclient/dpsfriends.h
DPS_HI_NATIVE	dpsclient/dpsfriends.h
DPS_IMMEDIATE	dpsclient/dpsfriends.h
DPS_INT	dpsclient/dpsfriends.h
DPS_LITERAL	dpsclient/dpsfriends.h
DPS_LO_IEEE	dpsclient/dpsfriends.h
DPS_LO_NATIVE	dpsclient/dpsfriends.h
DPS_MARK	dpsclient/dpsfriends.h
DPS_NAME	dpsclient/dpsfriends.h

DPS_NEXTERRORBASE	dpsclient/dpsclient.h
DPS_NULL	dpsclient/dpsfriends.h
DPS_REAL	dpsclient/dpsfriends.h
DPS_STRING	dpsclient/dpsfriends.h
DPSSYSNAME	dpsclient/dpsfriends.h
FALSE	appkit/nextstd.h
NBITSCHAR	appkit/nextstd.h
NBITSINT	appkit/nextstd.h
nil	objc/objc.h
Nil	objc/objc.h
NO	objc/objc.h
NX_ABOVE	dpsclient/dpsNeXT.h
NX_ALLEVENTS	dpsclient/event.h
NX_ALLOC_ERROR	appkit/tiff.h
NX_ALPHAMASK	appkit/graphics.h
NX_ALPHASHIFTMASK	dpsclient/event.h
NX_ALTERNATEMASK	dpsclient/event.h
NX_APPBASE	appkit/errors.h
NX_APPDEFINED	dpsclient/event.h
NX_APPDEFINEDMASK	dpsclient/event.h
NX_APPKITERRBASE	appkit/errors.h
NX_ASCIISET	dpsclient/event.h
NX_BAD_TIFF_FORMAT	appkit/tiff.h
NX_BELOW	dpsclient/dpsNeXT.h
NX_BIGENDIAN	appkit/tiff.h
NX_BLACK (0.0)	appkit/graphics.h
NX_BROADCAST	dpsclient/event.h
NX_BUFFERED	dpsclient/dpsNeXT.h
NX_BYPSCONTEXT	dpsclient/event.h
NX_BYTYPE	dpsclient/event.h
NX_CANREAD	streams/streams.h
NX_CANSEEK	streams/streams.h
NX_CANWRITE	streams/streams.h
NX_CLEAR	dpsclient/dpsNeXT.h
NX_COLORBLACK	appkit/color.h
NX_COLORBLUE	appkit/color.h
NX_COLORBROWN	appkit/color.h
NX_COLORCLEAR	appkit/color.h
NX_COLORCYAN	appkit/color.h
NX_COLORDKGRAY	appkit/color.h
NX_COLORGRAY	appkit/color.h
NX_COLORGREEN	appkit/color.h
NX_COLORLTGRAY	appkit/color.h
NX_COLORMAGENTA	appkit/color.h
NX_COLORMASK	appkit/graphics.h
NX_COLORORANGE	appkit/color.h
NX_COLORPURPLE	appkit/color.h
NX_COLORRED	appkit/color.h
NX_COLORWHITE	appkit/color.h
NX_COLORYELLOW	appkit/color.h

NX_COMMANDMASK	dpsclient/event.h
NX_COMPRESSION_NOT_YET_SUPPORTED	appkit/tiff.h
NX_CONTROLMASK	dpsclient/event.h
NX_COPY	dpsclient/dpsNeXT.h
NX_CURSORUPDATE	dpsclient/event.h
NX_CURSORUPDATEMASK	dpsclient/event.h
NX_DATA	dpsclient/dpsNeXT.h
NX_DATOP	dpsclient/dpsNeXT.h
NX_DEFAULTBUFSIZE (16 * 1024)	streams/streamsimpl.h
NX_DIN	dpsclient/dpsNeXT.h
NX_DINGBATSSET	dpsclient/event.h
NX_DKGRAY (1.0/3.0)	appkit/graphics.h
NX_DOUT	dpsclient/dpsNeXT.h
NX_DOVER	dpsclient/dpsNeXT.h
NX_EOS	streams/streams.h
NX_EVENTCODEMASK	dpsclient/event.h
NX_EXPLICIT	dpsclient/event.h
NX_FILE_IO_ERROR	appkit/tiff.h
NX_FIRSTEVENT	dpsclient/event.h
NX_FIRSTWINDOW	dpsclient/event.h
NX_FLAGSCHANGED	dpsclient/event.h
NX_FLAGSCHANGEDMASK	dpsclient/event.h
NX_FONTCHARDATA	appkit/afm.h
NX_FONTCOMPOSITES	appkit/afm.h
NX_FONTHEADER	appkit/afm.h
NX_FONTKERNING	appkit/afm.h
NX_FONTMETRICS	appkit/afm.h
NX_FONTWIDTHS	appkit/afm.h
NX_FOREVER	dpsclient/dpsNeXT.h
NX_FORMAT_NOT_YET_SUPPORTED	appkit/tiff.h
NX_FREEBUFFER	streams/streams.h
NX_FROMCURRENT	streams/streams.h
NX_FROMEND	streams/streams.h
NX_FROMSTART	streams/streams.h
NX_HIGHLIGHT	dpsclient/dpsNeXT.h
NX_IMAGE_NOT_FOUND	appkit/tiff.h
NX_JOURNALEVENT	dpsclient/event.h
NX_JOURNALEVENTMASK	dpsclient/event.h
NX_KEYDOWN	dpsclient/event.h
NX_KEYDOWNMASK	dpsclient/event.h
NX_KEYUP	dpsclient/event.h
NX_KEYUPMASK	dpsclient/event.h
NX_KITDEFINED	dpsclient/event.h
NX_KITDEFINEDMASK	dpsclient/event.h
NX_LASTEVENT	dpsclient/event.h
NX_LASTKEY	dpsclient/event.h
NX_LASTLEFT	dpsclient/event.h
NX_LASTRIGHT	dpsclient/event.h
NX_LITTLEENDIAN	appkit/tiff.h

NX_LMOUSEDOWN	dpsclient/event.h
NX_LMOUSEDOWNMASK	dpsclient/event.h
NX_LMOUSEDRAGGED	dpsclient/event.h
NX_LMOUSEDRAGGEDMASK	dpsclient/event.h
NX_LMOUSEUP	dpsclient/event.h
NX_LMOUSEUPMASK	dpsclient/event.h
NX_LTGRAY (2.0/3.0)	appkit/graphics.h
NX_MESHED	appkit/graphics.h
NX_MONOTONICMASK	appkit/graphics.h
NX_MOUSEDOWN	dpsclient/event.h
NX_MOUSEDOWNMASK	dpsclient/event.h
NX_MOUSEDRAGGED	dpsclient/event.h
NX_MOUSEDRAGGEDMASK	dpsclient/event.h
NX_MOUSEENTERED	dpsclient/event.h
NX_MOUSEENTEREDMASK	dpsclient/event.h
NX_MOUSEEXITED	dpsclient/event.h
NX_MOUSEEXITEDMASK	dpsclient/event.h
NX_MOUSEMOVED	dpsclient/event.h
NX_MOUSEMOVEDMASK	dpsclient/event.h
NX_MOUSEUP	dpsclient/event.h
NX_MOUSEUPMASK	dpsclient/event.h
NX_MOUSEWINDOW	dpsclient/event.h
NX_NEXTCTRLKEYMASK	dpsclient/event.h
NX_NEXTLALTKEYMASK	dpsclient/event.h
NX_NEXTLCMDKEYMASK	dpsclient/event.h
NX_NEXTLSHIFTKEYMASK	dpsclient/event.h
NX_NEXTRALTKEYMASK	dpsclient/event.h
NX_NEXTRCMDKEYMASK	dpsclient/event.h
NX_NEXTRSHIFTKEYMASK	dpsclient/event.h
NX_NEXTWINDOW	dpsclient/event.h
NX_NOALPHA	appkit/color.h
NX_NOBUF	streams/streams.h
NX_NONRETAINED	dpsclient/dpsNeXT.h
NX_NOWINDOW	dpsclient/event.h
NX_NULLEVENT	dpsclient/event.h
NX_NULLEVENTMASK	dpsclient/event.h
NX_NUMERICPADMASK	dpsclient/event.h
NX_ONES	dpsclient/dpsNeXT.h
NX_OUT	dpsclient/dpsNeXT.h
NX_PAGEHEIGHT	appkit/tiff.h
NX_PLANAR	appkit/graphics.h
NX_PLUS	dpsclient/dpsNeXT.h
NX_PLUSD	dpsclient/dpsNeXT.h
NX_PLUSL	dpsclient/dpsNeXT.h
NX_READFLAG	streams/streams.h
NX_READONLY	streams/streams.h
NX_READWRITE	streams/streams.h
NX_RETAINED	dpsclient/dpsNeXT.h
NX_RMOUSEDOWN	dpsclient/event.h
NX_RMOUSEDOWNMASK	dpsclient/event.h

NX_RMOUSEDRAGGED	dpsclient/event.h
NX_RMOUSEDRAGGEDMASK	dpsclient/event.h
NX_RMOUSEUP	dpsclient/event.h
NX_RMOUSEUPMASK	dpsclient/event.h
NX_SATOP	dpsclient/dpsNeXT.h
NX_SAVEBUFFER	streams/streams.h
NX_SHIFTMASK	dpsclient/event.h
NX_SIN	dpsclient/dpsNeXT.h
NX_SOUT	dpsclient/dpsNeXT.h
NX_SOVER	dpsclient/dpsNeXT.h
NX_STREAMERRBASE	streams/streams.h
NX_SYMBOLSET	dpsclient/event.h
NX_SYSDEFINED	dpsclient/event.h
NX_SYSDEFINEDMASK	dpsclient/event.h
NX_TIFF_CANT_APPEND	appkit/tiff.h
NX_TIFF_COMPRESSION_CCITFAX3	appkit/tiff.h
NX_TIFF_COMPRESSION_JPEG	appkit/tiff.h
NX_TIFF_COMPRESSION_LZW	appkit/tiff.h
NX_TIFF_COMPRESSION_NEXT	appkit/tiff.h
NX_TIFF_COMPRESSION_NONE	appkit/tiff.h
NX_TIFF_COMPRESSION_PACKBITS	appkit/tiff.h
NX_TIMER	dpsclient/event.h
NX_TIMERMASK	dpsclient/event.h
NX_TOPWINDOW	dpsclient/event.h
NX_TRANSMIT	dpsclient/event.h
NX_TRUNCATEBUFFER	streams/streams.h
NX_UNIQUEALPHABITMAP	appkit/obsoleteBitmap.h
NX_UNIQUEBITMAP	appkit/obsoleteBitmap.h
NX_USER_OWNS_BUF	streams/streams.h
NX_WHITE (1.0)	appkit/graphics.h
NX_WRITEFLAG	streams/streams.h
NX_WRITEONLY	streams/streams.h
NX_XMAX	appkit/graphics.h
NX_XMIN	appkit/graphics.h
NX_XOR	dpsclient/dpsNeXT.h
NX_YMAX	appkit/graphics.h
NX_YMIN	appkit/graphics.h
NXSYSTEMVERSION	objc/typedstream.h
NXSYSTEMVERSION082	objc/typedstream.h
NXSYSTEMVERSION083	objc/typedstream.h
NXSYSTEMVERSION090	objc/typedstream.h
NXSYSTEMVERSION0900	objc/typedstream.h
NXSYSTEMVERSION0901	objc/typedstream.h
NXSYSTEMVERSION0905	objc/typedstream.h
NXSYSTEMVERSION0930	objc/typedstream.h
TRUE	appkit/nextstd.h
TYPEDSTREAM_ERROR_RBASE	objc/typedstream.h
YES	objc/objc.h

Data Types

BOOL

DEFINED IN	objc/objc.h

```
typedef char  BOOL;
```

Cache

DEFINED IN	objc/objc-class.h

```
typedef struct objc_cache *Cache;
```

Category

DEFINED IN	objc/objc-class.h

```
typedef struct objc_category *Category;
```

Class

DEFINED IN	objc/objc.h

```
typedef struct objc_class *Class;
```

DPSBinObjRec

DEFINED IN	dpsclient/dpsfriends.h

```
typedef struct {
    unsigned char attributedType;
    unsigned char tag;
    unsigned short length;
    union {
        long int integerVal;
        float realVal;
        long int nameVal;    /* offset or index */
        long int booleanVal;
        long int stringVal;  /* offset */
        long int arrayVal;   /* offset */
    } val;
} DPSBinObjRec, *DPSBinObj;
```

DPSBinObjGeneric

DEFINED IN dpsclient/dpsfriends.h

```
typedef struct {
    unsigned char attributedType;
    unsigned char tag;
    unsigned short length;
    long int val;
} DPSBinObjGeneric;
```

DPSBinObjReal

DEFINED IN dpsclient/dpsfriends.h

```
typedef struct {
    unsigned char attributedType;
    unsigned char tag;
    unsigned short length;
    float realVal;
} DPSBinObjReal;
```

DPSBinObjSeqRec

DEFINED IN dpsclient/dpsfriends.h

```
typedef struct {
    unsigned char tokenType;
    unsigned char nTopElements;
    unsigned short length;
    DPSBinObjRec objects[1];
} DPSBinObjSeqRec, *DPSBinObjSeq;
```

DPSContextRec

DEFINED IN dpsclient/dpsfriends.h

```
typedef struct _t_DPSContextRec {
  char *priv;
  DPSSpace space;
  DPSProgramEncoding programEncoding;
  DPSNameEncoding nameEncoding;
  struct _t_DPSProcsRec const * procs;
  void (*textProc)();
  void (*errorProc)();
  DPSResults resultTable;
  unsigned int resultTableLength;
  struct _t_DPSContextRec *chainParent, *chainChild;
  DPSContextType type;  /* NeXT addition - denotes type of context */
  } DPSContextRec, *DPSContext;
```

DPSContextType

DEFINED IN dpsclient/dpsfriends.h

```
typedef enum {      /* NeXT addition */
    dps_machServer,/* a mach binary connection to a window server */
    dps_fdServer,  /* a socket binary connection to a window server */
    dps_stream     /* an ascii NXStream */
    } DPSContextType;
```

DPSDefinedType

DEFINED IN dpsclient/dpsfriends.h

```
typedef enum {
    dps_tBoolean,
    dps_tChar,   dps_tUChar,
    dps_tFloat,  dps_tDouble,
    dps_tShort,  dps_tUShort,
    dps_tInt,    dps_tUInt,
    dps_tLong,   dps_tULong } DPSDefinedType;
```

DPSErrorCode

DEFINED IN dpsclient/dpsclient.h

```
typedef enum _DPSErrorCode {
    dps_err_ps = DPS_ERRORBASE,
    dps_err_nameTooLong,
    dps_err_resultTagCheck,
    dps_err_resultTypeCheck,
    dps_err_invalidContext,
    dps_err_select = DPS_NEXTERRORBASE,
    dps_err_connectionClosed,
    dps_err_read,
    dps_err_write,
    dps_err_invalidFD,
    dps_err_invalidTE,
    dps_err_invalidPort,
    dps_err_outOfMemory,
    dps_err_cantConnect
} DPSErrorCode;
```

DPSErrorProc

DEFINED IN dpsclient/dpsclient.h

```
typedef void (*DPSErrorProc)(
    DPSContext ctxt,
    DPSErrorCode errorCode,
    long unsigned int arg1,
    long unsigned int arg2 );
```

DPSEventFilterFunc

DEFINED IN dpsclient/dpsNeXT.h

```
typedef int (*DPSEventFilterFunc)( NXEvent *ev );
```

DPSExtendedBinObjSeq

DEFINED IN dpsclient/dpsfriends.h

```
typedef struct {
    unsigned char tokenType;
    unsigned char escape;  /* zero if this is an extended sequence */
    unsigned short nTopElements;
    unsigned long length;
    DPSBinObjRec objects[1];
} DPSExtendedBinObjSeqRec, *DPSExtendedBinObjSeq;
```

DPSFDProc

DEFINED IN dpsclient/dpsNeXT.h:

```
typedef void (*DPSFDProc)( int fd, void *userData );
```

DPSNameEncoding

DEFINED IN dpsclient/dpsfriends.h

```
typedef enum {
    dps_indexed,
    dps_strings
    } DPSNameEncoding;
```

DPSNumberFormat

DEFINED IN dpsclient/dpsNeXT.h

```
typedef enum _DPSNumberFormat {
    dps_float = 48,
    dps_long = 0,
    dps_short = 32
} DPSNumberFormat;
```

DPSPortProc

DEFINED IN dpsclient/dpsNeXT.h

```
typedef void (*DPSPortProc)( msg_header_t *msg, void *userData );
```

DPSProcs

DEFINED IN dpsclient/dpsfriends.h

```
typedef struct _t_DPSProcsRec {
    void (*BinObjSeqWrite)(
            DPSContext ctxt,
            const void *buf,
            unsigned int count );
    void (*WriteTypedObjectArray)(
            DPSContext ctxt,
            DPSDefinedType type,
            const void *array,
            unsigned int length );
```

```
        void (*WriteStringChars)(
                DPSContext ctxt,
                const char *buf,
                unsigned int count );
        void (*WriteData)(
                DPSContext ctxt,
                const void *buf,
                unsigned int count );
        void (*WritePostScript)(
                DPSContext ctxt,
                const void *buf,
                unsigned int count );
        void (*FlushContext)( DPSContext ctxt );
        void (*ResetContext)( DPSContext ctxt );
        void (*UpdateNameMap)( DPSContext ctxt );
        void (*AwaitReturnValues)( DPSContext ctxt );
        void (*Interrupt)( DPSContext ctxt );
        void (*DestroyContext)( DPSContext ctxt );
        void (*WaitContext)( DPSContext ctxt );
        void (*Printf)(
                DPSContext ctxt,
                const char *fmt,
                va_list argList );
        } DPSProcsRec, *DPSProcs;
```

DPSProgramEncoding

DEFINED IN dpsclient/dpsfriends.h

```
    typedef enum {
        dps_ascii,
        dps_binObjSeq,
        dps_encodedTokens
        } DPSProgramEncoding;
```

DPSResultsRec

DEFINED IN dpsclient/dpsfriends.h

```
    typedef struct {
        DPSDefinedType type;
        int count;
        char *value;
        } DPSResultsRec, *DPSResults;
```

DPSSpaceRec

DEFINED IN dpsclient/dpsfriends.h

```
typedef struct {
    int lastNameIndex;
    struct _t_DPSSpaceProcsRec const * procs;
    } DPSSpaceRec, *DPSSpace;
```

DPSSpaceProcsRec

DEFINED IN dpsclient/dpsfriends.h

```
typedef struct _t_DPSSpaceProcsRec {
    void (*DestroySpace)( DPSSpace space );
       /* See DPSDestroySpace() in dpsclient.h */
} DPSSpaceProcsRec, *DPSSpaceProcs;
```

DPSTextProc

DEFINED IN dpsclient/dpsclient.h

```
typedef void (*DPSTextProc)(
    DPSContext ctxt,
    const char *buf,
    long unsigned int count );
```

DPSTimedEntry

DEFINED IN dpsclient/dpsNeXT.h

```
typedef struct __DPSTimedEntry *DPSTimedEntry;
```

DPSUserPathAction

DEFINED IN dpsclient/dpsNeXT.h

```
typedef enum _DPSUserPathAction {
    dps_uappend = 176,
    dps_ufill = 179,
    dps_ueofill = 178,
    dps_ustroke = 183,
    dps_ustrokepath = 364,
    dps_inufill = 93,
    dps_inueofill = 92,
    dps_inustroke = 312,
    dps_def = 51,
    dps_put = 120
} DPSUserPathAction;
```

DPSUserPathOp

DEFINED IN dpsclient/dpsNeXT.h

```
typedef enum _DPSUserPathOp {
    dps_setbbox = 0,
    dps_moveto,
    dps_rmoveto,
    dps_lineto,
    dps_rlineto,
    dps_curveto,
    dps_rcurveto,
    dps_arc,
    dps_arcn,
    dps_arct,
    dps_closepath,
    dps_ucache
} DPSUserPathOp;
```

id

DEFINED IN objc/objc.h

```
typedef struct objc_object {
    Class isa;
} *id;
```

IMP

DEFINED IN objc/objc.h

```
typedef id  (*IMP)(id, SEL, ...);
```

Ivar

DEFINED IN objc/objc-class.h

```
typedef struct objc_ivar *Ivar;
```

Method

DEFINED IN objc/objc-class.h

```
typedef struct objc_method *Method;
```

Module

DEFINED IN objc/objc-runtime.h

```
typedef struct objc_module *Module;
```

NXAppkitErrorTokens

DEFINED IN appkit/errors.h

```
typedef enum _NXAppkitErrorTokens {
    NX_longLine  = NX_APPKITERRBASE,
    NX_nullSel,         /* Text, operation attempted on empty
                           selection */
    NX_wordTablesWrite, /* error while writing word tables */
    NX_wordTablesRead,  /* error while reading word tables */
    NX_textBadRead,     /* Text, error reading from file */
    NX_textBadWrite,    /* Text, error writing to file */
    NX_powerOff,        /* poweroff */
    NX_pasteboardComm,  /* communications prob with pbs server */
    NX_mallocError,     /* malloc problem */
    NX_printingComm,    /* sending to npd problem */
    NX_abortModal,      /* used to abort modal panels */
    NX_abortPrinting,   /* used to abort printing */
    NX_illegalSelector, /* bogus selector passed to appkit */
    NX_appkitVMError,   /* error from vm_ call */
    NX_badRtfDirective,
    NX_badRtfFontTable,
    NX_badRtfStyleSheet,
    NX_newerTypedStream,
    NX_tiffError
} NXAppkitErrorTokens;
```

NXAtom

DEFINED IN objc/hashtable.h

```
typedef const char *NXAtom;
```

NXCharMetrics

DEFINED IN appkit/afm.h

```
typedef struct {  /* per character info */
    short charCode;
    unsigned char numKernPairs;
    unsigned char reserved;
    float xWidth;
    int name;
    float bbox[4];
    int kernPairIndex;
} NXCharMetrics;
```

NXChunk

DEFINED IN appkit/chunk.h

```
typedef struct _NXChunk {
    short  growby;      /* increment to grow by */
    int    allocated;   /* how much is allocated */
    int    used;        /* how much is used */
} NXChunk;
```

NXColor

DEFINED IN appkit/color.h

```
typedef struct _NXColor {
    unsigned short colorField[8];
} NXColor;
```

NXColorSpace

DEFINED IN appkit/graphics.h

```
typedef enum _NXColorSpaceType {
    NX_ONEISBLACK_COLORSPACE = 0,    /* monochrome, 1 is black */
    NX_ONEISWHITE_COLORSPACE = 1,    /* monochrome, 1 is white */
    NX_RGB_COLORSPACE = 2,
    NX_CMYK_COLORSPACE = 5
} NXColorSpace;
```

NXCompositeChar

DEFINED IN appkit/afm.h

```
typedef struct {    /* a composite char */
    int numParts;
    int firstPartIndex;
} NXCompositeChar;
```

NXCompositeCharPart

DEFINED IN appkit/afm.h

```
typedef struct {    /* elements of the composite char array */
    int partIndex;
    float dx;
    float dy;
} NXCompositeCharPart;
```

NXCoord

DEFINED IN dpsclient/event.h

```
typedef float  NXCoord
```

NXDefaultsVector

DEFINED IN appkit/defaults.h

```
typedef struct _NXDefault {
    char *name;
    char *value;
} NXDefaultsVector[];
```

NXEncodedLigature

DEFINED IN appkit/afm.h

```
typedef struct {  /* elements of the encoded ligature array */
    unsigned char firstChar;
    unsigned char secondChar;
    unsigned char ligatureChar;
    unsigned char reserved;
} NXEncodedLigature;
```

NXErrorReporter

DEFINED IN appkit/errors.h

```
typedef void NXErrorReporter(NXHandler *errorState);
```

NXEvent

DEFINED IN dpsclient/event.h

```
typedef struct _NXEvent {
    int type;              /* An event type from above */
    NXPoint location;
        /* Base coordinates in window, from lower-left */
    long time              /* vertical intervals since launch */
    int flags;             /* key state flags */
    unsigned int window;   /* window number of assigned window */
    NXEventData data;      /* type-dependent data */
    DPSContext ctxt;       /* context the event came from */
} NXEvent, *NXEventPtr;
```

NXEventData

DEFINED IN dpsclient/event.h

```
typedef union {
    struct {              /* For mouse-down and mouse-up events */
        short reserved;
        short eventNum;   /* unique identifier for this button */
        int   click;      /* click state of this event */
        int   unused;
    } mouse;
    struct {              /* For key-down and key-up events */
        short  reserved;
        short  repeat; /* for key-down: nonzero if really a repeat */
        unsigned short charSet;    /* character set code */
        unsigned short charCode;   /* character code in that set */
        unsigned short keyCode;    /* device-dependent key number */
        short    keyData;          /* device-dependent info */
    } key;
    struct {        /* For mouse-entered and mouse-exited events */
        short    reserved;
        short    eventNum;
                   /* unique identifier from mouse down event */
        int      trackingNum;  /* unique identifier from
                                  settrackingrect */
        int      userData;     /* uninterpreted integer from
                                  settrackingrect */
    } tracking;
    struct {    /* For appkit-defined, sys-defined, and app-defined
                   events */
        short   reserved;
        short   subtype;   /* event subtype for compound events */
        union {
            float   F[2];  /* for use in compound events */
            long    L[2];  /* for use in compound events */
            short   S[4];  /* for use in compound events */
            char    C[8];  /* for use in compound events */
        } misc;
    } compound;
} NXEventData;
```

NXExceptionRaiser

DEFINED IN objc/error.h

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

NXFontMetrics

DEFINED IN appkit/afm.h

```
typedef struct _NXFontMetrics {
    char *formatVersion;        /* version of afm file format */
    char *name;                 /* name of font for findfont */
    char *fullName;             /* full name of font */
    char *familyName;           /* "font family" name */
    char *weight;               /* weight of font */
    float italicAngle;          /* degrees ccw from vertical */
    char  isFixedPitch;         /* is the font mono-spaced? */
    char  isScreenFont;         /* is the font a screen font? */
    short screenFontSize;       /* If it is, how big is it? */
    float fontBBox[4];          /* bounding box (llx lly urx ury) */
    float underlinePosition;    /* dist from basline for underlines */
    float underlineThickness;   /* thickness of underline stroke */
    char *version;              /* version identifier */
    char *notice;               /* trademark or copyright */
    char *encodingScheme;       /* default encoding vector */
    float capHeight;            /* top of 'H' */
    float xHeight;              /* top of 'x' */
    float ascender;             /* top of 'd' */
    float descender;            /* bottom of 'p' */
    short hasYWidths;           /* do any chars have non-0 y width? */
    float *widths;              /* character widths in x */
    unsigned int widthsLength;
    char  *strings;             /* table of strings and other info */
    unsigned int stringsLength;
    char hasXYKerns;  /* Do any of the kern pairs have nonzero dy? */
    char reserved;
    short *encoding;            /* 256 offsets into charMetrics */
    float *yWidths;
        /* character widths in y.  NOT in encoding */
        /* order, but a parallel array to the charMetrics array */
    NXCharMetrics      *charMetrics;     /* array of NXCharMetrics */
    int                 numCharMetrics;  /* num elements */
    NXLigature         *ligatures;       /* array of NXLigatures */
    int                 numLigatures;    /* num elements */
    NXEncodedLigature  *encLigatures;    /* array of
                                            NXEncodedLigatures */
    int                 numEncLigatures; /* num elements */
    union {
        NXKernPair      *kernPairs;        /* array of NXKernPairs */
        NXKernXPair     *kernXPairs;       /* array of NXKernXPairs */
    } kerns;
    int                  numKernPairs;     /* num elements */
    NXTrackKern         *trackKerns;       /* array of NXTrackKerns */
    int                  numTrackKerns;    /* num elements */
    NXCompositeChar     *compositeChars;   /* array of
                                              NXCompositeChar */
    int                  numCompositeChars;  /* num elements */
    NXCompositeCharPart *compositeCharParts; /* array of
                                                NXCompositeCharPart */
    int numCompositeCharParts;               /* num elements */
} NXFontMetrics;
```

NXHandler

DEFINED IN objc/error.h

```
typedef struct _NXHandler {         /* a node in the handler chain */
    jmp_buf             jumpState;  /* place to longjmp to */
    struct _NXHandler *next;        /* ptr to next handler */
    int                 code;       /* error code of exception */
    const void *data1, *data2;      /* blind data for describing */
} NXHandler;                        /* error */
```

NXHashState

DEFINED IN objc/hashtable.h

```
typedef struct {int i; int j;} NXHashState;
```

NXHashTablePrototype

DEFINED IN objc/hashtable.h

```
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; /* reserved for future expansion; currently 0 */
    } NXHashTablePrototype;
```

NXImageInfo

DEFINED IN appkit/tiff.h

```
typedef struct _NXImageInfo {
    int width;              /* image width in pixels */
    int height;             /* image height in pixels */
    int bitsPerSample;      /* number of bits per data channel */
    int samplesPerPixel;    /* number of channels per pixel */
    int planarConfig;       /* NX_MESHED for mixed data channels */
    /* NX_PLANAR for separate data planes */
    int photoInterp;     /* various bits set for various photometric */
    /* interpretations, as in the table below */
} NXImageInfo;
```

NXKernPair

DEFINED IN appkit/afm.h

```
typedef struct {    /* elements of the kern pair array */
    int secondCharIndex;
    float dx;
    float dy;
} NXKernPair;
```

NXKernXPair

DEFINED IN appkit/afm.h

```
typedef struct {    /* elements of the kern X pair array */
    int secondCharIndex;
    float dx;
} NXKernXPair;
```

NXLigature

DEFINED IN appkit/afm.h

```
typedef struct {    /* elements of the ligature array */
    int firstCharIndex;
    int secondCharIndex;
    int ligatureIndex;
} NXLigature;
```

NXPoint

DEFINED IN dpsclient/event.h

```
typedef struct _NXPoint {    /* point */
    NXCoord  x, y;
} NXPoint;
```

NXPrintfProc

DEFINED IN streams/streams.h

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

NXRect

DEFINED IN appkit/graphics.h

```
typedef struct _NXRect {
    NXPoint  origin;
    NXSize   size;
} NXRect;
```

NXScreen

DEFINED IN appkit/screens.h

```
typedef struct _NXScreen {
    int           screenNumber;     /* Screen number (may be used as */
                                    /* argument to framebuffer op). */
    NXRect        screenBounds;     /* Bounds of the screen. */
    short         _reservedShort[6];  /* Don't use these. */
    NXWindowDepth depth;            /* Depth of the frame buffer */
    int           _reserved[3];     /* Don't use these either. */
} NXScreen;
```

NXSize

DEFINED IN dpsclient/event.h

```
typedef struct _NXSize {   /* size */
    NXCoord  width, height;
} NXSize;
```

NXStream

DEFINED IN streams/streams.h

```
typedef struct _NXStream {
    unsigned int    magic_number;/* to check stream validity */
    unsigned char  *buf_base;    /* data buffer */
    unsigned char  *buf_ptr;     /* current buffer pointer */
    int             buf_size;    /* size of buffer */
    int             buf_left;    /* # left till buffer operation */
    long            offset;      /* position of beginning of buffer */
    int             flags;       /* info about stream */
    int             eof;
    const struct stream_functions  *functions; /* functions to
                                                  implement stream */
    void           *info;        /* stream specific info */
} NXStream;
```

NXStreamErrors

DEFINED IN streams/streams.h

```
typedef enum _NXStreamErrors {
    NX_illegalWrite = NX_STREAMERRBASE,
    NX_illegalRead,
    NX_illegalSeek,
    NX_illegalStream,
    NX_streamVMError
} NXStreamErrors;
```

NXTIFFInfo

DEFINED IN appkit/tiff.h

```
typedef struct _NXTIFFInfo {
    int imageNumber;
    NXImageInfo image;
    int  subfileType;     /* only subfileType = 1 is supported */
    int  rowsPerStrip;
    int  stripsPerImage;
    int  compression;     /* compression id, 1 = no compression */
    int  numImages;       /* number of images in tiff */
    int  endian;          /* either NX_BIGENDIAN or NX_LITTLEENDIAN */
    int  version;         /* tiff version */
    int  error;
    int  firstIFD;        /* offset of first IFD entry */
    unsigned int  stripOffsets[NX_PAGEHEIGHT];
    unsigned int  stripByteCounts[NX_PAGEHEIGHT];
} NXTIFFInfo;
```

NXTopLevelErrorHandler

DEFINED IN appkit/errors.h

```
typedef void NXTopLevelErrorHandler(NXHandler *errorState);
```

NXTrackingTimer

DEFINED IN appkit/timer.h

```
typedef struct _NXTrackingTimer {
    double delay;
    double period;
    DPSTimedEntry te;
    BOOL freeMe;
    BOOL firstTime;
    NXHandler *errorData;
    int reserved1;
    int reserved2;
} NXTrackingTimer;
```

NXTrackKern

DEFINED IN appkit/afm.h

```
typedef struct {  /* elements of the track kern array */
    int degree;
    float minPointSize;
    float minKernAmount;
    float maxPointSize;
    float maxKernAmount;
} NXTrackKern;
```

NXTypedStream

DEFINED IN objc/typedstream.h

```
typedef void NXTypedStream;
```

NXUncaughtExceptionHandler

DEFINED IN objc/error.h

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

SEL

DEFINED IN objc/objc.h

```
typedef struct objc_selector  *SEL;
```

STR

DEFINED IN objc/objc.h

```
typedef char *STR;
```

Symtab

DEFINED IN objc/objc-runtime.h

```
typedef struct objc_symtab *Symtab;
```

TypedstreamErrors

DEFINED IN objc/typedstream.h

```
enum TypedstreamErrors {
    TYPEDSTREAM_CALLER_ERROR = TYPEDSTREAM_ERROR_RBASE,
    TYPEDSTREAM_FILE_INCONSISTENCY,
    TYPEDSTREAM_CLASS_ERROR,
    TYPEDSTREAM_TYPE_DESCRIPTOR_ERROR,
    TYPEDSTREAM_WRITE_REFERENCE_ERROR,
    TYPEDSTREAM_INTERNAL_ERROR
};
```

Chapter 2
Class Specifications

Chapter 2
Class Specifications

This chapter describes each of the classes defined in the Application Kit, as well as the classes that come with the NeXT compiler for the Objective-C language. The classes that come with the compiler can be used with any kit (and in programs that don't use the kits).

Each class specification details the instance variables the class declares, the methods it defines, and any special constants and defined types it uses. There's also a general description of the class and its place in the inheritance hierarchy. However, you won't find a discussion of any kit's design or an explanation of how to go about using the kit to program an application. You may occasionally encounter terms that assume some prior knowledge about the kits, Mach, the Display PostScript system, or object-oriented programming. These topics are covered in other volumes of the *NeXT Developer's Library.*

How to Read the Specifications

The class specifications are organized in two groups: common classes and Application Kit classes. Within each of these groups, the specifications are arranged in alphabetical order by class.

Organization

Information about a class is presented under the following headings:

INHERITS FROM

The first line of a class specification lists the classes that the class being described inherits from. For example:

Panel : Window : Responder : Object

The first class listed (Panel, in this example) is the class's superclass. The last class listed is always Object, the root of all inheritance hierarchies. The classes between show the chain of inheritance from Object to the superclass. (This particular example shows the inheritance hierarchy for the Menu class of the Application Kit.)

DECLARED IN

Each class lists the directory and header file in which its interface is declared.

In the Application Kit, a master header file includes almost all the other header files you need to program with the kit:

/usr/include/appkit/appkit.h

There's also a master header file for the classes that come with the compiler:

/usr/include/objc/objc.h

If you include the master header file for the Application Kit, you don't need to also include this file; it's included by the kit file.

Because the kits are written in the Objective-C language, they make use of constants and types defined in the principal header file for Objective-C, **objc.h**. Only a handful of these constants and types are used by the kits, but they're used pervasively. For convenience, they're listed below.

Defined Types:

id	An object.
STR	A C string. STR is a shorthand for (**char ***). It's used only for an array of characters that's terminated by the null character.
SEL	A method selector. SEL is another shorthand for (**char ***), where the character string can be thought of as a method name. However, SEL is used only as a unique code for a method name, rather than as a pointer to an actual occurrence of the name in memory. Values should be assigned to SEL variables only with the **@selector** operator: `SEL  aMethod;` `aMethod = @selector(moveTo::);` This allows selectors to be tested by matching the value of a SEL code, rather than by comparing all the characters in a string.
BOOL	A **char** that holds one of two values: YES (true) or NO (false).

Constants:

nil	A null object **id**, (id)0.
YES	Boolean true, (BOOL)1.
NO	Boolean false, (BOOL)0.

CLASS DESCRIPTION

This section gives a general description of the class. It tells how the class fits into the general design of its kit and how your application can make use of it.

- Some classes define "off-the-shelf" objects: Your program can create direct instances of the class, or modify it in a subclass definition.

- Other classes are "abstract superclasses": You wouldn't create an instance of the class itself, but only of its subclasses. The kits define some subclasses for each abstract superclass; others can be defined by your application.

Occasionally, the class description will recommend that you define a subclass of a kit class, even though the kit class isn't abstract. The subclass allows you to customize an object to the needs of your application.

INSTANCE VARIABLES

The instance variables that are incorporated into each object belonging to the class, including instance variables inherited from other classes, are listed next. The first instance variable in all the lists is one inherited from the Object class, **isa**. **isa** identifies the class that an object belongs to for the run-time system; it should never be altered or read directly.

After all the instance variables are listed, those declared in the class being described are explained.

However, instance variables that are for the internal use of the class are neither listed nor explained. These instance variables all begin with an underscore (_) to prevent collisions with names that you might choose for instance variables in a subclass you define.

METHOD TYPES

Methods are next listed by name and grouped by type—for example, methods used to draw are listed separately from methods used to handle events. This directory includes all the principal methods defined in the class and some that are defined in classes it inherits from. Inherited methods are followed by the name of the class where they're defined; they're included in the directory to let you know which inherited methods you might commonly use with instances of the class and where to look for a description of those methods.

CLASS METHODS
INSTANCE METHODS

A detailed description of each method defined in the class follows the classification by type. Methods that are used by class objects are presented first; if a class has no class methods, this section is left out. Methods that are used by instances (the objects produced by the class) are presented next. The descriptions within each group are ordered alphabetically by method name.

Each description begins with the syntax of the method's arguments and return values, continues with an explanation of the method, and ends, where appropriate, with a list of other related methods. Where a related method is defined in another class, it's followed by the name of the other class within parentheses.

Some methods listed in a class specification are prototypes for methods that you may want to implement in a subclass. A prototype is declared in the header file, but not actually implemented by the class. The description for such methods states that they are prototypes and describes the behavior and return value you should implement for the method.

All methods except prototypes have reliable return values which are included in the method description. Many methods return **self**; this allows you to chain messages together:

[[[*receiver message1*] *message2*] *message3*];

Internal methods used to implement the class aren't listed. Since you shouldn't override any of these methods, or use them in a message, they're excluded from both the method directory and the method descriptions. However, you may encounter them when looking at the call stack of your program from within the debugger. A private method is easily recognizable by the underscore (_) that begins its name.

METHODS IMPLEMENTED BY ANOTHER OBJECT

If a class lets you define another object—a delegate—that can intercede on behalf of instances of the class, the methods that the delegate can implement are described in a separate section. These are not methods defined in the class; rather, they're methods that you can define to respond to messages sent to the delegate.

If you define one of these methods, the delegate will receive automatic messages to perform it at the appropriate time. For example, if you define a **windowDidBecomeKey:** method for a Window's delegate, the delegate will receive **windowDidBecomeKey:** messages whenever the Window becomes the key window.

Messages are sent only if the delegate has a method that can respond. If you don't define a **windowDidBecomeKey:** method, no message will be sent.

Only certain classes provide for a delegate. In the Application Kit, they are:

Application
Listener
NXBrowser
Speaker
Text
Window

You can set a delegate for instances of these classes or for instances that inherit from these classes.

Some class specifications have separate sections with titles such as "Methods Implemented by the SuperView" or "Methods Implemented by the Owner." The methods described in these sections need to be implemented by another object, such as the superview of an instance of that class or, in the case of the Pasteboard, the owner of the Pasteboard instance. For example, the ClipView's superview needs to define the **scrollClip:to:** method to coordinate scrolling of multiple ClipViews. The owner of the Pasteboard should define **provideData:** if certain promised data types won't be immediately written to the Pasteboard. As is the case with the delegate methods, you won't invoke these methods directly; messages to perform them will be sent automatically when needed and only if they've been defined.

CONSTANTS AND DEFINED TYPES

If a class makes use of symbolic constants or defined types that are specific to the class, they're listed in the last section of the class specification. Defined types are likely to show up in instance variable declarations, and as return and parameter types in method declarations. Symbolic constants typically define permitted return and argument values.

Method Descriptions

By far, the major portion of each class specification is the description of methods defined in the class. When reading these descriptions, be especially attentive to four kinds of information that affect how the method can be used:

- Whether you should implement your own version of the method
- Whether you should have your version of the method include the kit-defined version
- Which method is a class's designated initializer, the method to override if you implement a subclass that performs initialization
- Whether you should ever send a message to an object to perform the method

The next four sections examine these questions.

Implementing Your Own Version of a Method

For the most part, the methods in a class definition act as a private library for objects belonging to that class. Just as programmers generally don't replace functions in the standard C library with their own versions, you generally wouldn't write your own versions of the methods provided for a class.

However, to add specific behavior to your application, you must override some of the methods that are defined in the kits. Often, the kit-defined method will do little or nothing that's of use to your application, but it will appear in messages initiated by other methods. To give content to the method, your application must implement its own version.

To override a kit method with one of your own design, simply define a subclass of the appropriate class and redefine the method. For example, the interface declaration for the CircleView class illustrated below shows that it does nothing more than override the View class's **drawSelf::** method.

```
@interface CircleView : View { }
- drawSelf:(NWRect *)drawRects :(int)rectCount;
@end
```

CircleView objects will perform its version of **drawSelf::** rather than the empty default version defined in View.

In contrast to methods that must be overridden, some methods should never be changed by the application. The kit depends on these methods doing just what they're currently programmed to do—nothing more and nothing less. While your application can use these methods, it's important that you don't override them when defining a subclass.

Most methods fit between these two extremes: They can be overridden, but it's not necessary for you to do so. If a method description is silent on the question of overriding the kit method, you can be certain that it fits into this middle category. It's a method that you can override, but like a function in the C library, you normally would have no reason to.

If a method is designed to be overridden, or if it should never be overridden, the method description explicitly says so.

Retaining the Kit's Version of a Method

Some methods can be overridden, but only to add behavior, not to alter the default actions of the kit-defined method. When your application overrides one of these methods, it's important that it incorporate the very method it overrides. This is done by messaging **super** to perform the kit-defined version of the method. For example, if you write a new version of the kit method that moves a Window, you'd most likely still want it to move a Window. The easiest way to have it do that is to include the old method in the new one through a message to **super**.

```
- moveTo:(NWCoord)x :(NWCoord)y {
    [super moveTo:x :y];
    /* your code goes here */
}
```

You may occasionally be required to implement a new version of a method while preserving the behavior of the method you override. An example is the **write:** method, which archives an object by writing it to a typed stream. When you define a kit subclass, you may need to implement a version of this method that can archive the instance variables your subclass declares. So that a **write:** message will archive all of an object's instance variables, not just those declared in the subclass, your version of the method should begin by incorporating the version used by its superclass.

```
- write:(NXTypedStream *)stream {
    [super write:stream];
    /* your code goes here */
}
```

Method descriptions explicitly mention that you should incorporate a method you override only when it's not obvious that you should preserve the default behavior in the new method.

Designated Initializer Methods

Initializer methods (those that begin with **init...**) initialize a new instance of a class by setting values for instance variables, creating support objects, and so on. Before a new instance receives class-specific initialization, it must be initialized as an instance of each class from which it inherits, in order, beginning with Object. To maintain this sequence, each common and Application Kit class has designated initializers, **init...** methods that invoke a designated initializer in the superclass before doing their work. Since Object is the root of the inheritance hierarchy, its designated initializer, the **init** method, is always the first method to initialize an object. The designated initializer for most other classes is the **init...** method with the most arguments (some classes have more than one designated initializer to perform different types of initialization). Other **init...** methods for a class initialize objects by invoking a designated initializer. Designated initializers are identified in their method descriptions.

In its discussion of the **alloc** and **init** methods, the Object class specification provides more detail on how new instances are allocated and initialized. This discussion includes some guidelines to follow when writing initializer methods in a subclass.

Sending a Message to Perform a Method

Some methods should never appear as messages in the code you write; you should never directly ask an object to perform the method. Typically, these are methods that your application will use indirectly, through other methods.

Most of these methods begin with a underscore and are treated as class-internal methods. However, some don't have an underscore and are included in the method descriptions. These are methods that your application can implement, even though it won't directly use them in a message. The messages to perform these methods originate in the kit.

The most notable example of this is the **drawSelf::** method that draws a View. Although you must implement a **drawSelf::** method for each View subclass you define, your code should never send a **drawSelf::** message. Instead, you send a display message; the display method (such as **display**, **displayIfNeeded**, or **display:::**) sees to it that the drawing context is properly set before initiating a **drawSelf::** message to the View.

The methods that respond to event messages (such as **mouseUp:**, **keyDown:**, and **windowExposed:**) also fall into this category. Event messages are initiated by the Application Kit when it receives events from the Window Server; you shouldn't initiate them in your own code.

The **write:** and **read:** methods for archiving and unarchiving are other examples of methods that shouldn't be sent directly to objects. They're generated by functions, such as **NXWriteObject()** and **NXReadObject()**.

If a method is designed to respond to messages generated by other methods or by a kit, the method description will generally say so. If there's a penalty for generating the message within the code you write (as there is for **drawSelf::**), the description will include an explicit warning.

Common Classes

A handful of classes come with the NeXT compiler for the Objective-C language. They include, most prominently, the Object class, which defines the basic functionality inherited by all objects. The Object class is at the root of all inheritance hierarchies.

The other classes that come with the compiler are similar in that they also define functionality that can serve a wide variety of applications. They can be used with any kit. The five common classes are shown in Figure 2-1.

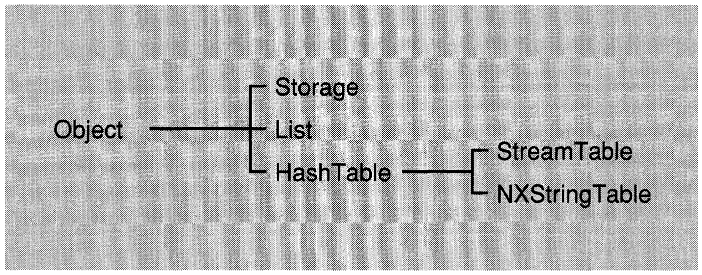

Figure 2-1. Inheritance Hierarchy of the Common Classes

HashTable

INHERITS FROM	Object
DECLARED IN	objc/HashTable.h

CLASS DESCRIPTION

The HashTable class defines objects that store associations of keys and values. You use a HashTable object when you need a convenient and efficient way to store and access unordered data. Hash tables double as their number of associations increase, thus guaranteeing both constant average access time and linear size.

HashTable objects are convenient to use, but when even greater efficiency of storage and access is required, consider using the C function interface to hash tables (see **NXCreateHashTable()**). Two alternatives to the HashTable class are NXStringTable and List. An NXStringTable is a HashTable that's designed to store associations between keys and values that are both character strings. List is useful when you need to store a collection of objects; however, it doesn't provide for storage of key/value pairs. Also, the time required to access an element in a List object grows linearly with the number of elements.

In a HashTable object, keys and values can be of type **id**, **int**, **void ***, **char ***, or any other 32-bit quantity that can be described by a type string. The following outlines the usage of key and value descriptions:

Hashing: A **hash** message is sent for object keys, a string hashing function is used for string keys, and a generic integer hashing function is used for all other cases.

Equality: An **isEqual:** message is sent for object keys, and a string comparison is used for string keys.

Descriptions must be invariant strings and are restricted to encode 32-bit quantities, typically the following:

"@" (id) "*" (char *) "i" (int) "!" (other)

Two other restrictions that a HashTable must satisfy are:

1. Keys must be invariant. In particular, when keys are strings, no copy is made, and the string is assumed to never change until the association is removed from the table.

2. If two keys are equal in the sense of **isEqual:**, then their hashed values must be equal. If you're creating a HashTable of List or Storage objects, note that these classes have an **isEqual:** method but no **hash** method; you can either subclass or define a **hash** method.

When freeing a HashTable, only object keys or object values are freed. Data is archived according to its type description. When description is "%", hashing and equality are same as for "*". On reading, however, the string is uniqued, using the **NXUniqueString()** function.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in HashTable	unsigned	count;
	const char	*keyDesc;
	const char	*valueDesc;

count	Current number of associations
keyDesc	Description of keys
valueDesc	Description of values

METHOD TYPES

Initializing and freeing a HashTable	
	– init
	– initKeyDesc:
	– initKeyDesc:valueDesc:
	– initKeyDesc:valueDesc:capacity:
	– free
	– freeObjects
	– freeKeys:values:
	– empty
Copying a HashTable	– copy
	– copyFromZone:
Manipulating table associations	– count
	– isKey:
	– valueForKey:
	– insertKey:value:
	– removeKey:
Iterating over all associations	– initState
	– nextState:key:value:
Archiving	– read:
	– write:

INSTANCE METHODS

copy

– **copy**

Returns a new HashTable. Keys nor values are copied.

copyFromZone:

– **copyFromZone:**

Returns a new HashTable. Memory for the new HashTable is allocated from *zone*. Keys nor values are copied.

count

– (unsigned)**count**

Returns the number of objects in the table.

empty

– **empty**

Empties the HashTable but retains its capacity.

free

– **free**

Deallocates the table, but not the objects that are in the table.

freeKeys:values:

– **freeKeys:**(void (*)(void *))*keyFunc* **values:**(void (*)(void *))*valueFunc*

Conditionally deallocates the HashTable's associations but does not deallocate the table itself.

freeObjects

– **freeObjects**

Deallocates every object in the HashTable, but not the HashTable itself. Strings are not recovered.

init

– **init**

Initializes a new HashTable to map object keys to object values. Returns **self**.

See also: – **initKeyDesc:key:value:capacity:**

initKeyDesc:

+ **initKeyDesc:**(const char *)*aKeyDesc*

Initializes a new HashTable to map keys as described by *aKeyDesc* to object values. Returns **self**.

See also: – **initKeyDesc:key:value:capacity:**

initKeyDesc:valueDesc:

– **initKeyDesc:**(const char *)*aKeyDesc* **valueDesc:**(const char *)*aValueDesc*

Initializes a new HashTable to map keys and values as described by *aKeyDesc* and *aValueDesc*. Returns **self**.

See also: – **initKeyDesc:key:value:capacity:**

initKeyDesc:valueDesc:capacity:

– **initKeyDesc:**(const char *)*aKeyDesc*
    valueDesc:(const char *)*aValueDesc*
    capacity:(unsigned)*aCapacity*

Initializes a new HashTable. This is the designated initializer for HashTable objects: If you subclass HashTable, your subclass's designated initializer must maintain the initializer chain by sending a message to **super** to invoke this method. See the introduction to the class specifications for more information.

A HashTable initialized by this method maps keys and values as described by *aKeyDesc* and *aValueDesc*. *aCapacity* is given only as a hint; you can use 0 to create a table of minimal size. As more space is needed, it will be allocated automatically. Returns **self**.

See also: – **initKeyDesc:key:value:capacity:**

initState

– (NXHashState)**initState**

Returns an NXHashState structure that's required when iterating through the HashTable. Iterating through all associations of a HashTable involves setting up an iteration state, conceptually private to HashTable, and then progressing until all entries have been visited. An example of counting associations in a table follows:

```
unsigned count = 0;
const   void  *key;
        void  *value;
NXHashState  state = [table initState];
while ([table nextState: &state key: &key value: &value])
    count++;
```

See also: – **nextState:key:value:**

insertKey:value:

– (void *)**insertKey:**(const void *)*aKey* **value:**(void *)*aValue*

Adds or updates a key and value pair, as specified by *aKey* and *aValue.* If *aKey* is already in the hash table, it's associated with *aValue* and its previously associated value is returned. Otherwise, **insertKey:value:** returns **nil**.

See also: – **removeKey:**

isKey:

– (BOOL)**isKey:**(const void *)*aKey*

Returns YES if *aKey* is in the table, otherwise NO.

See also: – **valueForKey:**

nextState:key:value:

– (BOOL)**nextState:**(NXHashState *)*aState*
    key:(const void **)*aKey*
    value:(void **)*aValue*

Moves to the next entry in the HashTable and provides the addresses of pointers to its key/value pair. No **insertKey:** or **removeKey:** should be done while iterating through the table. Returns NO when there are no more entries in the table; otherwise, returns YES.

See also: – **initState**

read:

– **read:**(NXTypedStream *)*stream*

Reads the HashTable from the typed stream *stream*. Returns **self**.

See also: – **write:**

removeKey:

– (void *)**removeKey:**(const void *)*aKey*

Removes the hash table entry identified by *aKey*. Always returns **nil**.

See also: – **insertKey:value:**

valueForKey:

– (void *)**valueForKey:**(const void *)*aKey*

Returns the value mapped to *aKey*. Returns **nil** if *aKey* is not in the table.

See also: – **isKey:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the HashTable to the typed stream *stream*. Returns **self**.

See also: – **read:**

List

INHERITS FROM	Object
DECLARED IN	objc/List.h

CLASS DESCRIPTION

A List is a collection of objects. The class provides an interface that permits easy manipulation of the collection as a fixed or variable-sized list, a set, or an ordered collection. Lists are implemented as arrays to allow fast random access using an index. Indices start at 0.

A List array contains object **id**s. An object isn't copied when it's added to a List; only its **id** is. There are no empty slots within the array. **nil** objects can't be inserted in a List, and the collection is contracted to fill in the empty space when an object is removed.

Lists grow dynamically when new objects are added. The default mechanism automatically doubles the capacity of the List when it becomes full, thus ensuring an average constant time for insertions, independent of the size of the List.

For manipulating sets of structures that aren't objects, see the Storage class.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in List	id	*dataPtr;
	unsigned int	numElements;
	unsigned int	maxElements;

dataPtr	The data managed by the List object (the array of objects).
numElements	The actual number of objects in the array.
maxElements	The total number of objects that can fit in currently allocated memory.

METHOD TYPES

Initializing a new List object	– init
	– initCount:
Copying and freeing a List	– copy
	– copyFromZone:
	– free
Manipulating objects by index	– insertObject:at:
	– addObject:
	– removeObjectAt:
	– removeLastObject
	– replaceObjectAt:with:
	– objectAt:
	– lastObject
	– count
Manipulating objects by **id**	– addObject:
	– addObjectIfAbsent:
	– removeObject:
	– replaceObject:with:
	– indexOf:
Comparing Lists	– isEqual:
Emptying a List	– empty
	– freeObjects
Sending messages to the objects	– makeObjectsPerform:
	– makeObjectsPerform:with:
Managing the storage capacity	– capacity
	– setAvailableCapacity:
Archiving	– read:
	– write:

INSTANCE METHODS

addObject:

– **addObject:***anObject*

Inserts *anObject* at the end of the List, and returns **self**. However, if *anObject* is **nil**, nothing is inserted and **nil** is returned.

See also: – **insertObject:at:**

addObjectIfAbsent:

– **addObjectIfAbsent:***anObject*

Inserts *anObject* at the end of the List and returns **self**, provided that *anObject* isn't already in the List. If *anObject* is in the List, it won't be inserted, but **self** is still returned.

If *anObject* is **nil**, nothing is inserted and **nil** is returned.

See also: – **insertObject:at:**

capacity

– (unsigned int)**capacity**

Returns the maximum number of objects that can be stored in the List without allocating more memory for it. When new memory is allocated, it's taken from the same zone that was specified when the List was created.

See also: – **count**, – **setAvailableCapacity:**

copy

– **copy**

Returns a new List object with the same contents as the receiver. The objects in the List aren't copied; therefore, both Lists contain pointers to the same set of objects. Memory for the new List is allocated from the same zone as the receiver.

See also: – **copyFromZone:**

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Returns a new List object, allocated from *zone*, with the same contents as the receiver. The objects in the List aren't copied; therefore, both Lists contain pointers to the same set of objects.

See also: – **copy**

count

– (unsigned int)**count**

Returns the number of objects currently in the List.

See also: – **capacity**

empty

– **empty**

Empties the List of all its objects without freeing them, and returns **self**. The current capacity of the List isn't changed.

See also: – **freeObjects**

free

– **free**

Deallocates the List object and the memory it allocated for the array of object **id**s. However, the objects themselves aren't freed.

See also: – **freeObjects**

freeObjects

– **freeObjects**

Removes every object from the List, sends each one of them a **free** message, and returns **self**. The List object itself isn't freed and its current capacity isn't altered.

The methods that free the objects shouldn't have the side effect of modifying the List.

See also: – **empty**

indexOf:

– (unsigned int)**indexOf:***anObject*

Returns the index of the first occurrence of *anObject* in the List, or **NX_NOT_IN_LIST** if *anObject* isn't in the List.

init

– **init**

Initializes the receiver, a new List object, but doesn't allocate any memory for its array of object **id**s. It's initial capacity will be 0. Minimal amounts of memory will be allocated when objects are added to the List. Or an initial capacity can be set, before objects are added, using the **setAvailableCapacity:** method. Returns **self**.

See also: – **initCount:**, – **setAvailableCapacity:**

initCount:

– **initCount:**(unsigned int)*numSlots*

Initializes the receiver, a new List object, by allocating enough memory for it to hold *numSlots* objects. Returns **self**.

This method is the designated initializer for the class. It should be used immediately after memory for the List has been allocated and before any objects have been assigned to it; it shouldn't be used to reinitialize a List that's already in use.

See also: – **capacity**

insertObject:at:

– **insertObject:***anObject* **at:**(unsigned int)*index*

Inserts *anObject* into the List at *index*, moving objects down one slot to make room. If *index* equals the value returned by the **count** method, *anObject* is inserted at the end of the List. However, the insertion fails if *index* is greater than the value returned by **count** or *anObject* is **nil**.

If *anObject* is successfully inserted into the List, this method returns **self**. If not, it returns **nil**.

See also: – **count**, – **addObject:**

isEqual:

– (BOOL)**isEqual:***anObject*

Compares the receiving List to *anObject*. If *anObject* is a List with exactly the same contents as the receiver, this method returns YES. If not, it returns NO.

Two Lists have the same contents if they each hold the same number of objects and the **id**s in each List are identical and occur in the same order.

lastObject

– **lastObject**

Returns the last object in the List, or **nil** if there are no objects in the List. This method doesn't remove the object that's returned.

See also: – **removeLastObject**

makeObjectsPerform:

– **makeObjectsPerform:**(SEL)*aSelector*

Sends an *aSelector* message to each object in the List in reverse order (starting with the last object and continuing backwards through the List to the first object), and returns **self**. The *aSelector* method must be one that takes no arguments. It shouldn't have the side effect of modifying the List.

makeObjectsPerform:with:

– **makeObjectsPerform:**(SEL)*aSelector* **with:***anObject*

Sends an *aSelector* message to each object in the List in reverse order (starting with the last object and continuing backwards through the List to the first object), and returns **self**. The message is sent each time with *anObject* as an argument, so the *aSelector* method must be one that takes a single argument of type **id**. The *aSelector* method shouldn't, as a side effect, modify the List.

objectAt:

– **objectAt:**(unsigned int)*index*

Returns the **id** of the object located at slot *index*, or **nil** if *index* is beyond the end of the List.

See also: – **count**

read:

– **read:**(NXTypedStream *)*stream*

Reads the List and all the objects it contains from the typed stream *stream.*

See also: – **write:**

removeLastObject

– **removeLastObject**

Removes the object occupying the last position in the List and returns it. If there are no objects in the List, this method returns **nil**.

See also: – **lastObject**, – **removeObjectAt:**

removeObject:

– **removeObject:***anObject*

Removes the first occurrence of *anObject* from the List, and returns it. If *anObject* isn't in the List, this method returns **nil**.

The positions of the remaining objects in the List are adjusted so there's no gap.

See also: – **removeLastObject**, – **removeObjectAt:**

removeObjectAt:

– **removeObjectAt:**(unsigned int)*index*

Removes the object located at *index* and returns it. If there's no object at *index*, this method returns **nil**.

The positions of the remaining objects in the List are adjusted so there's no gap.

See also: – **removeLastObject**, – **removeObject:**

replaceObject:with:

– **replaceObject:***anObject* **with:***newObject*

Replaces the first occurrence of *anObject* in the List with *newObject*, and returns *anObject*. However, if *newObject* is **nil** or *anObject* isn't in the List, nothing is replaced and **nil** is returned.

See also: – **replaceObjectAt:with:**

replaceObjectAt:with:

– **replaceObjectAt:**(unsigned int)*index* **with:***newObject*

Returns the object at *index* after replacing it with *newObject*. If there's no object at *index* or *newObject* is **nil**, nothing is replaced and **nil** is returned.

See also: – **replaceObject:with:**

setAvailableCapacity:

– **setAvailableCapacity:**(unsigned int)*numSlots*

Sets the storage capacity of the List to at least *numSlots* objects and returns **self**. However, if the List already contains more than *numSlots* objects (if the **count** method returns a number greater than *numSlots*), its capacity is left unchanged and **nil** is returned.

See also: – **capacity**, – **count**

write:

– **write:**(NXTypedStream *)*stream*

Writes the List, including all the objects it contains, to the typed stream *stream.*

See also: – **read:**

NXStringTable

INHERITS FROM	HashTable : Object
DECLARED IN	objc/NXStringTable.h

CLASS DESCRIPTION

NXStringTable defines an object that associates a key with a value. Both the key and the value must be character strings. For example, these keys and values might be associated in a particular NXStringTable:

Key	**Value**
"Yes"	"Oui"
"No"	"Non"

By using an NXStringTable object to store your application's character strings, you can reduce the effort required to adapt the application to different language markets. Interface Builder give you direct access to NXStringTables, letting you create and initialize a string table and connect it into your application.

A new NXStringTable instance can be created either through Interface Builder's Classes window or through the inherited **alloc...** and **init...** methods. Similarly, you can establish the contents of an NXStringTable either directly through Interface Builder or programmatically through NXStringTable methods that read keys and values that are stored in a file (see **readFromFile:** and **readFromStream:**). Each assignment in the file can be of either of these formats:

```
"key" = "value";
"key";
```

If only *key* is present for a particular assignment, the corresponding value is taken to be identical to *key.*

A valid key or value—a valid token—is composed of text enclosed in double quotes. The text can't include double quotes or the null character. It can include the escape sequences: \a, \b, \f, \n, \r, \t, \v, and \". The backslash is stripped for any other character; consequently, numeric escape codes aren't interpreted. White space between tokens is ignored. A key or value can't exceed MAX_NXSTRINGTABLE_LENGTH characters.

The file can also include standard C-language comments which the NXStringTable ignores. However, these comments can provide valuable information for a person who's translating or documenting the application.

To retrieve the value associated with a specific key, send a **valueForStringKey:** message to the NXStringTable. For example, assuming *myStringTable* is an NXStringTable containing the appropriate keys and values, this call would display an attention panel announcing a problem opening a file:

```
NXRunAlertPanel([myStringTable valueForStringKey:"openTitle"],
               [myStringTable valueForStringKey:"openError"],
               "OK",
               NULL,
               NULL);
```

If you're accessing NXStringTables through Interface Builder, please note the following. For efficiency, use several NXStringTables—each in its own interface file—rather than one large one. By using several NXStringTables, your application can load only those strings that it needs at a particular time. For example, you might place all the strings associated with a help system in an NXStringTable in one interface file and those associated with error messages in another NXStringTable in another file. When the user accesses the help system for the first time, the application can load the appropriate NXStringTable. Also, instantiate only one copy of any individual NXStringTable. Don't put an NXStringTable object in an interface file that will be loaded more than once, since multiple copies of the same table will result.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from HashTable	unsigned	count;
	const char	*keyDesc;
	const char	*valueDesc;
Declared in NXStringTable	(none)	

METHOD TYPES

Initializing and freeing an NXStringTable	
	– init
	– free
Querying an NXStringTable	– valueForStringKey:
Reading and writing elements	– readFromFile:
	– writeToFile:
	– readFromStream:
	– writeToStream:

INSTANCE METHODS

free

– **free**

Frees the string table and its strings. You should never send a **freeObjects** (HashTable) message to an NXStringTable.

init

– **init**

Initializes a new NXStringTable. This is the designated initializer for the NXStringTable class. Returns **self**.

readFromFile:

– **readFromFile:**(const char *)*fileName*

Reads an ASCII representation of the NXStringTable's keys and values from *fileName*. The NXStringTable opens a stream on the file and then sends itself a **readFromStream:** message to load the data. See "Class Description" above for the format of the data. Returns **nil** on error; otherwise, returns **self**.

See also: – **readFromStream:**

readFromStream:

– **readFromStream:**(NXStream *)*stream*

Reads an ASCII representation of the NXStringTable's keys and values from *stream*. See "Class Description" above for the format of the data. Returns **nil** on error; otherwise, returns **self**.

See also: – **readFromFile:**

valueForStringKey:

– (const char *)**valueForStringKey:**(const char *)*aString*

Searches the string table for the value that corresponds to the key *aString*. Returns NULL if and only if no value is found for that key; otherwise, returns a pointer to the value.

writeToFile:

– **writeToFile:**(const char *)*fileName*

Writes an ASCII representation of the NXStringTable's keys and values to *fileName*. The NXStringTable opens a stream on the file and then sends itself a **writeToStream:** message. See "Class Description" above for the format of the data. Returns **nil** if an error occurs; otherwise, returns **self**.

See also: – **writeToStream:**

writeToStream:

– **writeToStream:**(NXStream *)*stream*

Writes an ASCII representation of the NXStringTable's keys and values to *stream*. See "Class Description" above for the format of the data. Returns **self**.

See also: – **writeToFile:**

CONSTANTS AND DEFINED TYPES

```
#define MAX_NXSTRINGTABLE_LENGTH    1024
```

Object

INHERITS FROM	none *(Object is the root class.)*
DECLARED IN	objc/Object.h

CLASS DESCRIPTION

Object is an abstract superclass that defines a basic interface to the Objective-C run-time system that other classes use and build upon. It's the root of all Objective-C inheritance hierarchies, the only class that has no superclass. All other classes inherit from Object.

Among other things, the Object class provides its subclasses with a framework for creating, initializing, freeing, copying, comparing, and archiving objects, for performing methods selected at run-time, for querying an object about its methods and its position in the inheritance hierarchy, and for forwarding messages to other objects. For example, to query an object about what class it belongs to, you'd send it a **class** or a **name** message. To find out whether it implements a particular method, you'd send it a **respondsTo:** message.

This type of information is obtained through the object's **isa** instance variable, which points to a class structure that describes the object to the run-time system. Because all objects directly or indirectly inherit from the Object class, they all have this variable. The installation of the class structure (the initialization of **isa**) is one of the responsibilities of the **alloc**, **allocFromZone:**, and **new** methods, the same methods that create (allocate memory for) new instances of a class. The defining characteristic of an "object" is that its first instance variable is an **isa** pointer to a class structure.

INSTANCE VARIABLES

Declared in Object	Class	isa;
isa	A pointer to the instance's class structure.	

METHOD TYPES

Initializing the class	+ initialize

Creating, copying, and freeing instances	
	+ alloc
	+ allocFromZone:
	+ new
	– copy
	– copyFromZone:
	– zone
	– free
	+ free
Initializing a new instance	– init
Identifying classes	– class
	+ class
	– name
	– superClass
	+ superClass
Identifying and comparing instances	
	– hash
	– isEqual:
	– self
Testing inheritance relationships	– isKindOf:
	– isKindOfGivenName:
	– isMemberOf:
	– isMemberOfGivenName:
Testing class functionality	+ instancesRespondTo:
	– respondsTo:
Sending messages determined at run time	
	– perform:
	– perform:with:
	– perform:with:with:
Forwarding messages	– forward::
	– performv::
Obtaining method handles	+ instanceMethodFor:
	– methodFor:
Posing	+ poseAs:
Enforcing intentions	– notImplemented:
	– subclassResponsibility:
Error handling	– doesNotRecognize:
	– error:

Dynamic loading	+ finishLoading: + startUnloading
Archiving	– read: – write: – startArchiving: – awake – finishUnarchiving + setVersion: + version

CLASS METHODS

alloc

+ **alloc**

Returns a new instance of the receiving class. The **isa** instance variable of the new object is initialized to a data structure that describes the class; otherwise the object isn't initialized. A version of the **init** method should be used to complete the initialization process. For example:

```
id newObject = [[TheClass alloc] init];
```

Subclasses shouldn't override **alloc** to add code that initializes the new instance. Instead, class-specific versions of the **init** method should be implemented for that purpose. Versions of the **new** method can also be implemented to combine allocation and initialization.

Note: The **alloc** method doesn't invoke **allocFromZone:**. The two methods work independently.

See also: + **allocFromZone:**, – **init**, + **new**

allocFromZone

+ **allocFromZone:**(NXZone *)*zone*

Returns a new instance of the receiving class. The **isa** instance variable of the new object is initialized to a data structure that describes the class; its other instance variables aren't initialized. Memory for the new object is allocated from *zone*.

This method is always used in conjunction with an **init** method that completes the initialization of the new instance. For example:

```
id newObject = [[TheClass allocFromZone:someZone] init];
```

The **allocFromZone:** method shouldn't be overridden to include any initialization code. Instead, class-specific versions of the **init** method should be implemented for tha purpose.

When one object creates another, it's often a good idea to make sure they're both allocated from the same region of memory. The **zone** method can be used for this purpose; it returns the zone where the receiver is located. For example:

```
id myCompanion = [[TheClass allocFromZone:[self zone]] init];
```

See also: + **alloc**, – **zone**, – **init**

class

+ **class**

Returns **self**. Since this is a class method, it returns the class object.

See also: – **name**, – **class**

finishLoading:

+ **finishLoading:**(struct mach_header *)*header*

Implemented by subclasses to integrate a newly loaded class or category into a runnin program. A **finishLoading:** message is sent to the class object immediately after the class, or a category of the class, has been dynamically loaded—if the newly loaded class or category implements a method that can respond. *header* is a pointer to the structure that describes the modules that were just loaded.

Once a dynamically loaded class is used, it will also receive an **initialize** message. However, because the **finishLoading:** message is sent immediately after the class is loaded, it always precedes the **initialize** message, which is sent only when the class receives its first message from the program.

A **finishLoading:** method is specific to the class or category where it's defined, and isn't inherited by subclasses or shared with the rest of the class. Thus a class that ha four categories can define a total of five **finishLoading:** methods, one in each categor and one in the main class definition. The method that's performed is the one defined i the class or category just loaded.

There's no default **finishLoading:** method. The Object class declares a protocol for this method, but doesn't implement it.

See also: + **startUnloading**

free

+ free

Returns **nil**. This method is implemented to prevent class objects, which are "owned" by the Objective-C run-time system, from being accidentally freed. To free an instance, use the instance method **free**.

See also: – **free**

initialize

+ initialize

Initializes the class before it's used (before it receives its first message). The Objective-C run-time system generates an **initialize** message to each class just before the class, or any class that inherits from it, is sent its first message from within the program. Each class object receives the **initialize** message just once. Superclasses receive it before subclasses do.

For example, if the first message your program sends is this,

```
[Application alloc]
```

the run-time system will generate these three **initialize** messages,

```
[Object initialize];
[Responder initialize];
[Application initialize];
```

since Application is a subclass of Responder and Responder is a subclass of Object. All the **initialize** messages precede the **alloc** message and are sent in the order of inheritance, as shown.

If your program later begins to use the Text class,

```
[Text instancesRespondTo:someSelector]
```

the run-time system will generate these **initialize** messages,

```
[View initialize];
[Text initialize];
```

since the Text class inherits from Object, Responder, and View. The **instancesRespondTo:** message is sent only after all these classes are initialized. Note that the **initialize** messages to Object and Responder aren't repeated; each class is initialized only once.

You can implement your own versions of **initialize** to provide class-specific initialization as needed.

Because **initialize** methods are inherited, it's possible for the same method to be invoked many times, once for the class that defines it and once for each inheriting class. To prevent code from being repeated each time the method is invoked, it can be bracketed as shown in the example below:

```
+ initialize
{
    if ( self == [MyClass class] ) {
        /* put initialization code here */
    }
    return self;
}
```

See also: – **init**, – **class**

instanceMethodFor:

+ (IMP)**instanceMethodFor:**(SEL)*aSelector*

Locates and returns the address of the implementation of the *aSelector* instance method. An error is generated if instances of the receiver can't respond to *aSelector* messages.

This method, and the function pointer that it returns, are subject to the same constraints as those described for the instance method **methodFor:**.

See also: – **methodFor:**

instancesRespondTo:

+ (BOOL)**instancesRespondTo:**(SEL)*aSelector*

Returns YES if instances of the class are capable of responding to *aSelector* messages, and NO if they're not. The application is responsible for determining whether a NO response should be considered an error.

If an instance can successfully forward *aSelector* messages to other objects, it will be able to receive the message without error even though **instancesRespondTo:** returns NO.

See also: – **respondsTo:**

new

+ new

Creates a new instance of the receiving class, sends it an **init** message, and returns the initialized object returned by **init**.

As defined in the Object class, **new** is essentially a combination of **alloc** and **init**. Like **alloc**, it initializes the **isa** instance variable of the new object so that it points to the class data structure; it leaves the initialization of other instance variables up to the **init** method.

Unlike **alloc**, **new** is sometimes reimplemented in subclasses to have it invoke a class-specific initialization method. If the **init** method includes arguments, they're typically reflected in the **new** method. For example:

```
+ newArg:(int)tag arg:(struct info *)data
{
    return [[self alloc] initArg:tag arg:data];
}
```

However, there's little point in implementing **new...** methods if they're simply shorthand for **alloc** and **init...**, like the one shown above. Often **new...** methods will do more than just allocation and initialization. In some classes, they manage a set of instances, returning the one with the requested properties if it already exists, allocating and initializing a new one only if necessary. For example:

```
+ newArg:(int)tag arg:(struct info *)data
{
    id theInstance;

    if ( theInstance = findTheObjectWithTheTag(tag) )
        return theInstance;
    return [[self alloc] initArg:tag arg:data];
}
```

Although it's appropriate to define new **new...** methods in this way, the **alloc** and **allocFromZone:** methods should never be augmented to include initialization code.

See also: **– init**, **+ alloc**, **+ allocFromZone:**

poseAs:

+ **poseAs:***aClassObject*

Permits the receiver to "pose as" the *aClassObject* class. All messages to *aClassObject* will actually be sent to the receiver. The receiver should be defined as a subclass of *aClassObject* and shouldn't declare any instance variables of its own. A **poseAs:** message should be sent before any instances of *aClassObject* are created.

This facility allows you to add methods to an existing class by defining them in a subclass and having the subclass pose as the existing class. The new method definitions will be inherited by all subclasses of the existing class. Care should be taken to ensure that this doesn't generate errors.

Posing is useful as a debugging tool, but category definitions are a less complicated and more efficient way of augmenting existing classes.

Posing has only one feature that categories lack: The methods added by a posing class can override methods already defined for the existing class. You can therefore use posing to replace existing methods with new versions.

Returns **self**.

setVersion:

+ **setVersion:**(int)*aVersion*

Sets the class version number to *aVersion*, and returns **self**. The version number is helpful when instances of the class are to be archived and reused later.

See also: + **version**

startUnloading

+ **startUnloading**

Implemented by subclasses to prepare for the class or category being unloaded from a running program. A **startUnloading** message is sent to the class object immediately before the class, or category of the class, is unloaded—if a method that can respond is implemented in the class or category about to be unloaded.

A **startUnloading** method is specific only to the class or category where it's defined, and isn't inherited by subclasses or shared with the rest of the class. Thus a class that has four categories can define a total of five **startUnloading** methods, one in each category and one in the main class definition. The method that's performed is the one defined in the class or category that will be unloaded.

There's no default **startUnloading** method. The object class declares a protocol for this method but doesn't implement it.

See also: + **finishLoading:**

superClass

+ **superClass**

Returns the class object for the receiver's superclass.

See also: + **class**, – **superClass**

version

+ (int)**version**

Returns the version number assigned to the class. If no version has been set, this will be 0.

See also: + **setVersion:**

INSTANCE METHODS

awake

– **awake**

Implemented by subclasses to reinitialize the receiving object after it has been unarchived (by **read:**). An **awake** message is automatically sent to every object after it has been unarchived and after all the objects it refers to are in a usable state.

The default version of the method defined here merely returns **self**.

You can implement an **awake** method in any class to provide for more initialization than can be done in the **read:** method. Each implementation of **awake** should limit the work it does to the scope of the class definition, and incorporate the initialization of classes farther up the inheritance hierarchy through a message to **super**. For example:

```
- awake
{
    [super awake];
    /* class-specific initialization goes here */
    return self;
}
```

All implementations of **awake** should return **self**.

See also: – **read:**, – **finishUnarchiving**

class

– **class**

Returns the class object for the receiver s class.

See also: + **class**

copy

– **copy**

Returns a new instance that s an exact copy of the receiver. This method creates only one new object. If the receiver has instance variables that point to other objects, the instance variables in the copy will point to the same objects. The values of the instance variables are copied, but the objects they point to aren t.

See also: – **copyFromZone:**

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Returns a new instance that s an exact copy of the receiver. Memory for the new instance is allocated from *zone*. This method creates only one new object; it works exactly like the **copy** method, except that it allows you to determine where the copy will reside in memory.

See also: – **copy**, – **zone**

doesNotRecognize:

– **doesNotRecognize:**(SEL)*aSelector*

Handles *aSelector* messages that the receiver doesn t recognize. The Objective-C run-time system invokes this method whenever an object receives an *aSelector* message that it can t respond to or forward. It, in turn, invokes the **error:** method to generate an error message and abort the current process.

doesNotRecognize: messages should be sent only by the run-time system. Although they re sometimes used in program code to prevent a method from being inherited, it s better to use the **error:** method directly. For example, an Object subclass might renounce the **copy** method by reimplementing it to include an **error:** message as follows:

```
- copy
{
    [self error:"  %s objects should not be sent  %s  messages\n",
            [self name], sel_getName(_cmd)];
}
```

This code prevents instances of the subclass from recognizing or forwarding **copy** messages   although the **respondsTo:** method will still report that the receiver has access to a **copy** method.

(The **_cmd** variable identifies to the current selector; in the example above, it identifies the selector for the **copy** method. The **sel_getName()** function returns the method name corresponding to a selector code; in the example, it returns the name copy .)

See also: – **error:**, – **subclassResponsibility:**, – **name**

error:

– **error:**(STR)*aString*, **...**

Generates a formatted error message, in the manner of **printf()**, from *aString* followed by a variable number of arguments. For example:

```
[self error:"index %d exceeds limit %d\n", index, limit];
```

The message specified by *aString* is preceded by this standard prefix (where *<class>* is the name of the receiver s class):

```
"error: <class> "
```

This method doesn t return. After generating the error message, it calls **abort()** to create a core file and terminate the process. It works through the Objective-C run-time **_error** function.

See also: – **subclassResponsibility:**, – **notImplemented:**, – **doesNotRecognize:**

finishUnarchiving

– **finishUnarchiving**

Implemented by subclasses to replace an unarchived object with a new object if necessary. A **finishUnarchiving** message is sent to every object after it has been unarchived (using **read:**) and initialized (by **awake**), but only if a method has been implemented that can respond to the message.

The **finishUnarchiving** message gives the application an opportunity to test an unarchived and initialized object to see whether it s usable, and, if not, to replace it with another object that is. This method should return **nil** if the unarchived instance (**self**) is OK; otherwise, it should free the receiver and return another object to take its place.

There s no default implementation of the **finishUnarchiving** method. The Object class declares this method, but doesn t define it.

See also: – **read:**, – **awake**, – **startArchiving:**

forward::

– **forward:**(SEL)*aSelector* **:**(marg_list)*argFrame*

Implemented by subclasses to forward messages to other objects. When an object is sent an *aSelector* message, and the run-time system can't find an implementation of the method for the receiving object, it sends the object a **forward::** message to give it an opportunity to delegate the message to another object. (If that object can't respond to the message either, it too will be given a chance to forward it.)

The **forward::** message thus allows an object to establish relationships with other objects that will, for certain messages, act on its behalf. The forwarding object is, in a sense, able to "inherit" some of the characteristics of the object it forwards the message to.

A **forward::** message is generated only if the *aSelector* method isn't implemented by the receiving object's class or by any of the classes it inherits from.

An implementation of the **forward::** method can do more than just forward messages. It can, for example, locate code that responds to a variety of different messages, thus avoiding the necessity of having to write a separate method for each selector. A **forward::** method might also involve several other objects in the response to a message, rather than forward it to just one.

If implemented to forward messages, a **forward::** method has two tasks:

- To locate an object that can respond to the *aSelector* message. This need not be the same object for all messages.
- To send the message to that object, using the **performv::** method.

In the simple case, in which an object forwards messages to just one destination (such as the hypothetical **friend** instance variable in the example below), a **forward::** method could be as simple as this:

```
- forward:(SEL)aSelector :(marg_list)argFrame
{
    if ( [friend respondsTo:aSelector] )
        return [friend performv:aSelector :argFrame];
    return [self doesNotRecognize:aSelector];
}
```

argFrame is a pointer to the arguments included in the original *aSelector* message. It's passed directly to **performv::** without change.

The default version of **forward::** implemented in the Object class simply invokes the **doesNotRecognize:** method. It doesn't forward messages. Thus if you choose not to implement **forward::** methods, unrecognized messages will be handled in the usual way.

See also: – **performv::**, – **doesNotRecognize:**

free

– free

Frees the memory occupied by the receiver and returns **nil**. This method also sets the **isa** pointer of the freed object to **nil**, so that subsequent messages to the object will generate an error indicating that a message was sent to a freed object.

This method uses **object_deallocate**() to free the receiver's memory.

hash

– (unsigned int)**hash**

Returns an unsigned integer that's guaranteed to be the same for any two objects which are equal according to the **isEqual:** method. The integer is derived from the **id** of the receiver.

See also: – **isEqual:**

init

– init

Implemented by subclasses to initialize a new object (the receiver) immediately after memory for it has been allocated. An **init** message is generally coupled with an **alloc** or **allocFromZone:** message in the same line of code:

```
id newObject = [[TheClass alloc] init];
```

Subclass versions of this method should return the new object (**self**) if it has been successfully initialized. If it can't be initialized, they should free the object and return **nil**. The version of the method defined here simply returns **self**.

Every class must guarantee that the **init** method returns a fully functional instance of the class. This typically means overriding the method to add class-specific initialization code. Subclass versions of the method need to incorporate the initialization code for the classes they inherit from, through a message to **super**:

```
- init
{
    [super init];
    /* class-specific initialization goes here */
    return self;
}
```

Note that the message to super precedes the initialization code added in the method. This ensures that initialization proceeds in the order of inheritance.

Subclasses often add arguments to the **init** method to allow specific values to be set. The more arguments a method has, the more freedom it gives you to determine the character of initialized objects. Classes often have a set of **init...** methods, each with a different number of arguments. For example:

```
- init;
- initArg:(int)tag;
- initArg:(int)tag arg:(struct info *)data;
```

The convention is that at least one of these methods, usually the one with the most arguments, includes a message to **super** to incorporate the initialization of classes higher up the hierarchy. This method is the *designated initializer* for the class. The other **init...** methods defined in the class directly or indirectly invoke the designated initializer through messages to **self**. In this way, all **init...** methods are chained together. For example:

```
- init
{
    return [self initArg:-1];
}

- initArg:(int)tag
{
    return [self initArg:tag arg:NULL];
}

- initArg:(int)tag arg:(struct info *)data
{
    [super init. . .];
    /* class-specific initialization goes here */
}
```

In this example, the **initArg:arg:** method is the designated initializer for the class.

If a subclass does any initialization of its own, it must define its own designated initializer. This method should begin by sending a message to **super** to perform the designated initializer of its superclass. Suppose, for example, that the three methods illustrated above are defined in the B class. The C class, a subclass of B, might have this designated initializer:

```
- initArg:(int)tag arg:(struct info *)data arg:anObject
{
    [super initArg:tag arg:data];
    /* class-specific initialization goes here */
}
```

If inherited **init...** methods are to successfully initialize instances of the subclass, they must all be made to (directly or indirectly) invoke the new designated initializer. To accomplish this, the subclass is obliged to cover (override) only the designated initializer of the superclass. For example, in addition to its designated initializer, the C class would also implement this method:

```
- initArg:(int)tag arg:(struct info *)data
{
    return [self initArg:tag arg:data arg:nil];
}
```

This ensures that all three methods inherited from the B class also work for instances of the C class.

Often the designated initializer of the subclass overrides the designated initializer of the superclass. If so, the subclass need only implement the one **init...** method.

These conventions maintain a direct chain of **init...** links, and ensure that the **new** method and all inherited **init...** methods return usable, initialized objects. They also prevent the possibility of an infinite loop wherein a subclass method sends a message (to **super**) to perform a superclass method, which in turn sends a message (to **self**) to perform the subclass method.

The Object class also has a designated initializer—this **init** method. Subclasses that do their own initialization should override it, as described above.

See also: + **new**, + **alloc**, + **allocFromZone:**

isEqual:

– (BOOL)**isEqual:***anObject*

Returns YES if the receiver is the same as *anObject*, and NO if it isn't. This is determined by comparing the **id** of the receiver to the **id** of *anObject.*

The **hash** method is guaranteed to return the same integer for both objects when this method returns YES.

See also: **hash**

isKindOf:

– (BOOL)**isKindOf:***aClassObject*

Returns YES if the receiver is an instance of *aClassObject* or an instance of any class that inherits from *aClassObject*. Otherwise, it returns NO. For example, in this code **isKindOf:** would return YES:

```
id  aMenu = [[Menu alloc] init];
if ( [aMenu isKindOf:[Window class]] )
    . . .
```

In the Application Kit, the Menu class inherits from Window.

See also: – **isMemberOf:**

isKindOfGivenName:

– (BOOL)**isKindOfGivenName:**(STR)*aClassName*

Returns YES if the receiver is an instance of *aClassName* or an instance of any class that inherits from *aClassName*. This method is the same as **isKindOf:**, except it takes the class name, rather than the class **id**, as its argument.

STR is defined, in **objc/objc.h**, as a character pointer (**char ***).

See also: – **isMemberOfGivenName:**

isMemberOf:

– (BOOL)**isMemberOf:***aClassObject*

Returns YES if the receiver is an instance of *aClassObject*. Otherwise, it returns NO. For example, in this code, **isMemberOf:** would return NO:

```
id  aMenu = [[Menu alloc] init];
if ([aMenu isMemberOf:[Window class]])
    . . .
```

See also: – **isKindOf:**

isMemberOfGivenName:

– (BOOL)**isMemberOfGivenName:**(STR)*aClassName*

Returns YES if the receiver is an instance of *aClassName*, and NO if it isn't. This method is the same as **isMemberOf:**, except it takes the class name, rather than the class **id**, as its argument.

STR is defined, in **objc/objc.h**, as a character pointer (**char ***).

See also: – **isKindOfGivenName:**

methodFor:

– (IMP)**methodFor:**(SEL)*aSelector*

Locates and returns the address of the receiver's implementation of the *aSelector* method. An error is generated if the receiver has no implementation of the method (if it can't respond to *aSelector* messages).

IMP is defined (in the **objc/objc.h** header file) as a pointer to a function that takes a variable number of arguments and returns an **id**. It's the only prototype provided for the function pointer that **methodFor:** returns. Therefore, if the *aSelector* method takes any arguments or returns anything but an **id**, its function counterpart will be inadequately prototyped. Lacking a prototype, the compiler will promote **float**s to **double**s and **char**s to **int**s, which the implementation won't expect. It will therefore behave differently (and erroneously) when called as a function than when performed as a method.

To remedy this situation, it's necessary to provide your own prototype. In the example below, **IMPEqual** is used to prototype the implementation of the **isEqual:** method. It's defined as pointer to a function that returns a BOOL and takes an **id** in addition to the two "hidden" arguments (**self**, the current receiver, and **_cmd**, the current selector) that are passed to every method implementation.

```
typedef BOOL (*IMPEqual)(id, SEL, id);
IMPEqual tester;
tester = (IMPEqual)[target methodFor:@selector(isEqual:)];

while ( !tester(target, @selector(isEqual:), someObject) ) {
    . . .
}
```

Note that turning a method into a function by obtaining the address of its implementation "unhides" the **self** and **_cmd** arguments.

See also: + **instanceMethodFor:**

name

– (const char *)**name**

Returns a character string with the name of the receiver's class. This information is often used in error messages or debugging statements.

See also: + **class**

notImplemented:

– **notImplemented:**(SEL)*aSelector*

Used in the body of a method definition to indicate that the programmer intended to implement the method, but left it as a stub for the time being. *aSelector* is the selector for the unimplemented method; **notImplemented:** messages are sent to **self**. For example:

```
- methodNeeded
{
    [self notImplemented:_cmd];
}
```

When a **methodNeeded** message is received, **notImplemented:** will invoke the **error:** method to generate an appropriate error message and abort the process. (In this example, **_cmd** refers to the **methodNeeded** selector.)

See also: – **subclassResponsibility:**, – **error:**

perform:

– **perform:**(SEL)*aSelector*

Sends an *aSelector* message to the receiver and returns the result of the message. This allows you to send messages that aren't determined until run time. For example, all three of the following messages do the same thing:

```
id myClone = [anObject copy];
id myClone = [anObject perform:@selector(copy)];
id myClone = [anObject perform:sel_getUid("copy")];
```

aSelector should identify a method that takes no arguments. If the method returns anything but an object, the return must be cast to the correct type. For example:

```
char *myClass;
myClass = (char *)[anObject perform:@selector(name)];
```

Casting works for any integral type (**char**, **short**, **int**, **long**, or **enum**) or any pointer. However, it doesn't work if the return is a floating type (**float** or **double**) or a structure or union. This is because the C language doesn't permit a pointer (like **id**) to be cast to these types.

Therefore, **perform:** shouldn't be asked to perform any method that returns a floating type, structure, or union. An alternative is to get the address of the method implementation (using **methodFor:**) and call it as a function. For example:

```
float grayValue;
grayValue = ((float (*)())[anObject methodFor:@selector(gray)])();
```

See also: – **perform:with:**, – **perform:with:with:**, – **methodFor:**

perform:with:

– **perform:**(SEL)*aSelector* **with:***anObject*

Sends an *aSelector* message to the receiver with *anObject* as an argument. This method is the same as **perform:**, except that you can supply an argument for the *aSelector* message. *aSelector* should identify a method that takes a single argument of type **id**.

See also: – **perform:**

perform:with:with:

– **perform:**(SEL)*aSelector*
  with:*object1*
  with:*object2*

Sends the receiver an *aSelector* message with *object1* and *object2* as arguments. This method is the same as **perform:**, except that you can supply two arguments for the *aSelector* message. *aSelector* should identify a method that can take the two arguments of type **id**.

See also: – **perform:**

performv::

– **performv:**(SEL)*aSelector* **:**(marg_list)*argFrame*

Sends the receiver an *aSelector* message with the arguments in *argFrame*. **performv::** messages are used within implementations of the **forward::** method. Both arguments, *aSelector* and *argFrame*, are identical to the arguments the run-time system passes to **forward::**. They can be taken directly from that method and passed through without change to **performv::**.

performv:: should be restricted to implementations of the **forward::** method. Although it may seem like a more flexible way of sending messages than **perform:**, **perform:with:**, or **perform:with:with:**, in that it doesn't restrict the number of arguments in the *aSelector* message or their type, it's not an appropriate substitute for those methods. First, it's more expensive than they are. The run-time system must parse the arguments in *argFrame* based on information stored for *aSelector*. Second, in future releases **performv::** may not work in contexts other than the **forward::** method.

See also: – **forward::**, – **perform:**

read:

– **read:**(NXTypedStream *)*stream*

Implemented by subclasses to read the receiver's instance variables from the typed stream *stream*. You need to implement a **read:** method for any class you create, if you want its instances (or instance of classes that inherit from it) to be archivable.

The method you implement should unarchive the instance variables defined in the class in a manner that matches they way they were archived by **write:**. In each class, the **read:** method should begin with a message to **super**:

```
- read:(NXTypedStream *)stream
{
    [super read:stream];
    /* class-specific code goes here */
    return self;
}
```

This ensures that all inherited instance variables will also be unarchived.

All implementations of the **read:** method should return **self**.

After an object has been read, it's sent an **awake** message so that it can reinitialize itself, and may also be sent a **finishUnarchiving** message.

See also: – **awake**, – **finishUnarchiving**, – **write:**

respondsTo:

– (BOOL)**respondsTo:**(SEL)*aSelector*

Returns YES if the receiver implements or inherits a method that can respond to *aSelector* messages, and NO if it doesn't. The application is responsible for determining whether a NO response should be considered an error.

Note that if the receiver is able to forward the *aSelector* message to another object, it will be able to respond to the message (albeit indirectly), even though this method returns NO.

See also: – **forward::**, + **instancesRespondTo:**

self

– **self**

Returns the receiver.

See also: + **class**

startArchiving:

– **startArchiving:**(NXTypedStream *)*stream*

Implemented by subclasses to prepare an object for being archived—that is, for being written to the typed stream *stream*. A **startArchiving:** message is sent to an object just before it's archived—but only if it implements a method that can respond. The message gives the object an opportunity to do anything necessary to get itself, or the stream, ready before a **write:** message begins the archiving process.

There's no default implementation of the **startArchiving:** method. The Object class declares the method, but doesn't define it.

See also: – **awake**, – **finishUnarchiving**, – **write:**

subclassResponsibility:

– **subclassResponsibility:**(SEL)*aSelector*

Used in an abstract superclass to indicate that its subclasses are expected to implement *aSelector* methods. If a subclass fails to implement the method, the method is inherited from the superclass and an error is generated.

For example, if subclasses are expected to implement **doSomething** methods, the superclass would define this version of the method:

```
- doSomething
{
    [self subclassResponsibility:_cmd];
}
```

When this method is invoked, **subclassResponsibility:** will, working through Object's **error:** method, abort the process and generate an appropriate error message.

The **_cmd** variable identifies the current method selector, just as **self** identifies the current receiver. In the example above, it identifies the selector for the **doSomething** method.

Subclass implementations of the *aSelector* method shouldn't include messages to **super** to incorporate the superclass version. If they do, they'll also generate an error.

See also: – **doesNotRecognize:**, – **notImplemented:**, – **error:**

superClass

– **superClass**

Returns the class object for the receiver's superclass.

See also: + **superClass**

write:

– **write:**(NXTypedStream *)*stream*

Implemented by subclasses to write the receiver's instance variables to the typed stream *stream*. You need to implement a **write:** method for any class you create if you want to be able to archive its instances (or instances of classes that inherit from it).

The methods you implement should archive only the instance variables defined in the class, but should begin with a message to **super** so that all inherited instance variables will also be archived:

```
- write:(NXTypedStream *)stream
{
    [super write:stream];
    /* class-specific archiving code goes here */
    return self;
}
```

All implementations of the **write:** method should return **self**.

During the archiving process, **write:** methods may be performed twice, so they shouldn't do anything other than write instance variables to a typed stream.

See also: – **read:**, – **startArchiving:**

zone

– (NXZone *)**zone**

Returns a pointer to the zone from which the receiver was allocated. Objects created without specifying a zone are allocated from the default zone, which is returned by **NXDefaultMallocZone**().

See also: + **allocFromZone:**, + **alloc**, + **copyFromZone:**

Storage

INHERITS FROM	Object
DECLARED IN	objc/Storage.h

CLASS DESCRIPTION

The Storage class implements a general storage allocator. Each Storage object manages an array containing data elements of an arbitrary type. When an element is added to the object, it's copied into the array.

As is the case with List objects, Storage arrays grow dynamically when necessary. Their capacity doesn't need to be explicitly adjusted.

Because a Storage object holds elements of an arbitrary type, you don't have to define a special class for each type of data you want to store. When setting up a new instance of the class, you specify the size of the elements and a description of their type. The type description is needed for archiving the object and must agree with the specified element size. It's encoded in a string using the descriptor codes listed below:

Type	**Code**	**Type**	**Code**
char	c	Class	#
char *	*	id	@
NXAtom	%	SEL	:
int	i	int (ignored)	!
short	s	structure	{<*types*>}
float	f	array	[<*count*><*types*>]
double	d		

For example, "[15d]" means an array of fifteen **double**s, and "{csi*@}" means a structure containing a **char**, a **short**, an **int**, a character pointer, and an object. The descriptor "%" specifies a unique string pointer. When it's unarchived, the **NXUniqueString()** function is used to make sure that it's also unique within the new context. The "!" descriptor requires that the data be the same size as an **int**; the data won't be archived.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in Storage	void	*dataPtr;
	const char	*description;
	unsigned int	numElements;
	unsigned int	maxElements;
	unsigned int	elementSize;

dataPtr	A pointer to the data stored by the object.
description	A string encoding the type of data stored.
numElements	The number of elements actually in the Storage array.
maxElements	The total number of elements that can fit within currently allocated memory.
elementSize	The size of each element in the array.

METHOD TYPES

Initializing a new Storage instance	– init – initCount:elementSize:description:
Copying and freeing Storage objects	– copy – copyFromZone: – free
Getting, adding, and removing elements	– addElement: – insert:at: – removeAt: – removeLastElement – replace:at: – empty – elementAt:
Comparing Storage objects	– isEqual:
Managing the storage capacity and type	– count – description – setAvailableCapacity: – setNumSlots:
Archiving	– read: – write:

INSTANCE METHODS

addElement:

– **addElement:**(void *)*anElement*

Adds *anElement* at the end of the Storage array and returns **self**. The size of the array is increased if necessary.

See also: – **insert:at:**

copy

– **copy**

Returns a new Storage object containing the same data as the receiver. The data as well as the object is copied, but the two objects share the same description string. Memory for the copy is taken from the same zone as the receiver.

See also: – **copyFromZone:**

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Returns a new Storage object containing the same data as the receiver. The data as well as the object is copied, and memory for both is taken from *zone*. The two objects share the same description string.

See also: – **copy**

count

– (unsigned)**count**

Returns the number of elements currently in the Storage array.

See also: – **setNumSlots:**

description

– (const char *)**description**

Returns the string encoding the data type of elements in the Storage array.

See also: – **initCount:elementSize:description:**

elementAt:

– (void *)**elementAt:**(unsigned)*index*

Returns a pointer to the element at *index* in the Storage array. If no element is stored at *index* (*index* is beyond the end of the array), a NULL pointer is returned.

Before using the pointer that's returned, you must convert it into the appropriate type by a cast. The pointer can be used either to read the element at *index* or to alter it.

See also: – **replace:at:**, – **insert:at:**

empty

– **empty**

Empties the Storage array of all its elements and returns **self**. The current capacity of the array remains unchanged.

See also: – **free**

free

– **free**

Frees the Storage object and all the elements it contains. Pointers stored in the object will be freed, but the data they point to won't be (unless the data is also stored in the object). You might want to free the data before freeing the Storage object. The description string isn't freed.

See also: – **empty**

init

– **init**

Initializes the Storage object so that it's ready to store object **id**s. The initial capacity of the array isn't set. In general, it's better to store object **id**s in a List object. Returns **self**.

See also: – **initCount:elementSize:description:**, – **initCount:** (List)

initCount:elementSize:description:

– **initCount:**(unsigned)*count*
    elementSize:(unsigned)*sizeInBytes*
    description:(const char *)*string*

Initializes the Storage object so that it will have room for at least *count* elements. Each element is of size *sizeInBytes* and of the type described by *string*. If *string* is NULL, the object won't be archivable. Once set, the description string should never be modified. Returns **self**.

This method is the designated initializer for the class. It's used to initialize Storage objects immediately after they have been allocated; it should never be used to reinitialize a Storage object that's already been used.

insert:at:

– **insert:**(void *)*anElement* **at:**(unsigned)*index*

Puts *anElement* in the Storage array at *index*. All elements between *index* and the last element are shifted to make room. The size of the array is increased if necessary. Returns **self**.

See also: – **addElement:**, – **setNumSlots:**

isEqual:

– (BOOL)**isEqual:***anObject*

Compares the receiver with *anObject*, and returns YES if they're the same and NO if they're not. Two Storage objects are considered to be the same if they have the same number of elements and the elements at each position in the array match.

read:

– **read:**(NXTypedStream *)*stream*

Reads the Storage object and the data it stores from the typed stream *stream*.

See also: – **write:**

removeAt:

– **removeAt:**(unsigned)*index*

Removes the element located at *index* from the Storage array and returns **self**. All elements between *index* and the last element are shifted to close the gap.

See also: – **removeLastElement**

removeLastElement

– **removeLastElement**

Removes the last element from the Storage array and returns **self**.

See also: – **removeAt:**

replace:at:

– **replace:**(void *)*anElement* **at:**(unsigned)*index*

Replaces the data at *index* with the data pointed to by *anElement*. However, if no element is stored at *index* (*index* is beyond the end of the array), nothing is replaced. Returns **self**.

See also: – **elementAt:**, – **insert:at:**

setAvailableCapacity:

– **setAvailableCapacity:**(unsigned)*numSlots*

Sets the storage capacity of the array to at least *numSlots* elements and returns **self**. If the array already contains more than *numSlots* elements, its capacity is left unchanged and **nil** is returned.

See also: – **setNumSlots:**, – **count**

setNumSlots:

– **setNumSlots:**(unsigned)*numSlots*

Sets the number of elements in the Storage array to *numSlots* and returns **self**. If *numSlots* is greater than the current number of elements in the array (the value returned by **count**), the new slots will be filled with zeros. If *numSlots* is less than the current number of elements in the array, access to all elements with indices equal to or greater than *numSlots* will be lost.

If necessary, this method increases the capacity of the storage array so there's room for at least *numSlots* elements.

See also: – **setAvailableCapacity:**, – **count**

write:

– **write:**(NXTypedStream *)*stream*

Writes the Storage object and its data to the typed stream *stream*.

See also: – **read:**

StreamTable

INHERITS FROM	HashTable : Object
DECLARED IN	objc/StreamTable.h

CLASS DESCRIPTION

This class reads and writes a set of independent data structures on streams. Its goal is to provide incremental saving of files, as a cheap way to implement very primitive data bases. Both read and write operations are lazy, e.g., reading a StreamTable file only implies reading of the directory.

Although StreamTable inherits from HashTable, very few methods can be directly inherited because internal representations of values differ. Nevertheless, the HashTable abstraction is retained, and StreamTable is described as an object class in order to simplify usage and implementation. The only inherited methods are **count** and **isKey:**. In order to read and write a StreamTable, the usual **read:** and **write:** methods can be performed.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from HashTable	unsigned	count;
	const char	*keyDesc;
	const char	*valueDesc;
Declared in StreamTable	(none)	

METHOD TYPES

Creating and freeing a StreamTable	
	– free
	– freeObjects
	+ new
	+ newKeyDesc:
Manipulating table elements	– insertStreamKey:value:
	– removeStreamKey:
	– valueForStreamKey:
Iterating over all elements	– initStreamState
	– nextStreamState:key:value:
Archiving	– read:
	– write:

CLASS METHODS

new

+ new

Returns a new StreamTable with objects as keys.

newKeyDesc:

+ newKeyDesc:(const char *)*aKeyDesc*

Returns a new StreamTable. Keys must be 32-bit quantities described by *aKeyDesc*.

INSTANCE METHODS

free

– free

Deallocates the table, but not the objects that are in the table.

freeObjects

– freeObjects

Deallocates every object in the StreamTable, but not the StreamTable itself. Strings are not recovered.

initStreamState

– (NXHashState)**initStreamState**

Iterating over all elements of a StreamTable involves setting up an iteration state, conceptually private to StreamTable, and then progressing until all entries have been visited. An example of counting elements in a table follows:

```
unsigned count = 0;
const void  *key;
void        *value;
NXHashState state = [table initStreamState];
while ([table nextStreamState:&state key:&key value: &value])
    count++;
```

initState begins the process of iteration through the StreamTable.

See also: **nextStreamState:key:value:**

insertStreamKey:value:

– (id)**insertStreamKey:**(const void *)*aKey* **value:**(id)*aValue*

Adds or updates *akey/avalue* pair.

nextStreamState:key:value:

– (BOOL)**nextStreamState:**(NXHashState *)*aState*
  key:(const void **)*aKey*
  value:(id *)*aValue*

Moves to the next entry in the StreamTable. No **insertStreamKey:** or **removeStreamKey:** should be done while iterating through the table.

See also: **initStreamState**

read:

– **read:**(NXTypedStream *)*stream*

Reads the StreamTable from the typed stream *stream.*

removeStreamKey:

– (id)**removeStreamKey:**(const void *)*aKey*

Removes *akey/avalue* pair. Always returns **nil**.

valueForStreamKey:

– (id)**valueForStreamKey:**(const void *)*aKey*

Returns the value mapped to *aKey.* Returns **nil** if *aKey* is not in the table.

write:

– **write:**(NXTypedStream *)*stream*

Writes the StreamTable to the typed stream *stream.*

Application Kit Classes

The class specifications for the Application Kit describe over 50 classes. The inheritance hierarchy for these classes is shown in Figure 2-2.

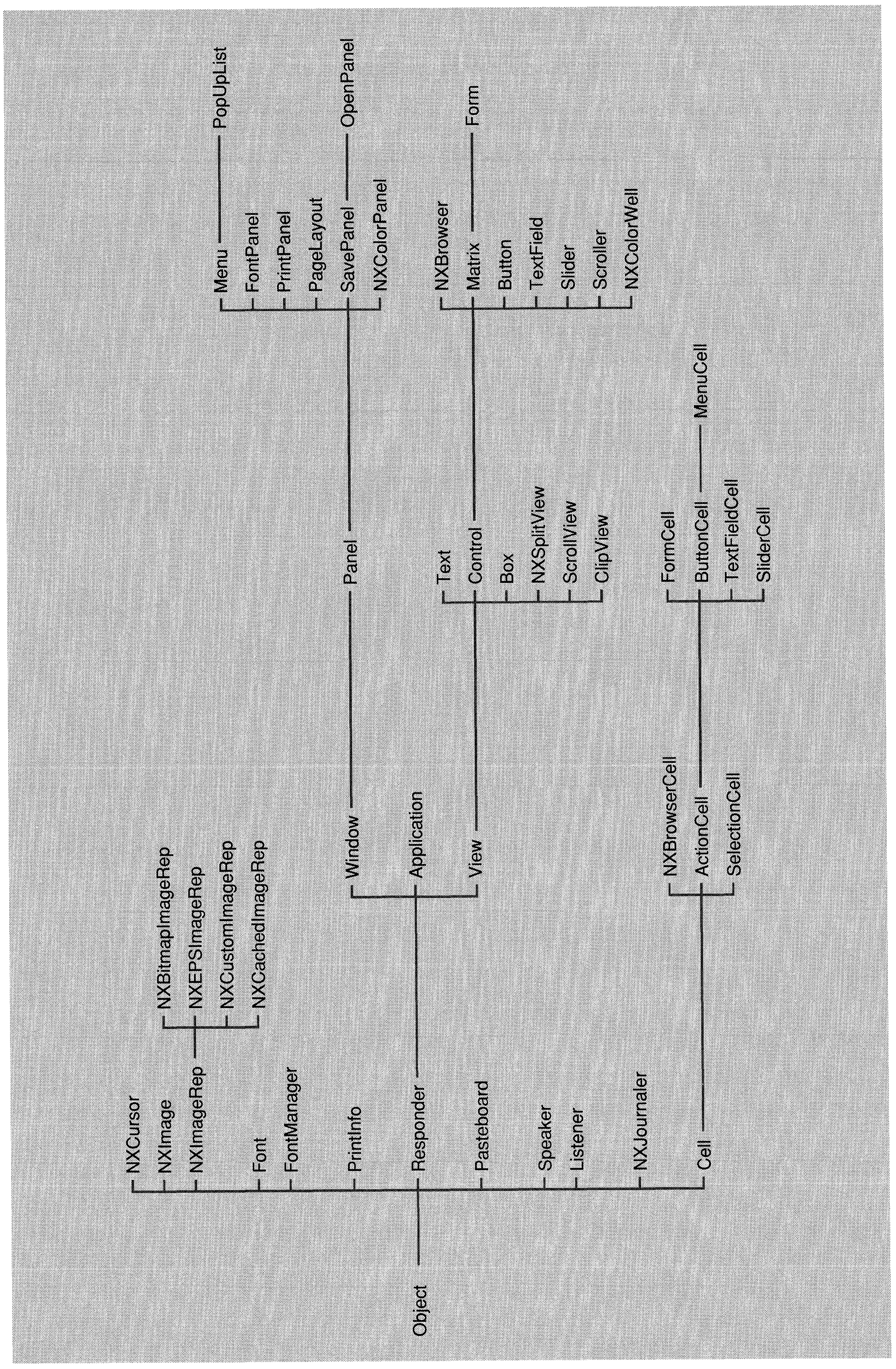

Figure 2-2. Application Kit Inheritance Hierarchy

ActionCell

INHERITS FROM	Cell: Object
DECLARED IN	appkit/ActionCell.h

CLASS DESCRIPTION

An ActionCell defines the active area inside a control (an instance of Control or one of its subclasses). You can set an ActionCell's control only by sending the **drawSelf:inView:** message to the ActionCell, passing the control as the second argument.

A single control may have more than one ActionCell. An integer tag, provided as the instance variable **tag**, is used to identify an ActionCell object; this is of particular importance to controls that contain more than one ActionCell. Note, however, that no checking is done by the ActionCell object itself to ensure that the tag is unique. See the Matrix class for an example of a subclass of Control that contains multiple ActionCells.

ActionCell defines the **target** and **action** instance variables and methods for setting them. These define the ActionCell's target object and action method. As the user manipulates a control, ActionCell's **trackMouse:inRect: ofView:** method (inherited from Cell) sends the action message to the target object with the **id** of the Control object as the only argument.

Many of the methods that define the contents and look of an ActionCell, such as **setFont:** and **setBordered:**, are reimplementations of methods inherited from Cell. They're subclassed to ensure that the ActionCell is redisplayed if it's currently in a control.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Cell	char	*contents;
	id	support;
	struct _cFlags1	cFlags1;
	struct _cFlags2	cFlags2;
Declared in ActionCell	int	tag;
	id	target;
	SEL	action;

tag	Reference number for the object.
target	The object's notification target.
action	The message to send to the target.

METHOD TYPES

Configuring the ActionCell	– setAlignment: – setBezeled: – setBordered: – setEnabled: – setFloatingPointFormat:left:right: – setFont: – setIcon:
Manipulating ActionCell values	– doubleValue – floatValue – intValue – setStringValue: – setStringValueNoCopy:shouldFree: – stringValue
Displaying	– controlView – drawSelf:inView:
Target and action	– action – setAction: – setTarget: – target
Assigning a tag	– setTag: – tag
Archiving	– read: – write:

INSTANCE METHODS

action

– (SEL)**action**

Returns the selector for the receiver's action method. Keep in mind that the argument to an ActionCell's action method is the object's Control (the object returned by **controlView**).

See also: – **setAction:**

controlView

– **controlView**

Returns the Control object in which the receiver was most recently drawn. In general, you should use the object returned by this method only to (indirectly) redisplay the receiver. For example, the subclasses of ActionCell defined by the Application Kit invoke this method in order to send the returned object a message such as **updateCellInside:**.

The Control in which an ActionCell is drawn is set through the **drawSelf:inView:** method (only).

See also: – **drawSelf:inView:**

doubleValue

– (double)**doubleValue**

Returns the receiver's contents as a **double**.

See also: – **setDoubleValue:**(Cell), – **doubleValue** (Cell)

drawSelf:inView:

– **drawSelf:**(const NXRect *)*cellFrame* **inView:***controlView*

Displays the ActionCell by sending

```
[super drawSelf:cellFrame inView:controlView];
```

Sets the receiver's Control (the **controlView** instance variable) to *controlView* if and only if *controlView* is a Control object (in other words, an instance of Control or a subclass thereof).

See also: – **drawSelf:inView:** (Cell)

floatValue

– (float)**floatValue**

Returns the receiver's **contents** as a float.

See also: – **setFloatValue:**(Cell), – **floatValue** (Cell)

intValue

– (int)**intValue**

Returns the receiver's **contents** as an **int**.

See also: – **setIntValue:**(Cell), – **intValue** (Cell)

read:

– **read:**(NXTypedStream *)*stream*

Reads and returns an object of class ActionCell from *stream.*

setAction:

– **setAction:**(SEL)*aSelector*

Sets the receiver's action method to *aSelector.* Keep in mind that the argument to an ActionCell's action method is the object's Control (the object returned by **controlView**). Returns **self.**

See also: **– setTarget:**, – **sendAction:to:** (Control)

setAlignment:

– **setAlignment:**(int)*mode*

If the receiver is a text Cell (type NX_TEXTCELL), this sets its text alignment to *mode*, which should be NXLEFTALIGNED, NX_CENTERED, or NX_RIGHTALIGNED. If it's currently in a Control view, the receiver is redisplayed. Returns **self.**

setBezeled:

– **setBezeled:**(BOOL)*flag*

Adds or removes the receiver's bezel, as *flag* is YES or NO. Adding a bezel will remove the receiver's (flat) border, if any. If it's currently in a Control view, the receiver is redisplayed. Returns **self.**

See also: – **setBordered:**

setBordered:

– **setBordered:**(BOOL)*flag*

Adds or removes the receiver's border, as *flag* is YES or NO. The border is black and has a width of 1.0. Adding a border will remove the receiver's bezel, if any. If it's currently in a Control view, the receiver is redisplayed. Returns **self.**

See also: – **setBezeled:**

setEnabled:

– **setEnabled:**(BOOL)*flag*

Enables or disables the receiver's ability to receive mouse events as *flag* is YES or NO. If it's currently in a Control view, the receiver is redisplayed. Returns **self**.

setFloatingPointFormat:left:right:

– **setFloatingPointFormat:**(BOOL)*autoRange*
  left:(unsigned int)*leftDigits*
  right:(unsigned int)*rightDigits*

Sets the receiver's floating point format. If it's currently in a Control view, the receiver is redisplayed. Returns **self**.

See also: – **setFloatingPointFormat:left:right:** (Cell)

setFont:

– **setFont:***fontObj*

If the receiver is a text Cell (type NX_TEXTCELL), this sets its font to *fontObj*. In addition, if it's currently in a Control view, the receiver is redisplayed. Returns **self**.

setIcon:

– **setIcon:**(const char *)*iconName*

Sets the receiver's icon to *iconName* and sets its Cell type to NX_ICONCELL. If it's currently in a Control view, the receiver is redisplayed. Returns **self**.

See also: – **setIcon:** (Cell)

setStringValue:

– **setStringValue:**(const char *)*aString*

Sets the receiver's contents to a copy of *aString*. If it's currently in a Control view, the receiver is redisplayed. Returns **self**.

See also: – **setStringValue:** (Cell)

setStringValueNoCopy:shouldFree:

– **setStringValueNoCopy:**(char *)*aString* **shouldFree:**(BOOL)*flag*

Sets the receiver's contents to a *aString*. If *flag* is YES, *aString* will be freed when the receiver is freed. If it's currently in a Control view, the receiver is redisplayed. Returns **self**.

See also: – **setStringValueNoCopy:shouldFree:** (Cell)

setTag:

– **setTag:**(int)*anInt*

Sets the receiver's tag to *anInt*. Returns **self**.

setTarget:

– **setTarget:***anObject*

Sets the receiver's target to *anObject*. Returns **self**.

See also: – **setAction:**

stringValue

– (const char *)**stringValue**

Returns the receiver's contents as a string. Returns **self**.

See also: – **setStringValue:**, – **stringValue** (Cell)

tag

– (int)**tag**

Returns the receiver's **tag**.

target

– **target**

Returns the receiver's target.

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiver to *stream*. Returns **self**.

Application

INHERITS FROM	Responder : Object
DECLARED IN	appkit/Application.h

CLASS DESCRIPTION

The Application class provides the framework for program execution; every program must have exactly one Application object. Creating the object connects the program to the Window Server and initializes its PostScript environment. The Application object maintains a list of all the Windows in the application, thereby allowing it to retrieve every View in the application. To make it readily accessible to other objects, the Application object for your program is assigned to the global variable **NXApp**.

The main task of the Application object is to receive events from the Window Server and distribute them to the proper Responders. System events are handled by the Application object itself. Window events are translated into event messages for the affected Window object. Key-down events that occur when the Command key is pressed are translated into **commandKey:** messages that every Window has an opportunity to respond to. Other keyboard and mouse events are sent to the Window associated with the event; the Window then distributes them to the objects in its view hierarchy.

Subclassing the Application class is discouraged. Instead of placing the functionality of your program in an Application object, you should place that functionality in one or more modules which are subclasses of the Object class. Your program will then tend to be more reusable, and can be invoked from a small dispatcher object rather than being closely tied to the Application code.

The Application object can be assigned a delegate that responds to notification messages on the Application object's behalf. The easiest way to make your own object the Application object's delegate is to Control-drag a connection from the File's Owner icon to your object in Interface Builder, and connect it as the delegate. Many of the notification methods are sent back to the Application object if the delegate doesn't respond, but the preferred technique is to have the delegate respond to these messages. The notification messages are listed below, divided into two categories:

Delegate Only	**Delegate or Application subclass**
appDidHide:	appAcceptsAnotherFile:
appDidUnhide:	app:openFile:type:
appWillUpdate:	app:openTempFile:type:
appDidUpdate:	appDidInit:
appDidBecomeActive:	app:powerOffIn:andSave:
appDidResignActive:	app:unmounting:
powerOff:	applicationDefined:

Note that of the methods in the second category the Application class implements only the **applicationDefined:** method, and that it implements that method only to forward the message to the delegate.

Since an application must have one and only one Application object, you must use **new** to create it. You can't use **alloc**, **allocFromZone:**, or **init** to create or initialize an Application object.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Declared in Application	char	*appName;
	NXEvent	currentEvent;
	id	windowList;
	id	keyWindow;
	id	mainWindow;
	id	delegate;
	int	*hiddenList;
	int	hiddenCount;
	const char	*hostName;
	DPSContext	context;
	int	contextNum;
	id	appListener;
	id	appSpeaker;
	port_t	replyPort;
	NXSize	screenSize;
	short	running;
	struct __appFlags {	
	unsigned int	hidden:1;
	unsigned int	autoupdate:1;
	unsigned int	active:1;
	}	appFlags;

appName	The name of your application; used by the defaults system and the application's Listener object.
currentEvent	The event most recently retrieved from the event queue.
windowList	A List of all the windows belonging to the application.
keyWindow	The Window that receives keyboard events.

mainWindow	The Window that receives menu commands and action messages from a Panel.
delegate	The object that responds to delegated messages.
hiddenList	The Window Server's List for Windows in the application at the time the application is hidden.
hiddenCount	The number of windows referred to by **hiddenList**.
hostName	The name of the machine running the Window Server.
context	The Display PostScript context connected to the Window Server.
contextNum	A number identifying the application's Display PostScript context.
appListener	The Application object's Listener.
appSpeaker	The Application object's Speaker.
replyPort	A general purpose reply port for the Application object's Speakers.
screenSize	The size of the screen that this application is running on.
running	The nested level of **run** and **runModalFor:**.
appFlags.hidden	YES if the application's windows are currently hidden.
appFlags.autoupdate	YES if the Application object is to send an **update** message to each Window after an event has been processed.
appFlags.active	YES if the application is the active application.

METHOD TYPES

Initializing the class	+ initialize
Creating and freeing instances	+ new
	– free
Setting up the application	– loadNibFile:owner:
	– loadNibFile:owner:withNames:
	– loadNibFile:owner:withNames:fromZone:
	– loadNibSection:owner:
	– loadNibSection:owner:withNames:
	– loadNibSection:owner:withNames: fromHeader:
	– loadNibSection:owner:withNames: fromZone:
	– loadNibSection:owner:withNames: fromHeader:fromZone:
	– appName
	– setMainMenu:
	– mainMenu
Changing the active application	– activate:
	– activateSelf:
	– activeApp
	– becomeActiveApp
	– deactivateSelf
	– isActive
	– resignActiveApp
Running the event loop	– run
	– stop:
	– runModalFor:
	– stopModal
	– stopModal:
	– abortModal
	– beginModalSession:for:
	– runModalSession:
	– endModalSession:
	– delayedFree:
	– isRunning
	– sendEvent:
Getting and peeking at events	– currentEvent
	– getNextEvent:
	– getNextEvent:waitFor:threshold:
	– peekAndGetNextEvent:
	– peekNextEvent:into:
	– peekNextEvent:into:waitFor:threshold:

Journaling	– isJournalable
	– setJournalable:
	– masterJournaler
	– slaveJournaler
Handling user actions and events	– applicationDefined:
	– hide:
	– isHidden
	– unhide
	– unhide:
	– unhideWithoutActivation:
	– powerOff:
	– powerOffIn:andSave:
	– rightMouseDown:
	– unmounting:ok:
Sending action messages	– sendAction:to:from:
	– tryToPerform:with:
	– calcTargetForAction:
Remote messaging	– setAppListener:
	– appListener
	– setAppSpeaker:
	– appSpeaker
	– appListenerPortName
	– replyPort
Managing Windows	– appIcon
	– findWindow:
	– getWindowNumbers:count:
	– keyWindow
	– mainWindow
	– makeWindowsPerform:inOrder:
	– setAutoupdate:
	– updateWindows
	– windowList
Managing the Windows menu	– setWindowsMenu:
	– windowsMenu
	– arrangeInFront:
	– addWindowsItem:title:filename:
	– removeWindowsItem:
	– changeWindowsItem:title:filename:
	– updateWindowsItem:
Managing the Services menu	– setServicesMenu:
	– servicesMenu
	– registerServicesMenuSendTypes: andReturnTypes:
	– validRequestorForSendType:andReturnType:

Managing screens	– mainScreen – colorScreen – getScreens:count: – getScreenSize:
Querying the application	– context – focusView – hostName
Language	– systemLanguages
Opening files	– openFile:ok: – openTempFile:ok:
Printing	– setPrintInfo: – printInfo – runPageLayout:
Color	– orderFrontColorPanel:
Terminating the application	– terminate:
Assigning a delegate	– setDelegate: – delegate

CLASS METHODS

alloc

Generates an error message. This method cannot be used to create an Application object. Use **new** instead.

See also: + **new**

allocFromZone:

Generates an error message. This method cannot be used to create an Application object. Use **new** instead.

See also: + **new**

initialize

+ initialize

Registers defaults used by the Application class. You never send this message directly; it's sent for you when your application starts. Returns **self**.

new

+ **new**

Creates a new Application object and assigns it to the global variable **NXApp**. A program can have only one Application object, so this method just returns **NXApp** if the Application object already exists. This method also makes a connection to the Window Server, loads the PostScript procedures the application needs, and completes other initialization. Your program should generally invoke this method as one of the first statements in **main()**; this is done for you if you create your application with Interface Builder. The Application object is returned.

See also: – **run**

INSTANCE METHODS

abortModal

– (void)**abortModal**

Aborts the modal event loop by raising the NX_abortModal exception, which is caught by **runModalFor:**, the method that started the modal loop. Since this method raises an exception, it never returns; **runModalFor:**, when stopped with this method, returns NX_RUNABORTED. This method is typically invoked from procedures registered with **DPSAddTimedEntry()**, **DPSAddPort()**, or **DPSAddFD()**. Note that you can't use this method to abort modal sessions, where you control the modal loop and periodically invoke **runModalSession:**.

See also: – **runModalFor:**, – **stopModal**, – **stopModal:**

activate:

– (int)**activate:**(int)*contextNumber*

Makes the application identified by *contextNumber* the active application. *contextNumber* is the PostScript context number of the application to be activated. Normally, you shouldn't invoke this method; the Application Kit is responsible for proper activation. The previously active application's PostScript context number is returned.

See also: – **isActive**, – **activateSelf:**, – **deactivateSelf**

activateSelf:

– (int)**activateSelf:**(BOOL)*flag*

Makes the receiving application the active application. If *flag* is NO, the application is activated only if no other application is currently active. Normally, this method is invoked with *flag* set to NO. When the WorkSpace Manager launches an application, it deactivates itself, so **activateSelf:**NO allows the application to become active if the user waits for it to launch, but the application remains unobtrusive if the user activates another application. If *flag* is YES, the application will always activate. Regardless of the setting of *flag*, there may be a time lag before the application activates; you should not assume that the application will be active immediately after sending this message.

Note that you can make one of your Windows the key window without changing the active application; when you send a **makeKeyWindow** message to a Window, you simply ensure that the Window will be the key window when the application is active.

You should rarely have a need to invoke this method. Under most circumstances the Application Kit takes care of proper activation. However, you might find this method useful if you implement your own methods for inter-application communication. This method returns the PostScript context number of the previously active application.

See also: – **activeApp**, – **activate:**, – **deactivateSelf**, – **makeKeyWindow** (Window)

activeApp

– (int)**activeApp**

Returns the active application's PostScript context number. If no application is active, returns zero.

See also: – **isActive**, – **activate:**

addWindowsItem:title:filename:

– **addWindowsItem:***aWindow* **title:**(const char *)*aString*
    filename:(BOOL)*isFilename*

Adds an item to the Windows menu corresponding to the Window *aWindow*. If *isFilename* is NO, *aString* appears literally in the menu. If *isFilename* is YES, *aString* is assumed to be a converted name with the filename preceding the path, as placed in a Window title by Window's **setTitleAsFilename:** method. If an item for *aWindow* already exists in the Windows menu, this method has no effect. You rarely invoke this method because an item is placed in the Windows menu for you whenever a Window's title is set. Returns **self**.

See also: – **changeWindowsItem:title:filename:**, – **setTitle:** (Window),
– **setTitleAsFilename:** (Window)

appIcon

– **appIcon**

Returns the Window that represents the application in the Workspace Manager.

applicationDefined:

– **applicationDefined:**(NXEvent *)*theEvent*

Handles the application-defined (NX_APPDEFINED) event *theEvent*. The default implementation forwards the message to the receiver's delegate (if the delegate responds to the message). You should either provide a delegate implementation or override this method in your subclass of Application if you want to handle such events. If the delegate responds to this message, the delegate's return value is returned; otherwise returns **self**.

appListener

– **appListener**

Returns the Application object's Listener—the object that will receive messages sent to the port that's registered for the application's name. If you don't send a **setAppListener:** message before your application starts running, an instance of Listener is created for you.

See also: – **setAppListener:**, – **appListenerPortName**, – **run**

appListenerPortName

– (const char *)**appListenerPortName**

Returns the name used to register the Application object's Listener. The default is the same name that's returned by the Application object's **appName** method. If a different name is desired, this method should be overridden. Messages sent by name to **appListenerPortName** will be received by your Application object.

See also: – **checkInAs:** (Listener), – **appName**, **NXPortFromName()**

appName

– (const char *)**appName**

Returns the name under which the Application object has been registered for defaults. This name is also used for messaging unless the messaging name was changed with an override of **appListenerPortName**.

See also: – **appListenerPortName**

appSpeaker

– **appSpeaker**

Returns the Application object's Speaker. You can use this object to send messages to other applications.

See also: – **setSendPort:** (Speaker)

arrangeInFront:

– **arrangeInFront:***sender*

Arranges all of the windows listed in the Windows menu in front of all other windows. Windows associated with the application but not listed in the Windows menu are not ordered to the front. Returns **self**.

See also: – **removeWindowsItem:**, – **makeKeyAndOrderFront:** (Window)

becomeActiveApp

– **becomeActiveApp**

Sends the **appDidBecomeActive:** message to the Application object's delegate. This method is invoked when the application is activated. You never send a **becomeActiveApp** message directly, but you can override this method in a subclass. Returns **self**.

See also: – **activateSelf:**, – **appDidBecomeActive:** (delegate)

beginModalSession:for:

– (NXModalSession *)**beginModalSession:**(NXModalSession *)*session*
  for:*theWindow*

Prepares the application for a modal session with *theWindow.* In other words, prepares the application so that mouse events get to it only if they occur in *theWindow.* If *session* is NULL, a NXModalSession is allocated; otherwise the given storage is used. (The sender could declare a local NXModalSession variable for this purpose.) *theWindow* is made the key window and ordered to the front.

beginModalSession:for: should be balanced by **endModalSession:**. If an exception is raised, **beginModalSession:for:** arranges for proper cleanup. Do NOT use NX_DURING constructs to send an **endModalSession:** message in the event of an exception. Returns the NXModalSession pointer that's used to refer to this session.

See also: – **runModalSession:**, – **endModalSession:**

calcTargetForAction:

– **calcTargetForAction:**(SEL)*theAction*

Returns the first object in the responder chain that responds to the message *theAction*. The message isn't actually dispatched. Note that this method doesn't test the value that the responding object would return should the message be sent; specifically, it doesn't test to see if the responder would return **nil**. Returns **nil** if no responder is found.

See also: – **sendAction:to:from:**

changeWindowsItem:title:filename:

– **changeWindowsItem:***aWindow* **title:**(const char *)*aString*
filename:(BOOL)*isFilename*

Changes the item for *aWindow* in the Windows menu to *aString*. If *aWindow* doesn't have an item in the Windows menu, this method adds the item. If *isFilename* is NO, *aString* appears literally in the menu. If *isFilename* is YES, *aString* is assumed to be a converted name with the filename preceding the path, as placed in a Window title by Window's **setTitleAsFilename:** method. Returns **self**.

See also: – **addWindowsItem:title:filename:**, – **setTitle:** (Window),
– **setTitleAsFilename:** (Window)

colorScreen

– (const NXScreen *)**colorScreen**

Returns the screen that can best represent color. This method will always return a screen, even if no color screen is present.

See also: **NXBPSFromDepth()**

context

– (DPSContext)**context**

Returns the Application object's Display PostScript context.

currentEvent

– (NXEvent *)**currentEvent**

Returns a pointer to the last event the Application object retrieved from the event queue. A pointer to the current event is also passed with every event message.

See also: – **getNextEvent:waitFor:threshold:**,
– **peekNextEvent:waitFor:threshold:**

deactivateSelf

– **deactivateSelf**

Deactivates the application if it's active. Normally, you shouldn't invoke this method; the Application Kit is responsible for proper deactivation. Returns **self**.

See also: – **activeApp**, – **activate:**, – **activateSelf:**

delayedFree:

– **delayedFree:***theObject*

Frees *theObject* by sending it the **free** message after the application finishes responding to the current event and before it gets the next event. If this method is performed during a modal loop, *theObject* is freed after the modal loop ends. Returns **self**.

delegate

– **delegate**

Returns the Application object's delegate.

See also: – **setDelegate:**

endModalSession:

– **endModalSession:**(NXModalSession *)*session*

Cleans up after a modal session. *session* should be from a previous invocation of **beginModalSession:for:**.

See also: – **runModalSession:**, – **beginModalSession:for:**

findWindow:

– **findWindow:**(int)*windowNum*

Returns the Window object that corresponds to the window number *windowNum*. This method is of primary use in finding the Window object associated with a particular event.

See also: – **windowNum** (Window)

focusView

– **focusView**

Returns the View that is currently focused on, or **nil** if no View is focused on.

See also: – **lockFocus** (View)

free

– **free**

Closes all the Application object's windows, breaks the connection to the Window Server, and frees the Application object.

getNextEvent:

– (NXEvent *)**getNextEvent:**(int)*mask*

Gets the next event from the Window Server and returns a pointer to its event record. This method is similar to **getNextEvent:waitFor:threshold:** with an infinite timeout and a threshold of NX_MODALRESPTHRESHOLD.

See also: – **getNextEvent:waitFor:threshold**, – **run**, – **runModalFor:**, – **currentEvent**

getNextEvent:waitFor:threshold:

– (NXEvent *)**getNextEvent:**(int)*mask*
  waitFor:(double)*timeout*
  threshold:(int)*level*

Gets the next event from the Window Server and returns a pointer to its event record. Only events that match *mask* are returned; **getNextEvent:waitFor:threshold:** goes through the event queue, starting from the head, until it finds an event matching *mask*. Events that are skipped are left in the queue. Note that **getNextEvent:waitFor:threshold:** doesn't alter the window event masks that determine which events the Window Server will send to the application.

If an event matching the mask doesn't arrive within *timeout* seconds, this method returns a NULL pointer.

You can use this method to short circuit normal event dispatching and get your own events. For example, you may want to do this in response to a mouse-down event in order to track the mouse while it's down. In this case, you would set *mask* to accept mouse-dragged, mouse-entered, mouse-exited, or mouse-up events.

level determines what other tasks should be performed when the event queue is examined. Tasks that may be performed include procedures to deal with timed-entries, procedures to handle messages received on ports, or procedures to read new data from files. Any such procedure that needs to be called will be called if its priority (specified when the procedure is registered) is equal to or higher than *level*.

In general, modal responders should pass NX_MODALRESPTHRESHOLD for *level*. The main run loop uses a threshold of NX_BASETHRESHOLD, allowing all procedures (except those registered with priority 0) to be checked and invoked if needed.

See also: – **peekNextEvent:waitFor:threshold:**, – **run**, – **runModalFor:**

getScreens:count:

– **getScreens:**(const NXScreen **)*list* **count:**(int *)*numScreens*

Gets screen information for every screen connected to the system. A pointer to an array of NXScreen structures is placed in the variable indicated by *list*, and the number of NXScreen structures in that array is placed in the variable indicated by *numScreens*. Returns **self**.

getScreenSize:

– **getScreenSize:**(NXSize *)*theSize*

Gets the size of the main screen, in units of the screen coordinate system, and places it in the structure pointed to by *theSize*. Returns **self**.

getWindowNumbers:count:

– **getWindowNumbers:**(int **)*list* **count:**(int *)*numWindows*

Gets the window numbers for all the Application object's Windows. A pointer to a non-NULL-terminated **int** array is placed in the variable indicated by *list*. The number of entries in this array is placed in the integer indicated by *numWindows*. The order of window numbers in the array is the same as their order in the Window Server's screen list, which is their front-to-back order on the screen. The application is responsible for freeing the *list* array when done. Returns **self**.

See also: **NXWindowList()**

hide:

– **hide:***sender*

Collapses the application's graphics—including all its windows, menus, and panels—into a single small window. The **hide:** message is usually sent using the Hide command in the application's main menu. Returns **self**.

See also: – **unhide:**

hostName

– (const char *)**hostName**

Returns the name of the host machine on which the Window Server that serves the Application object is running. This method returns the name that was passed to the receiving Application object through the NXHost default; this name is set either from its value in the defaults database or by providing a value for NXHost through the command line. If a value for NXHost isn't specified, NULL is returned.

isActive

– (BOOL)**isActive**

Returns YES if the application is currently active, and NO if it isn't.

See also: – **activateSelf:**, – **activate:**

isHidden

– (BOOL)**isHidden**

Returns YES if the application is currently hidden, and NO if it isn't.

isJournalable

– (BOOL)**isJournalable**

Returns YES if the application can be journaled, and NO if it can't. By default, applications can be journaled.

See also: – **setJournalable:**

isRunning

– (BOOL)**isRunning**

Returns YES if the application is running, and NO if the **stop:** method has ended the main event loop.

See also: – **run**, – **stop:**, – **terminate:**

keyWindow

– **keyWindow**

Returns the key window—the Window that receives keyboard events. If there is no key window, or if the key window belongs to another application, this method returns **nil**.

See also: – **mainWindow**, – **isKeyWindow** (Window)

loadNibFile:owner:

– **loadNibFile:**(const char *)*filename* **owner:***anOwner*

Loads objects from the specified interface file. This method is a cover for **loadNibFile:owner:withNames:fromZone:**. The objects and their names are read from the specified interface file into storage allocated from the default zone. Returns non-**nil** if the file *filename* is successfully opened and read; otherwise it returns **nil**.

See also: – **loadNibFile:owner:withNames:fromZone:**, **NXDefaultMallocZone()**

loadNibFile:owner:withNames:

– **loadNibFile:**(const char *)*filename*
  owner:*anObject*
  withNames:(BOOL)*flag*

Loads objects from the specified interface file. This method is a cover for **loadNibFile:owner:withNames:fromZone:**. The objects are read from the specified interface file into storage allocated from the default zone. Returns non-**nil** if the file *filename* is successfully opened and read; otherwise it returns **nil**.

See also: – **loadNibFile:owner:withNames:fromZone:**, **NXDefaultMallocZone()**

loadNibFile:owner:withNames:fromZone:

– **loadNibFile:**(const char *)*filename*
  owner:*anObject*
  withNames:(BOOL)*flag*
  fromZone:(NXZone *)*zone*

Loads objects from the specified interface file into memory allocated from *zone*. This method returns non-**nil** if the file *filename* is successfully opened and read; otherwise it returns **nil**.

anObject is the object that corresponds to the "File's Owner" object in Interface Builder's File window. As the objects are loaded, the outlet initialization methods in *anObject* are invoked to bind the outlets.

If *flag* is YES, the names of the objects are loaded. If you use only the outlet mechanism to get to objects in the interface file, you can save some memory by specifying NO as the value of *flag*. However, you won't be able to use **NXGetNamedObject()** to get at the objects.

See also: – **loadNibSection:owner:withNames:fromZone:**

loadNibSection:owner:

– **loadNibSection:**(const char *)*sectionName* **owner:***anObject*

Loads objects and their names from the specified section of the application's executable file into memory allocated from the default zone. This method returns non-**nil** if the section is successfully loaded; otherwise it returns **nil**.

See also: – **loadNibSection:owner:withNames:fromZone:**, **NXDefaultMallocZone()**

loadNibSection:owner:withNames:

– **loadNibSection:**(const char *)*name*
  owner:*anObject*
  withNames:(BOOL)*flag*

Loads objects from the interface data in the specified section in the __NIB segment of the executable file into memory allocated from the default zone. This method returns non-**nil** if the section is successfully loaded; otherwise it returns **nil** (for example if section *name* doesn't exist).

See also: – **loadNibSection:owner:withNames:fromZone:**, **NXDefaultMallocZone()**

loadNibSection:owner:withNames:fromHeader:

– **loadNibSection:**(const char *)*name*
  owner:*anObject*
  withNames:(BOOL)*flag*
  fromHeader:(const struct mach_header *)*header*

Loads objects from a dynamically loaded header into memory allocated from the default zone. A class can use this method in its + **finishLoading** method to load associated interface data.

See also: – **loadNibSection:owner:withNames:fromZone:**, **NXDefaultMallocZone()**

loadNibSection:owner:withNames:fromHeader:fromZone:

– **loadNibSection:**(const char *)*name*
  owner:*anObject*
  withNames:(BOOL)*flag*
  fromHeader:(const struct mach_header *)*header*
  fromZone:(NXZone *)*zone*

Loads objects from a dynamically loaded header into memory allocated from the specified zone. A class can use this method in its + **load** method to load associated interface data.

See also: – **loadNibSection:owner:withNames:fromZone:**

loadNibSection:owner:withNames:fromZone:

– **loadNibSection:**(const char *)*name*
  owner:*anObject*
  withNames:(BOOL)*flag*
  fromZone:(NXZone *)*zone*

Loads objects from the interface data in the specified section in the __NIB segment of the executable file into memory allocated from the specified zone. This method returns non-**nil** if the section is successfully loaded; otherwise it returns **nil** (for example if section *name* doesn't exist).

anObject is the object that corresponds to the "File's Owner" object in the Interface Builder's File window. As the objects are loaded, the outlet initialization methods in *anObject* are performed to bind the outlets.

If *flag* is YES, the names of the objects are loaded. If you use only the outlet mechanism to get to objects in the interface section, you can save some memory by specifying NO as the value of *flag*. In that case you won't be able to use **NXGetNamedObject()** to get the **id** of objects.

See also: – **loadNibSection:owner:withNames:fromZone:**

mainMenu

– **mainMenu**

Returns the Application object's main menu.

mainScreen

– (const NXScreen *)**mainScreen**

Returns the main screen. If there is only one screen, that screen is returned. Otherwise, this method attempts to return the key window's screen. If there is no key window, it attempts to return the main menu's screen. If there is no main menu, this method returns the screen that contains the screen coordinate system origin.

See also: – **screen** (Window)

mainWindow

– **mainWindow**

Returns the main window. This method returns **nil** if there is no main window, if the main window belongs to another application, or if the application is hidden.

See also: – **keyWindow**, – **isMainWindow** (Window)

makeWindowsPerform:inOrder:

– **makeWindowsPerform:**(SEL)*aSelector* **inOrder:**(BOOL)*flag*

Sends the Application object's Windows a message to perform the *aSelector* method. The message is sent to each Window in turn until one of them returns YES; this method then returns that Window. If no Window returns YES, this method returns **nil**.

If *flag* is YES, the Application object's Windows receive the *aSelector* message in the front-to-back order in which they appear in the Window Server's window list. If *flag* is NO, Windows receive the message in the order they appear in the Application object's window list. This order generally reflects the order in which the Windows were created.

The *aSelector* method can't take any arguments.

masterJournaler

– **masterJournaler**

Returns the Application object's master journaler.

See also: – **slaveJounaler**

openFile:ok:

– (int)**openFile:**(const char *)*fullPath* **ok:**(int *)*flag*

Responds to a remote message requesting the application to open a file. The **openFile:ok:** message is typically sent to the application from the Workspace Manager, although other applications can send it directly to a specific application. The Application object's delegate is queried with the **appAcceptsAnotherFile:** message and if the result is YES, it's sent the **app:openFile:type:** message. If the delegate doesn't respond to either of these messages, they're sent to the Application object (if it implements them).

The variable pointed to by *flag* is set to YES if the file is successfully opened, NO if the file is not successfully opened, and (–1) if the application does not accept another file. Returns zero.

See also: – **app:openFile:type:** (Application delegate), – **openFile:ok:** (Speaker)

openTempFile:ok:

– (int)**openTempFile:**(const char *)*fullPath* **ok:**(int *)*flag*

Same as the **openFile:ok:** method, but **app:openTempFile:type:** is sent. Returns zero.

See also: – **app:openTempFile:type:** (Application delegate),
– **openTempFile:ok:** (Speaker)

orderFrontColorPanel:

– **orderFrontColorPanel:***sender*

Displays the color panel. Returns **self**.

peekAndGetNextEvent:

– (NXEvent *)**peekAndGetNextEvent:**(int)*mask*

This method is similar to **getNextEvent:waitFor:threshold:** with a zero timeout and a threshold of NX_MODALRESPTHRESHOLD.

See also: – **getNextEvent:waitFor:threshold**, – **run**, – **runModalFor:**, – **currentEvent**

peekNextEvent:into:

– (NXEvent *)**peekNextEvent:**(int)*mask* **into:**(NXEvent *)*eventPtr*

This method is similar to **peekNextEvent:into:waitFor:threshold:** with a zero timeout and a threshold of NX_MODALRESPTHRESHOLD.

See also: – **peekNextEvent:into:waitFor:threshold**, – **run**, – **runModalFor:**, – **currentEvent**

peekNextEvent:into:waitFor:threshold:

– (NXEvent *)**peekNextEvent:**(int)*mask*
    into:(NXEvent *)*eventPtr*
    waitFor:(float)*timeout*
    threshold:(int)*level*

This method is similar to **getNextEvent:waitFor:threshold:** except the matching event isn't removed from the event queue nor is it placed in **currentEvent**; instead, it's copied into storage pointed to by *eventPtr*.

If no matching event is found, NULL is returned; otherwise, *eventPtr* is returned.

See also: – **getNextEvent:waitFor:threshold:**, – **run**, – **runModalFor:**, – **currentEvent**

powerOff:

– **powerOff:**(NXEvent *)*theEvent*

A **powerOff:** message is generated when a power-off event is sent from the Window Server. If the application was launched by the Workspace Manager, this method does nothing; instead, the Application object will wait for the **powerOffIn:andSave:** message from the Workspace Manager. If the application wasn't launched from the Workspace Manager, this method sends the delegate a **powerOff:** message, assuming there's a delegate and it implements the method. Returns **self**.

powerOffIn:andSave:

– (int)**powerOffIn:**(int)*ms* **andSave:**(int)*aFlag*

You never invoke this method directly; it's sent from the Workspace Manager. The delegate or your subclass of Application will be given the chance to receive the **app:powerOffIn:andSave** message. This method raises an exception, so it never returns.

See also: – **app:powerOffIn:andSave:** (delegate)

printInfo

– **printInfo**

Returns the Application object's global PrintInfo object. If none exists, a default one is created.

registerServicesMenuSendTypes:andReturnTypes:

– **registerServicesMenuSendTypes:**(const char *const *)*sendTypes*
  andReturnTypes:(const char *const *)*returnTypes*

Registers pasteboard types that the application can send and receive in response to service requests. If the application has a Services menu, a menu item is added for each service provider that can accept one of the specified send types or return one of the specified return types. This method should typically be invoked at application startup time or when an object that can use services is created. It can be invoked more than once; its purpose is to ensure that there is a menu item for every service that may be used by the application. The individual items will be dynamically enabled and disabled by the event handling mechanism to indicate which services are currently appropriate. An application (or object instance that can cut or paste) should register every possible type that it can send and receive. Returns **self**.

See also: – **validRequestorForSendType:andReturnType:** (Responder),
– **readSelectionFromPasteboard:** (Object method),
– **writeSelectionToPasteboard:** (Object method)

removeWindowsItem:

– **removeWindowsItem:***aWindow*

Removes the item for *aWindow* in the Windows menu. Returns **self**.

See also: – **changeWindowsItem:title:filename:**

replyPort

– (port_t)**replyPort**

Returns the Application object's reply port. This port is allocated for you automatically by the **run** method, and is the default reply port which can be shared by all the Application object's Speakers.

See also: – **setReplyPort:** (Speaker)

resignActiveApp

– **resignActiveApp**

This method is invoked immediately after the application is deactivated. You never send **resignActiveApp** messages directly, but you could override this method in your Application object to notice when your application is deactivated. Alternatively, your delegate could implement **appDidResignActive:**. Returns **self**.

See also: – **deactivateSelf:**, – **appDidResignActive:** (delegate)

rightMouseDown:

– **rightMouseDown:**(NXEvent *)*theEvent*

Pops up the main menu. Returns **self**.

run

– **run**

Initiates the Application object's main event loop. The loop continues until a **stop:** or **terminate:** message is received. Each iteration through the loop, the next available event from the Window Server is stored, and is then dispatched by sending the event to the Application object using **sendEvent:**

A **run** message should be sent as the last statement from **main()**, after the application's objects have been initialized. Returns **self** if terminated by **stop:**, but never returns if terminated by **terminate:**.

See also: – **runModalFor:**, – **sendEvent:**, – **stop:**, – **terminate:**, – **appDidInit:** (delegate)

runModalFor:

– (int)**runModalFor:***theWindow*

Establishes a modal event loop for *theWindow.* Until the loop is broken by a **stopModal**, **stopModal:**, or **abortModal** message, the application won't respond to any mouse, keyboard, or window-close events unless they're associated with *theWindow.* If **stopModal:** is used to stop the modal event loop, this method returns the argument passed to **stopModal:**. If **stopModal** is used, it returns the constant NX_RUNSTOPPED. If **abortModal** is used, it returns the constant NX_RUNABORTED. This method is functionally similar to the following:

```
NXModalSession session;
[NXApp beginModalSession:&session for:theWindow];
for (;;) {
   if ([NXApp runModalSession:&session] != NX_RUNCONTINUES)
         break;
}
[NXApp endModalSession:&session];
```

See also: – **stopModal**, – **stopModal:**, – **abortModal**, – **runModalSession:**

runModalSession:

– (int)**runModalSession:**(NXModalSession *)*session*

Runs a modal session represented by *session,* as defined in a previous invocation of **beginModalSession:for:**. A loop using this method is similar to a modal event loop run with **runModalFor:**, except that with this method the application can continue processing between method invocations. When you invoke this method, events for the window of this session are dispatched as normal; this method returns when there are no more events. You must invoke this method frequently enough that the window remains responsive to events.

If the modal session was not stopped, this method returns NX_RUNCONTINUES. If **stopModal** was invoked as the result of event procession, NX_RUNSTOPPED is returned. If **stopModal:** was invoked, this method returns the value passed to **stopModal:**. The NX_abortModal exception raised by **abortModal** isn't caught.

See also: – **beginModalSession:**, – **endModalSession**, – **stopModal:**, – **stopModal**, – **runModalFor:**

runPageLayout:

– **runPageLayout:***sender*

Brings up the Application object's Page Layout panel, which allows the user to select the page size and orientation. Returns **self**.

sendAction:to:from:

– (BOOL)**sendAction:**(SEL)*aSelector* **to:***aTarget* **from:***sender*

Sends an action message to an object. If *aTarget* is **nil**, the message is sent down the responder chain. Returns YES if the action is applied; otherwise returns NO.

sendEvent:

– **sendEvent:**(NXEvent *)*theEvent*

Sends an event to the Application object. You rarely send **sendEvent:** messages directly although you might want to override this method to perform some action on every event. **sendEvent:** messages are sent from the main event loop (the **run** method). **sendEvent** is the method that dispatches events to the appropriate responders; the Application object handles application events, the Window indicated in the event record handles window related events, and mouse and key events are forwarded to the appropriate Window for further dispatching. Returns **self**.

See also: – **setAutoupdate:**

servicesMenu

– **servicesMenu**

Returns the Application object's Services menu. Returns **nil** if no Services menu has been created.

See also: – **setServicesMenu:**

setAppListener:

– **setAppListener:***aListener*

Sets the Listener that will receive messages sent to the port that's registered for the application. If you want to have a special Listener reply to these messages, you must either send a **setAppListener:** message before the **run** message is sent to the Application object, or send this message from the delegate method **appWillInit:**, so that *aListener* is properly registered. This method doesn't free the Application object's previous Listener object. Returns **self**.

See also: – **appListenerPortName**, – **appWillInit:** (delegate)

setAppSpeaker:

– **setAppSpeaker:***aSpeaker*

Sets the Application object's Speaker. If you don't send a **setAppSpeaker:** message before the Application object initializes, a default Speaker is created for you. This method doesn't free the Application object's previous Speaker object.

See also: – **appWillInit:** (delegate)

setAutoupdate:

– **setAutoupdate:**(BOOL)*flag*

Turns on or off automatic updating of windows. If automatic updating is on, **update** is sent to each of the application's Windows after each event has been processed. This can be used to keep the appearance of menus and panels synchronized with your application. Returns **self**.

setDelegate:

– **setDelegate:***anObject*

Sets the Application object's delegate. The notification messages that a delegate can expect to receive are listed at the end of the Application class specifications. The delegate doesn't need to implement all the methods. Returns **self**.

See also: – **delegate**

setJournalable:

– **setJournalable:**(BOOL)*flag*

Sets whether the application is journalable. Returns **self**.

setMainMenu:

– **setMainMenu:***aMenu*

Makes *aMenu* the Application object's main menu. Returns **self**.

See also: – **mainMenu**

setPrintInfo:

– **setPrintInfo:***info*

Sets the Application object's global PrintInfo object. Returns the previous PrintInfo object, or **nil** if there was none.

setServicesMenu:

– **setServicesMenu:***aMenu*

Makes *aMenu* the Application object's Services menu. Returns **self**.

setWindowsMenu:

– **setWindowsMenu:***aMenu*

Makes *aMenu* the Application object's Windows menu. Returns **self**.

slaveJournaler

– **slaveJournaler**

Returns the Application object's slave journaler.

stop:

– **stop:***sender*

Stops the main event loop. This method will break the flow of control out of the **run** method, thereby returning to the **main()** function. A subsequent **run** message will restart the loop.

If this method is applied during a modal event loop, it will break that loop but not the main event loop. Returns **self**.

See also: – **terminate:**, – **run**, – **runModalFor:**, – **runModalSession:**

stopModal

– **stopModal**

Stops a modal event loop. This method should always be paired with a previous **runModalFor:** or **beginModalSession:for:** message. When **runModalFor:** is stopped with this method, it returns NX_RUNSTOPPED. This method will stop the loop only if it's executed by code responding to an event. If you need to stop a **runModalFor:** loop from a procedure registered with **DPSAddTimedEntry()**, **DPSAddPort()**, or **DPSAddFD()**, use the **abortModal** method. Returns **self**.

See also: – **runModalFor:**, – **runModalSession:**, – **abortModal**

stopModal:

– **stopModal:**(int)*returnCode*

Just like **stopModal** except argument *returnCode* allows you to specify the value that **runModalFor:** will return. Returns **self**.

See also: – **stopModal**, – **runModalFor:**, – **abortModal**

systemLanguages

– (const char *const *)**systemLanguages**

Returns a NULL-terminated list of NULL-terminated strings which specify the user's preferred languages (human languages, not computer languages) in order of preference. If this method returns NULL, the user has no preference. This should be used to do any localization of your application.

terminate:

– **terminate:***sender*

Terminates the application. This method invokes **appWillTerminate:** to notify the delegate that the application will terminate. If **appWillTerminate:** returns **nil**, **terminate:** returns **self**; control is returned to the main event loop, and the application isn't terminated. Otherwise, this method frees the Application object and terminates the application by using **exit()**. **terminate:** is the default action method for the application's "Quit" menu item. Note that you should not put final cleanup code in your application's **main()** function; it will never be executed.

See also: – **stop**, – **appWillTerminate:** (delegate), **exit()**

tryToPerform:with:

– (BOOL)**tryToPerform:**(SEL)*aSelector* **with:***anObject*

Aids in dispatching action messages. The Application object tries to perform the method selector *aSelector* using its inherited Responder method **tryToPerform:with:**. If the Application object doesn't perform *aSelector*, the delegate is given the opportunity to perform it using its inherited Object method **perform:with:**. If either the Application object or the Application object's delegate accept *aSelector*, this method returns YES; otherwise it returns NO.

See also: – **tryToPerform:with:** (Responder), – **respondsTo:** (Object),
– **perform:with:** (Object)

unhide

– (int)**unhide**

Responds to an **unhide** message sent from Workspace Manager. You shouldn't invoke this method; invoke **unhide:** instead. Returns zero.

See also: – **unhide:**

unhide:

– **unhide:***sender*

Restores a hidden application to its former state (all of the windows, menus, and panels visible), and makes it the active application. This method is usually invoked as the result of double-clicking in the icon for the hidden application. Returns **self**.

See also: – **hide:**, – **unhideWithoutActivation:**, – **activateSelf:**

unhideWithoutActivation:

– **unhideWithoutActivation:***sender*

Unhides the application but does not make it the active application. You might want to invoke **activateSelf:**NO after invoking this method to make the receiving application active if there is no active application. Returns **self**.

See also: – **hide:**, – **activateSelf:**

unmounting:ok:

– (int)**unmounting:**(const char *)*fullPath* **ok:**(int *)*flag*

Replies to an **unmounting:ok:** message sent from the Workspace Manager. You shouldn't directly send **unmounting:ok:** messages. This method attempts to invoke the **app:unmounting:** method of the Application object's delegate or of the Application object itself. If neither object implements **app:unmounting:**, and the current working directory is on the same volume as *fullPath*, this method changes the working directory to the user's home directory. Returns zero.

updateWindows

– **updateWindows**

Sends an **update** message to the Application object's visible Windows. If automatic updating is enabled, this method is invoked automatically in the main event loop after each event. An application can also send **updateWindows** messages at other times to have Windows update themselves.

If the delegate implements **appWillUpdate:**, that message is sent to the delegate before the windows are updated. Similarly, if the delegate implements **appWillUpdate:**, that message is sent to the delegate after the windows are updated. Returns **self**.

See also: – **setAutoupdate:**, – **appWillUpdate:** (delegate), – **appDidUpdate:** (delegate)

updateWindowsItem:

– **updateWindowsItem:***win*

Updates the item for *aWindow* in the Windows menu to reflect the edited status of *aWindow.* You rarely need to invoke this method because it is invoked automatically when the edited status of a Window is set. Returns **self**.

See also: – **changeWindowsItem:title:filename:**, – **setDocEdited:** (Window)

validRequestorForSendType:andReturnType:

– **validRequestorForSendType:**(NXAtom)*sendType*
    andReturnType:(NXAtom)*returnType*

Passes this message on to the Application object's delegate, if the delegate can respond (and isn't a Responder with its own next responder). If the delegate can't respond or returns **nil**, this method returns **nil**, indicating that no object was found that could supply *typeSent* data for a remote message from the Services menu and accept back *typeReturned* data. If such an object was found, it is returned.

Messages to perform this method are initiated by the Services menu. This method might not be in the Application class header file at this time.

See also: – **validRequestorForSendType:andReturnType:** (Responder), – **registerServicesMenuSendTypes:andReturnTypes:**, – **writeSelectionToPasteboard:types:** (Object Method), – **readSelectionFromPasteboard:** (Object Method)

windowList

– **windowList**

Returns the List object used to keep track of the Application object's Windows.

windowsMenu

– **windowsMenu**

Returns the Application object's Windows menu. Returns **nil** if no Windows menu has been created.

app:openFile:type:

– (int)**app:***sender* **openFile:**(const char *)*filename* **type:**(const char *)*aType*

Invoked from within **openFile:ok:** after it has been determined that the application can open another file. The method should attempt to open the file *filename* with the extension *aType*, returning YES if the file is successfully opened, and NO otherwise.

This method is also invoked from within **openTempFile:ok:** if neither the delegate nor the Application subclass responds to **app:openTempFile:type:**

See also: – **openFile:ok:**, – **openTempFile:ok:**

app:openTempFile:type:

– (int)**app:***sender* **openTempFile:**(const char *)*filename* **type:**(const char *)*aType*

Invoked from within **openTempFile:ok:** after it has been determined that the application can open another file. The method should attempt to open the file *filename* with the extension *aType*, returning YES if the file is successfully opened, and NO otherwise.

By design, a file opened through this method is assumed to be temporary; it's the application's responsibility to remove the file at the appropriate time.

See also: – **openTempFile:ok:**

app:powerOffIn:andSave:

– **app:***sender* **powerOffIn:**(int)*ms* **andSave:**(int)*aFlag*

Invoked when the Application object receives a power-off event through the **powerOffIn:andSave:** method. This method is invoked only if the application was launched from the Workspace Manager. *ms* is the number of milliseconds to wait before powering down or logging out. *aFlag* has no particular meaning at this time. You can ask for additional time by sending the **extendPowerOffBy:actual:** message to the Workspace Manager. The Workspace Manager will power the machine down (or log out the user) as soon as all applications terminate, even if there's time remaining on the time extension.

See also: – **extendPowerOffBy:actual:** (Speaker)

app:unmounting:

– (int)**app:***sender* **unmounting:**(const char *)*fullPath*

Invoked when the device mounted at *fullPath* is about to be unmounted. This method is invoked from **unmounting:ok:** and is invoked only if the application was launched from the Workspace Manager. The Application object or its delegate should do whatever is necessary to allow the device to be unmounted. Specifically, all files on the device should be closed and the current working directory should be changed if it's on the device.

appAcceptsAnotherFile:

– (BOOL)**appAcceptsAnotherFile:***sender*

Invoked from within Application's **openFile:ok:** and **openTempFile:ok:** methods, this method should return YES if it's okay for the application to open another file, and NO if isn't. If neither the delegate nor the Application object responds to the message, then the file shouldn't be opened.

See also: – **openFile:ok:**, – **openTempFile:ok:**

appDidBecomeActive:

– **appDidBecomeActive:***sender*

Invoked immediately after the application is activated.

appDidHide:

– **appDidHide:***sender*

Invoked immediately after the application is hidden.

appDidInit:

– **appDidInit:***sender*

Invoked after the application has been launched and initialized, but before it has received its first event. The delegate or the Application subclass can implement this method to perform further initialization.

See also: – **appWillInit:** (delegate)

appDidResignActive:

– **appDidResignActive:***sender*

Invoked immediately after the application is deactivated.

appDidUnhide:

– **appDidUnhide:***sender*

Invoked immediately after the application is unhidden.

appDidUpdate:

– **appDidUpdate:***sender*

Invoked immediately after the Application object updates its Windows.

applicationDefined:

– **applicationDefined:**(NXEvent *)*theEvent*

Invoked when the application receives an application-defined (NX_APPDEFINED) event. See the description of this method under INSTANCE METHODS, above.

appWillInit:

– **appWillInit:***sender*

Invoked before the Application object is initialized. This method is invoked before the Application object has initialized its Listener and Speaker objects and before any **app:openFile:type:** messages are sent to your delegate. The Application object's Listener and Speaker objects will be created for you immediately after invoking this method if they have not been previously created.

See also: – **appDidInit:** (delegate), – **appListener**, – **appSpeaker**

appWillTerminate:

– **appWillTerminate:***sender*

Invoked from within the **terminate:** method immediately before the application terminates. If this method returns **nil**, the application is not terminated, and control is returned to the main event loop. If you want to allow the application to terminate, you should put your clean up code in this method and return non-**nil**.

See also: – **terminate:**

appWillUpdate:

– **appWillUpdate:***sender*

Invoked immediately before the Application object updates its Windows.

powerOff:

– **powerOff:**(NXEvent *)*theEvent*

Invoked when the Application object receives a power-off event through the **powerOff:** method. Note that **powerOff:** (and so, too, this method) is invoked only if the application *wasn't* launched from the Workspace Manager.

CONSTANTS AND DEFINED TYPES

```
/* KITDEFINED subtypes */
#define NX_WINEXPOSED    0
#define NX_APPACT        1
#define NX_APPDEACT      2
#define NX_WINRESIZED    3
#define NX_WINMOVED      4
#define NX_SCREENCHANGED 8

/* SYSDEFINED subtypes */
#define NX_POWEROFF      1

/* Additional flags */
#define NX_JOURNALFLAG  31
#define NX_JOURNALFLAGMASK (1 << NX_JOURNALFLAG)

/* Thresholds passed to DPSGetEvent() and DPSPeekEvent(). */
#define NX_BASETHRESHOLD       1
#define NX_RUNMODALTHRESHOLD   5
#define NX_MODALRESPTHRESHOLD 10

/*
 * Predefined return values for runModalFor: and
 * runModalSession:.  All values below these (-1003, -1004, and
 * so on) are also reserved.
 */
#define NX_RUNSTOPPED    (-1000)
#define NX_RUNABORTED    (-1001)
#define NX_RUNCONTINUES  (-1002)
```

```
/*
 * The NXModalSession structure contains information used by the
 * system between beginModalSession:for: and endModalSession:
 * messages.  This structure can either be allocated on the stack
 * frame of the caller, or by beginModalSession:for:.  The
 * application should not access any of the elements of this
 * structure.
 */

typedef struct _NXModalSession {
    id app;
    id window;
    struct _NXModalSession *prevSession;
    int oldRunningCount;
    BOOL oldDoesHide;
    BOOL freeMe;
    int winNum;
    NXHandler *errorData;
    int reserved1;
    int reserved2;
} NXModalSession;
```

Box

INHERITS FROM	View : Responder : Object
DECLARED IN	appkit/Box.h

CLASS DESCRIPTION

A Box is a View that visually groups other Views. A Box has one subview, its *content view*, which is used to group the Box's contents. A Box also typically displays a title and a border around its content view. The Box class includes methods to change the Box's border style and title position, and to set the text and font of the title. In addition, you can add subviews to the Box's content view and then resize the Box to fit around these subviews.

INSTANCE VARIABLES

```
Inherited from Object       Class              isa;

Inherited from Responder    id                 nextResponder;

Inherited from View         NXRect             frame;
                            NXRect             bounds;
                            id                 superview;
                            id                 subviews;
                            id                 window;
                            struct __vFlags    vFlags;

Declared in Box             id                 cell;
                            id                 contentView;
                            NXSize             offsets;
                            NXRect             borderRect;
                            NXRect             titleRect;
                            struct _bFlags {
                                unsigned int       borderType:2;
                                unsigned int       titlePosition:3;
                                unsigned int       transparent:1;
                            }                  bFlags;
```

cell	The cell that draws the Box's title.
contentView	The Box's subview that contains the Views that are grouped within the Box.
offsets	Offset of the content view from the Box's border.
borderRect	The Box's border rectangle.

titleRect	The location of the title cell.
bFlags.borderType	Indicates the Box's border type.
bFlags.titlePosition	Indicates the Box's title position.
bFlags.transparent	Reserved. Do not use.

METHOD TYPES

Initializing a new Box object	– initFrame:
Freeing a Box object	– free
Modifying graphic attributes	– setBorderType: – borderType – setOffsets:: – getOffsets:
Modifying the title	– cell – setFont: – font – setTitle: – title – setTitlePosition: – titlePosition
Putting Views in the Box	– addSubview: – setContentView: – contentView
Resizing the Box	– setFrameFromContentFrame: – sizeTo:: – sizeToFit
Drawing the Box	– drawSelf::
Archiving	– awake – read: – write:

INSTANCE METHODS

addSubview:

– **addSubview:***aView*

Adds *aView* as a subview of the Box's content view. Since the content view is a subview of the Box, the frame rectangles of Views added to the Box should reflect their position within the content rectangle rather than the Box's bounds rectangle. After you've added a subview, you'll probably want to use the **sizeToFit** method to adjust the Box's size to accommodate its new subview. Returns **self.**

See also: – **sizeToFit**

awake

– **awake**

Lays out the Box during the unarchiving process so that it can be displayed. You should never directly invoke this method.

borderType

– (int)**borderType**

Returns the Box's border type, which is NX_LINE, NX_GROOVE, NX_BEZEL, or NX_NOBORDER.

See also: – **setBorderType:**

cell

– **cell**

Returns the cell used to display the title of the Box.

contentView

– **contentView**

Returns the Box's content view.

See also: – **setContentView:**

drawSelf::

– **drawSelf:**(const NXRect *)*rects* **:**(int)*rectCount*

Draws the Box. You never invoke this method directly; it's invoked from Box's inherited display methods. Returns **self**.

See also: – **display** (View)

font

– **font**

Returns the **id** of the font object used to draw the title of the Box.

See also: – **setFont:**

free

– **free**

Releases the storage for the Box and all its subviews.

See also: – **free** (View)

getOffsets:

– **getOffsets:**(NXSize *)*theSize*

Gets the horizontal and vertical distances between the border of the Box and the content view, and places them in the structure indicated by *theSize*. Returns **self**.

See also: – **setOffsets::**

initFrame:

– **initFrame:**(const NXRect *)*frameRect*

Initializes the Box, which must be a newly allocated Box instance. The Box's frame rectangle is made equivalent to that pointed to by *frameRect*. The title is "Title," the border type is NX_GROOVE, the title position is NX_ATTOP, and the offsets are 5.0-by-5.0. The Box's content view is created, but it has no size; you will probably want to set its size with the **sizeToFit** method. This method is the designated initializer for the Box class, and can be used to initialize a Box allocated from your own zone. Returns **self**.

See also: – **initFrame** (View), + **alloc** (Object), + **allocFromZone:** (Object), – **addSubview:**, – **sizeToFit**

read:

– **read:**(NXTypedStream *)*stream*

Reads the Box from the typed stream *stream*. Returns **self**.

See also: – **write:**

setBorderType:

– **setBorderType:**(int)*aType*

Sets the border type to *aType*, which must be NX_LINE, NX_GROOVE, NX_BEZEL, or NX_NOBORDER. The default is NX_GROOVE. Returns **self**.

See also: – **borderType**

setContentView:

– **setContentView:***aView*

Replaces the Box's content view with *aView* and recalculates the size of the Box based on the size of the new content view. The old content view is returned.

See also: – **addSubview:**, – **contentView**, – **sizeToFit**

setFont:

– **setFont:***fontObj*

Sets the title's font to *fontObj*. By default, the title will be displayed using 12-point Helvetica.

See also: + **newFont:size:** (Font)

setFrameFromContentFrame:

– **setFrameFromContentFrame:**(const NXRect *)*contentFrame*

Resizes the Box so that its content view lies on *contentFrame*. *contentFrame* is in the coordinate system of the Box's superview. Returns **self**.

See also: – **setOffsets::**, – **setFrame:** (View)

setOffsets::

– **setOffsets:**(NXCoord)*w* **:**(NXCoord)*h*

Sets the horizontal and vertical distance between the border of the Box and its content view. *w* refers to the horizontal offset and *h* refers to the vertical offset; these offsets are applied to both sides of the content view. After changing the offsets, you'll want to resize the Box using the **setFrameFromContentFrame:** method. This method returns **self**. In the following example, the offsets are modified but the content view's size and location within the Box's superview remain unchanged:

```
id contentView;
NXRect contentRect;
NXCoord w = 10.0, h = 10.0;

contentView = [myBox contentView];
[contentView getFrame:&contentRect];
[myBox convertRectToSuperview:&contentRect];

[myBox setOffsets:w :h];
[myBox setFrameFromContentFrame:&contentRect];
```

See also: – **setFrameFromContentFrame:**, – **convertRectToSuperview:** (View)

setTitle:

– **setTitle:**(const char *)*aString*

Sets the title to *aString*. The default title is "Title." Returns **self**.

See also: – **setFont:**

setTitlePosition:

– **setTitlePosition:**(int)*aPosition*

Sets the title position to *aPosition*, which can be one of the values listed in the following table. The default position is NX_ATTOP. Returns **self**.

***aPosition* value**	**Meaning**
NX_NOTITLE	The Box has no title
NX_ABOVETOP	Title positioned above the Box's top border
NX_ATTOP	Title positioned within the Box's top border
NX_BELOWTOP	Title positioned below the Box's top border
NX_ABOVEBOTTOM	Title positioned above the Box's bottom border
NX_ATBOTTOM	Title positioned within the Box's bottom border
NX_BELOWBOTTOM	Title positioned below the Box's bottom border

sizeTo::

– **sizeTo:**(NXCoord)*width* **:**(NXCoord)*height*

Resizes the Box to *width* and *height*. The Box is laid out to fit inside this new boundary. If the new width or height of the Box is too small to accommodate its border or offsets, the respective dimension of the content view will be zero. Returns **self**.

See also: – **setFrameFromContentFrame:**, – **getOffsets:**

sizeToFit

– **sizeToFit**

Calculates the appropriate size for the Box's content rectangle so that it just encloses all the content view's subviews. A **setFrameFromContentFrame:** message is then sent to resize the Box to enclose the new content rectangle. Returns **self**.

See also: – **setFrameFromContentFrame:**

title

– (const char *)**title**

Returns the title of the Box.

See also: – **setTitle:**

titlePosition

– (int)**titlePosition**

Returns an integer representing the title position. See the description for **setTitlePosition:** for possible title position values.

See also: – **setTitlePosition:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving Box to the typed stream *stream*. Returns **self**.

See also: – **read:**

Button

INHERITS FROM	Control : View : Responder : Object
DECLARED IN	appkit/Button.h

CLASS DESCRIPTION

A Button is a Control subclass that intercepts mouse-down events and sends an action message to a target object whenever the Button is pressed.

Button essentially provides the Control view needed to display a ButtonCell object. Most of its methods simply delegate to the same method in ButtonCell. To change the look or behavior of a Button, create a subclass of ButtonCell and use the method **setCellClass:** to get the Button class to use it.

Buttons can display any NXImage object. The icon methods **altIcon**, **icon**, **setAltIcon:**, and **setIcon:** are provided for use with named images. The corresponding image methods **altImage**, **image**, **setAltImage:**, and **setImage:** are provided for use with the **id**s of image objects.

The **initFrame:icon:tag:target:action:key:enabled:** method is the designated initializer for Buttons that display icons. Buttons that display text have the designated initializer **initFrame:text:tag:target:action:key:enabled:**. Override one of these methods if you create a subclass of Button that performs its own initialization.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from View	NXRect	frame;
	NXRect	bounds;
	id	superview;
	id	subviews;
	id	window;
	struct __vFlags	vFlags;
Inherited from Control	int	tag;
	id	cell;
	struct _conFlags	conFlags;
Declared in Button	(none)	

METHOD TYPES

Setting Button's Cell Class	+ setCellClass:
Initializing a Button Instance	– init
	– initFrame:
	– initFrame:icon:tag:target:action:key:enabled:
	– initFrame:title:tag:target:action:key:enabled:
Setting the Button Type	– setType:
Setting the State	– setState:
	– state
Setting Button Repeat	– getPeriodicDelay:andInterval:
	– setPeriodicDelay:andInterval:
Modifying the Title	– altTitle
	– setAltTitle:
	– setTitle:
	– setTitleNoCopy:
	– title
Modifying the Icon	– altIcon
	– altImage
	– icon
	– image
	– iconPosition
	– setAltIcon:
	– setAltImage:
	– setIcon:
	– setImage:
	– setIcon:position:
	– setIconPosition:
Modifying Graphic Attributes	– isBordered
	– isTransparent
	– setBordered:
	– setTransparent:
Displaying	– display
	– highlight:
Handling Events and Action Messages	
	– acceptsFirstMouse
	– keyEquivalent
	– performClick:
	– performKeyEquivalent:
	– setKeyEquivalent:

Setting the Sound	– setSound: – sound

CLASS METHODS

setCellClass:

+ **setCellClass:***classId*

Initializes the Button to work with a subclass of ButtonCell. The *classId* will usually be the value returned by the message [myButtonCell class], where myButtonCell is an instance of the subclass. Returns **self**.

INSTANCE METHODS

acceptsFirstMouse

– (BOOL)**acceptsFirstMouse**

Returns YES. Buttons always accept the mouse-down event that activates a Window.

altIcon

– (const char *)**altIcon**

Returns the Button's alternate icon by name. This icon will appear on the Button when it's in its alternate state.

altImage

– altImage

Returns the Button's alternate icon by **id**. This image will appear on the Button when it's in its alternate state.

altTitle

– (const char *)**altTitle**

Returns the current value of the Button's alternate title. This is the string that appears on the Button when it's in its alternate state.

display

– display

Overridden from View so that **displayFromOpaqueAncestor:::** is called if the Button has some non-opaque parts. Returns **self**.

getPeriodicDelay:andInterval:

– **getPeriodicDelay:**(float *)*delay* **andInterval:**(float *)*interval*

This method returns **self** explicitly and two values by reference. *delay* returns the amount of time (in seconds) that a continuous button will pause before starting to periodically send action messages to the target object. *interval* returns the amount of time (also in seconds) between those messages.

See also: – **setContinuous:** (Control), – **setPeriodicDelay:andInterval:**

highlight:

– **highlight:**(BOOL)*flag*

If the highlighted flag of the cell is not equal to *flag*, the Button is highlighted and the highlighted flag of the cell is set to *flag*. Issues a **flushWindow** after highlighting the Button. Returns **self**.

See also: – **performClick:**

icon

– (const char *)**icon**

Returns the Button's icon by name.

iconPosition

– (int)**iconPosition**

Returns a constant representing the position of the icon on the Button. See **setIconPosition:** for the list of position constants.

image

– **image**

Returns the **id** of the Button's icon.

See also: – **altImage**, – **setAltIcon:**, – **setAltImage:**

init

– **init**

Initializes and returns the receiver, a new Button instance. The new instance displays the word "Button" and has no icon associated with it. You usually invoke **initFrame:{title,icon}:tag:target:action:key:enabled:** to initialize a Button.

initFrame:

– **initFrame:**(const NXRect *)*frameRect*

Initializes and returns the receiver, a new Button instance, with default parameters in the given frame. The default title is "Button," the default action is NULL and the default target is **nil**. You usually invoke **initFrame:{title,icon}:tag:target:action:key:enabled:** to initialize a Button.

initFrame:icon:tag:target:action:key:enabled:

– **initFrame:**(const NXRect *)*frameRect*
  icon:(const char *)*aString*
  tag:(int)*anInt*
  target:*anObject*
  action:(SEL)*aSelector*
  key:(unsigned short)*charCode*
  enabled:(BOOL)*flag*

Initializes and returns the receiver, a new Button instance that displays an icon. The arguments and operation of this method are exactly like those of **initFrame:title:tag:target:action:key:enabled:**, except that the Button displays the named icon represented by *aString* rather than displaying a text string. This method is the designated initializer for Buttons that display icons.

initFrame:title:tag:target:action:key:enabled:

– **initFrame:**(const NXRect *)*frameRect*
  title:(const char *)*aString*
  tag:(int)*anInt*
  target:*anObject*
  action:(SEL)*aSelector*
  key:(unsigned short)*charCode*
  enabled:(BOOL)*flag*

Initializes and returns the receiver, a new Button instance that displays a text string. *anInt* is a unique tag to identify your Button View. *frameRect* is the rectangle the Button will occupy in its superview's coordinates. *aString* contains the title for the Button. *anObject* is the target that will be notified via the action message *aSelector* when the Button is successfully pressed. If *anObject* is **nil**, the target will default to the Button's superview. *aSelector* should be a valid selector. *charCode* is the key equivalent for this Button. *flag* determines whether your Button is initially enabled. This method is the designated initializer for Buttons that display text.

isBordered

– (BOOL)**isBordered**

Returns YES if the Button has a border, NO otherwise.

See also: – **setBordered:**

isTransparent

– (BOOL)**isTransparent**

Returns YES if the Button is transparent, NO otherwise.

See also: – **setTransparent:**

keyEquivalent

– (unsigned short)**keyEquivalent**

Returns the key equivalent character of the Button.

See also: – **performKeyEquivalent:**

performClick:

– **performClick:***sender*

Highlights the Button, sends its action message to the target object, then unhighlights the Button. Invoke this method when you want the Button to behave exactly as if the user had clicked it with the mouse.

performKeyEquivalent:

– (BOOL)**performKeyEquivalent:**(NXEvent *)*theEvent*

Simulates the user clicking the Button and returns YES if the character in the event record matches the Button's key equivalent. Otherwise, does nothing and returns NO.

See also: – **keyEquivalent**

setAltIcon:

– **setAltIcon:**(const char *)*iconName*

Sets the Button's alternate icon by name; *iconName* is the name of an image to be displayed. Does not display the Button even if autodisplay is on.

See also: – **setIcon:**

setAltImage:

– **setAltImage:***altImage*

Sets the Button's alternate icon by **id**; *altImage* is the **id** of the image to be displayed. Does not display the Button even if autodisplay is on.

See also: – **setImage:**

setAltTitle:

– **setAltTitle:**(const char *)*aString*

Sets the alternate title of your Button to *aString*, the title that will display when the Button is clicked. Does not display the Button even if autodisplay is on.

setBordered:

– **setBordered:**(BOOL)*flag*

If *flag* is YES, the Button displays a border; if NO, no border is displayed. This method redraws the Button if the bordered state is changed. Returns **self**.

setIcon:

– **setIcon:**(const char *)*iconName*

Sets the Button's icon by name; *iconName* is the name of an image to be displayed. Returns **self**.

See also: – **getBitmapFor:** (Bitmap)

setIcon:position:

– **setIcon:**(const char *)*iconName* **position:**(int)*aPosition*

Combines **setIcon:** and **setIconPosition:** into one message. Returns **self**.

setIconPosition:

– **setIconPosition:**(int)*aPosition*

Sets the position of the icon when a Button simultaneously displays both text and an icon. *aPosition* can be one of the following constants:

NX_TITLEONLY	title only (no icon on the Button)
NX_ICONONLY	icon only (no text on the Button)
NX_ICONLEFT	icon is to the left of the text
NX_ICONRIGHT	icon is to the right of the text
NX_ICONBELOW	icon is below the text
NX_ICONABOVE	icon is above the text
NX_ICONOVERLAPS	icon and text overlap

If the position is top or bottom, the alignment of the text will be set to NX_CENTERED. This behavior can be overridden with a subsequent **setAlignment:**. Returns **self**.

setImage:

– **setImage:***image*

Sets the Button's icon by **id**; *image* is the **id** of the image to be displayed. Returns **self**.

See also: + **findImageNamed:**(NXImage)

setKeyEquivalent:

– **setKeyEquivalent:**(unsigned short)*charCode*

Sets the key equivalent character of the Button. Returns **self**.

See also: – **keyEquivalent**, – **performKeyEquivalent:**

setPeriodicDelay:andInterval:

– **setPeriodicDelay:**(float)*delay* **andInterval:**(float)*interval*

Sets two values that are in effect if the Button is set to continuously send the action message to the target object while tracking the mouse. *delay* is the amount of time (in seconds) that a continuous button will pause before starting to periodically send action messages to the target object. *interval* is the amount of time (also in seconds) between those messages. Returns **self**.

See also: – **getPeriodicDelay:andInterval:**, – **setContinuous**(Control)

setSound:

– **setSound:***soundObj*

Sets the sound played when the Button is pressed. Returns **self**.

setState:

– **setState:**(int)*value*

Sets the Button's state to *value* and redraws the Button. Returns **self**.

setTitle:

– **setTitle:**(const char *)*aString*

Sets the title of the Button to *aString*. Returns **self**.

setTitleNoCopy:

– **setTitleNoCopy:**(const char *)*aString*

Similar to **setTitle:** but does not make a copy of *aString*. Returns **self**.

setTransparent:

– **setTransparent:**(BOOL)*flag*

Sets whether the Button is transparent. A transparent Button tracks the mouse and sends its action, but it doesn't draw anything. Returns **self**.

setType:

– **setType:**(int)*aType*

Sets the way the Button shows its state and highlighting, and returns **self**. *aType* can be one of five constants:

NX_MOMENTARYPUSH (the default). States 0 and 1 are displayed in the same manner. Highlighting is shown by the Button's "pushing in" to the screen.

NX_MOMENTARYCHANGE. States 0 and 1 look identical. When the Button is highlighted, the alternate icon or alternate text will be displayed. The miniaturize Button in the window frame is a good example of this type of Button.

NX_PUSHONPUSHOFF. State 1 differs from state 0 by the fact that different colors are used. Highlighting is achieved by "pushing in."

NX_TOGGLE. State 1 uses the altContents and/or altIcon. Highlighting is performed by "pushing in."

NX_SWITCH. A variant of NX_TOGGLE that has no border, and that has a default icon called "switch."

sound

– **sound**

Returns the sound played when the button is pressed.

state

– (int)**state**

Returns the Button's state (0 or 1).

title

– (const char *)**title**

Returns a pointer to the current string value of the Button's title.

ButtonCell

INHERITS FROM ActionCell : Cell : Object

DECLARED IN appkit/ButtonCell.h

CLASS DESCRIPTION

The ButtonCell class is a subclass of Cell that is used to implement Button. Different modes of button operation are distinguished according to the values of the changeXXX and lightByXXX bitfields.

changeXXX refers to what changes when the state changes. Thus, if **changeGray** is set, then, when a button is in state 1, all light gray areas in the button become white, and all white areas become light gray. If **changeBackground** is set, then the background in state 1 is white instead of the default light gray used in state 0. If **changeContents** is set, then **altContents** and/or icon.bmap.alternate are used to draw the button when it is in state 1. If both **changeBackground** and **changeGray** are set, then the ButtonCell will use **changeGray** unless the ButtonCell has an icon and alpha values, in which case it will use **changeBackground**. The lightByXXX flags have similar meanings, but are used when the button is pressed to highlight the button. The **pushIn** flag is used to determine whether the button appears to "push in" to the screen when pressed. This only has meaning when the bordered flag is set.

For all ButtonCells, the "default" icon is the keyEquivalent for the button. Therefore, if you want the button to display its keyEquivalent, just use **setIconPosition:** to determine where on the button the keyEquivalent should appear. MenuCells use this, for example (by issuing a **setIconPosition:**NX_ICONRIGHT). If you set an icon (or an altIcon) for the button, then the icon will be displayed instead of the keyEquivalent, so if you want the keyEquivalent, don't invoke **setIcon:**!

ButtonCells can display any type of image. The icon methods **altIcon**, **icon**, **setAltIcon:**, and **setIcon:** work with named images. The corresponding image methods **altImage**, **image**, **setAltImage:**, and **setImage:** work with **id**s of image objects.

The **initIconCell:** method is the designated initializer for ButtonCells that display icons. The **initTextCell:** method is the designated initializer for ButtonCells that display text. Override one of these methods if you create a subclass of ButtonCell that does its own initialization.

INSTANCE VARIABLES

```
Inherited from Object        Class               isa;

Inherited from Cell          char                *contents;
                             id                  support;
                             struct _cFlags1     cFlags1;
                             struct _cFlags2     cFlags2;

Inherited from ActionCell    int                 tag;
                             id                  target;
                             SEL                 action;

Declared in ButtonCell       char                *altContents;
                             union _icon {
                               struct _bmap {
                                 id                  normal;
                                 id                  alternate;
                               }                   bmap;
                               struct _ke {
                                 id                  font;
                                 float               descent;
                               }                   ke;
                             }                   icon;
                             id                  sound;
                             struct _bcFlags1 {
                               unsigned int        pushIn:1;
                               unsigned int        changeContents:1;
                               unsigned int        changeBackground:1;
                               unsigned int        changeGray:1;
                               unsigned int        lightByContents:1;
                               unsigned int        lightByBackground:1;
                               unsigned int        lightByGray:1;
                               unsigned int        hasAlpha:1;
                               unsigned int        bordered:1;
                               unsigned int        iconOverlaps:1;
                               unsigned int        horizontal:1;
                               unsigned int        bottomOrLeft:1;
                               unsigned int        iconAndText:1;
                               unsigned int        lastState:1;
                               unsigned int        iconSizeDiff:1;
                               unsigned int        iconIsKeyEquivalent:1;
                             }                   bcFlags1;
                             struct _bcFlags2 {
                               unsigned int        keyEquivalent:8;
                               unsigned int        transparent:1;
                             }                   bcFlags2;
                             unsigned short      periodicDelay;
                             unsigned short      periodicInterval;
```

altContents	Alternate contents used instead of contents in certain state configurations.
bmap.normal	Name of the icon for this button.
bmap.alternate	Name of the alternate icon.
ke.font	Font used to draw the key equivalent.
ke.descent	The descent of descenders in the key equivalent font.
sound	The button's sound.
bcFlags1.pushIn	Button appears to push into the screen when pressed.
bcFlags1.changeContents	Show alternate state by using alternate contents.
bcFlags1.changeBackground	Show alternate state by changing the background.
bcFlags1.changeGray	Show alternate state by inverting the button.
bcFlags1.lightByContents	Show highlighting by using alternate contents.
bcFlags1.lightByBackground	Show highlighting by changing the background.
bcFlags1.lightByGray	Show highlighting by inverting the button.
bcFlags1.hasAlpha	Icon has alpha values.
bcFlags1.bordered	Button has border.
bcFlags1.iconOverlaps	Icon overlaps text.
bcFlags1.horizontal	Icon to side of text.
bcFlags1.bottomOrLeft	Icon on left or bottom.
bcFlags1.iconAndText	Button has icon *and* text.
bcFlags1.lastState	Last state drawn.
bcFlags1.iconSizeDiff	Alternate icon is a different size than the normal icon.
bcFlags1.iconIsKeyEquivalent	The icon is the key equivalent.
bcFlags2.keyEquivalent	The key equivalent.

bcFlags2.transparent	Whether to draw.
periodicDelay	The delay before sending the first send by a continuous button.
periodicInterval	The interval at which a continuous button sends its action.

METHOD TYPES

Copying, Initializing and Freeing a ButtonCell	– copyFromZone – init – initIconCell: – initTextCell: – free
Determining Component Sizes	– calcCellSize:inRect: – getDrawRect: – getIconRect: – getTitleRect:
Modifying the Title	– altTitle – setAltTitle: – setFont: – setTitle: – setTitleNoCopy: – title
Modifying the Icon	– altIcon – altImage – icon – image – iconPosition – setAltIcon: – setAltImage: – setIcon: – setImage: – setIconPosition:
Modifying the Sound	– setSound: – sound

Setting the State	– doubleValue
	– floatValue
	– intValue
	– setDoubleValue:
	– setFloatValue:
	– setIntValue:
	– setStringValue:
	– setStringValueNoCopy:
	– stringValue
Setting the Button Repeat	– getPeriodicDelay:andInterval:
	– setPeriodicDelay:andInterval:
Tracking the Mouse	– trackMouse:inRect:ofView:
Setting the Key Equivalent	– keyEquivalent
	– setKeyEquivalent:
	– setKeyEquivalentFont:
	– setKeyEquivalentFont:size:
Setting Parameters	– getParameter:
	– setParameter:to:
Modifying Graphic Attributes	– highlightsBy
	– isBordered
	– isOpaque
	– isTransparent
	– setBordered:
	– setHighlightsBy:
	– setShowsStateBy:
	– setTransparent:
	– setType:
	– showsStateBy
Simulating a Click	– performClick:
Displaying	– drawInside:inView:
	– drawSelf:inView:
	– highlight:inView:lit:
Archiving	– read:
	– write:

INSTANCE METHODS

altIcon

– (const char *)**altIcon**

Returns the ButtonCell's alternate icon by name. This icon will appear on the Button when it is in its alternate state. If there is no alternate icon, it returns NULL. This is the icon that will be displayed if the **iconPosition** is not NX_TITLEONLY and the **changeContents** or **lightByContents** flag is set.

altImage

– **altImage**

Returns the ButtonCell's alternate icon by **id**. This image will appear on the Button when it is in its alternate state. If there is no alternate image, it returns **nil**. This is the image that will be displayed if the **iconPosition** is not NX_TITLEONLY and the **changeContents** or **lightByContents** flag is set.

altTitle

– (const char *)**altTitle**

Returns the ButtonCell's alternate title. This is the text string that will appear on the button if the **iconPosition** is not NX_ICONONLY and the **changeContents** or **lightByContents** flag is set.

calcCellSize:inRect:

– **calcCellSize:**(NXSize *)*theSize* **inRect:**(const NXRect *)*aRect*

Returns, by reference, the minimum width and height required for displaying the button in *aRect*. The computation is done as follows:

1. The size of the contents instance variable is computed.
2. The size of the altContents is computed.
3. The maximum width and height are set in *theSize*.
4. If the button has an additional icon, its width and height are calculated; if either is bigger than the contents size, the size is increased to accommodate the icon.
5. If the button has a border, then the width and the height are incremented by the border width.

copyFromZone

– **copyFromZone:**(NXZone *)*zone*

Allocates, initializes, and returns a copy of the ButtonCell. Allocates the copy from *zone*.

doubleValue

– (double)**doubleValue**

Returns the ButtonCell's state cast as a double (0.0 or 1.0).

drawInside:inView:

– **drawInside:**(const NXRect *)*aRect* **inView:***controlView*

Draws the inside of the ButtonCell (the text and the icon and their background, but not the bezel). This method is called by **drawSelf:inView:** and by the Control classes' **drawCellInside:** method. It is provided so that when a ButtonCell's state is set (via **setState:**, **setIntValue:**, and others), a minimal update of the ButtonCell's visual appearance can occur. If you subclass ButtonCell and override **drawSelf:inView:** you MUST override this method as well (however, you are free to override only this method and not **drawSelf:inView:** as long as your subclass draws inside the same area as ButtonCell does). Returns **self**.

See also: – **drawInside:inView:** (Cell)

drawSelf:inView:

– **drawSelf:**(const NXRect *)*cellFrame* **inView:***controlView*

Displays the ButtonCell in the given rectangle of the given view. Focus must be locked on *controlView*. It draws the border of the ButtonCell if necessary, then calls **drawInside:inView:**. Returns **self**.

floatValue

– (float)**floatValue**

Returns the ButtonCell's state cast as a float (0.0 or 1.0).

free

– **free**

Disposes of the memory used by the ButtonCell and returns **nil**.

getDrawRect:

– **getDrawRect:**(NXRect *)*theRect*

Returns **self** and, by reference, the bounds of the area into which the text and/or icon will be drawn. You must pass the bounds of the ButtonCell in *theRect* (the same bounds passed to **drawSelf:inView:**). It assumes that the ButtonCell is being drawn in a flipped view.

getIconRect:

– **getIconRect:**(NXRect *)*theRect*

Returns **self** and, by reference, the bounds of the area into which the icon of the ButtonCell will be drawn. If the button has no icon, then *theRect* will not be touched. You must pass the bounds of the ButtonCell in *theRect* (the same bounds passed to **drawSelf:inView:**). It assumes that the ButtonCell is being drawn in a flipped view.

getParameter:

– (int)**getParameter:**(int)*aParameter*

Returns the state of a number of frequently accessed flags for a ButtonCell. The following constants correspond to the different flags:

NX_CELLDISABLED
NX_CELLSTATE
NX_CELLHIGHLIGHTED
NX_CELLEDITABLE
NX_CHANGECONTENTS
NX_CHANGEBACKGROUND
NX_CHANGEGRAY
NX_LIGHTBYCONTENTS
NX_LIGHTBYBACKGROUND
NX_LIGHTBYGRAY
NX_PUSHIN
NX_OVERLAPPINGICON
NX_ICONHORIZONTAL
NX_ICONONLEFTORBOTTOM
NX_ICONISKEYEQUIVALENT

You don't normally invoke this method since all of these flags are available via normal querying methods (e.g., **isEnabled**, **highlightsBy:**, etc.).

getPeriodicDelay:andInterval:

– **getPeriodicDelay:**(float *)*delay* **andInterval:**(float *)*interval*

Returns two values: The amount of time (in seconds) that a continuous button will pause before starting to periodically send action messages to the target object, and the interval (also in seconds) at which those messages are sent. Returns **self**.

See also: – **setContinuous:** (Cell), – **setPeriodicDelay:andInterval:**

getTitleRect:

– **getTitleRect:**(NXRect *)*theRect*

Returns **self** and, by reference, a copy of the bounds of the area into which the text of the ButtonCell will be drawn. You must pass the bounds of the ButtonCell in *theRect* (the same bounds passed to **drawSelf:inView:**). It assumes that the ButtonCell is being drawn in a flipped view.

highlight:inView:lit:

– **highlight:**(const NXRect *)*cellFrame*
  inView:*controlView*
  lit:(BOOL)*flag*

Highlights the ButtonCell if its highlighted flag is not equal to *flag*. You must **lockFocus** on *controlView* before calling this method. If possible, this method tries to use **NXHighlightRect** (i.e., if the button is not pushIn and **changeContents** and **lightByContents** are not set). If it cannot use **NXHighlightRect**, then it simply calls **drawSelf:inView:** or **drawInside:inView:** dependent upon whether the border of the button is involved in the highlighting process (e.g., in a pushIn button). Does nothing if the button is disabled or transparent. Returns **self**.

highlightsBy

– (int)**highlightsBy**

Returns the logical OR of one or more flags that indicate the way the ButtonCell highlights when the button is pressed. See **setHighlightsBy:** for the list of flags.

icon

– (const char *)**icon**

Returns the ButtonCell's icon by name.. If there is no icon displayed in the ButtonCell, or if the icon is the key equivalent, then it returns NULL.

See also: – **setIcon:**

iconPosition

– (int)**iconPosition**

Returns the position of the ButtonCell's icon. See **setIconPosition:** for the valid positions. The default is NX_TITLEONLY if the ButtonCell is created with **newTextCell:** or NX_ICONONLY if created with **newIconCell:**.

image

– **image**

Returns the ButtonCell's icon by **id**. If there is no image displayed in the ButtonCell, or if the image is the key equivalent, then it returns **nil**.

See also: – **setImage:**

init

– **init**

Initializes and returns the receiver, a new ButtonCell, as a text cell with the word "Button" on it.

initIconCell:

– **initIconCell:**(const char *)*iconName*

Initializes and returns the receiver, a new ButtonCell, with default size. By default, the ButtonCell is bordered and is pushIn. None of the changeXXX flags is set. The **lightByGray** and **lightByBackground** flags are set. This means that, when pressed, the button will perform **NXHighlightRect()** if the icon has no alpha or will change the background (from light gray to white) if the icon does have alpha values. An icon is a named NXImage; see the NXImage class for details. This is the designated initializer for ButtonCells that display icons.

See also: – **findImageNamed:** (NXImage)

initTextCell:

– **initTextCell:**(const char *)*aString*

Initializes the receiver, a new ButtonCell, with default size, font, title, and centered alignment. By default, the ButtonCell is bordered and is pushIn. None of the changeXXX is set and the button will "light up" when pressed (**lightByGray** and **lightByBackground** are set). This is the designated initializer for ButtonCells that display text.

intValue

– (int)**intValue**

Returns the ButtonCell's state (0 or 1).

isBordered

– (BOOL)**isBordered**

Returns YES if the button has a border, NO if not.

isOpaque

– (BOOL)**isOpaque**

Returns YES if drawing the ButtonCell touches all the bits in its frame, NO if not. The ButtonCell is opaque if it is not transparent and if it has a border.

isTransparent

– (BOOL)**isTransparent**

Returns YES if the ButtonCell is transparent, NO if not.

See also: – **setTransparent:**

keyEquivalent

– (unsigned short)**keyEquivalent**

Returns the key equivalent character of the ButtonCell.

performClick:

– **performClick:***sender*

If this ButtonCell is contained in a Control, then invoking this method causes the ButtonCell to act exactly as if the user had clicked the button.

read:

– **read:**(NXTypedStream *)*stream*

Reads the ButtonCell from the typed stream *stream*.

setAltIcon:

– **setAltIcon:**(const char *)*iconName*

Sets the ButtonCell's alternate icon by name; *iconName* is the name of an image to be displayed. This icon is displayed if the **changeContents** or **lightByContents** flag is set; these are set by the **setShowsStateBy:** and **setHighlightsBy:** methods, respectively. Note that no icon will be displayed in a ButtonCell unless **setIcon:** or **setImage:** is invoked (thus, **setAltIcon:** by itself has no affect on the appearance of the button). Returns **self**.

See also: – **setIcon:**

setAltImage:

– **setAltImage:***altImage*

Sets the ButtonCell's alternate icon by **id**; *altImage* is the **id** of the image to be displayed. This image is displayed if the **changeContents** or **lightByContents** flag is set; these are set by the **setShowsStateBy:** and **setHighlightsBy:** methods, respectively. Note that no image will be displayed in a ButtonCell unless **setIcon:** or **setImage:** is invoked (thus, **setAltImage:** by itself has no effect on the appearance of the button). Returns **self**.

See also: – **setImage:**

setAltTitle:

– **setAltTitle:**(const char *)*aString*

Invoke this method to set the alternate title to a copy of *aString*. If the ButtonCell was not an NX_TEXTCELL, it is automatically converted, in which case its **support** instance variable is set to the default font. If there is an icon associated with this ButtonCell, then the iconAndText flag is set. Returns **self**.

setBordered:

– **setBordered:**(BOOL)*flag*

If *flag* is YES, sets the ButtonCell to display a border; if *flag* is NO, it has none. Redraws the ButtonCell if its bordered status changes. Returns **self**.

setDoubleValue:

– **setDoubleValue:**(double)*aDouble*

Sets the ButtonCell's state to 1 if *aDouble* is nonzero, 0 otherwise. Returns **self**.

setFloatValue:

– **setFloatValue:**(float)*aFloat*

Sets the ButtonCell's state to 1 if *aFloat* is non-zero, 0 otherwise. Returns **self**.

setFont:

– **setFont:***fontObj*

Sets the font to be used when displaying text. Does nothing if the cell type is not NX_TEXTCELL. Returns **self**.

setHighlightsBy:

– **setHighlightsBy:**(int)*aType*

Sets the way the button highlights itself. *aType* can be the logical OR of one or more of the following constants:

NX_PUSHIN	The button "pushes in" when pressed (default)
NX_NONE	No difference when highlighted
NX_CONTENTS	Use the alternate contents
NX_CHANGEGRAY	Light gray -> white, white -> light gray
NX_CHANGEBACKGROUND	Same as NX_CHANGEGRAY, but only touches background

If you specify both NX_CHANGEGRAY and NX_CHANGEBACKGROUND, then a choice will be made between the two based on whether the icon of your button (if any) has any alpha. If it does, then NX_CHANGEBACKGROUND will be used; otherwise, NX_CHANGEGRAY will be used. If your button has no icon, then NX_CHANGEGRAY will be used. Returns **self**.

setIcon:

– **setIcon:**(const char *)*iconName*

Sets the ButtonCell's icon by name; *iconName* is the name of an image to be displayed. If there is no text associated with the ButtonCell, then it is converted to NX_ICONCELL; otherwise, the iconOverlaps flag is set. An icon is a named NXImage. Returns **self**.

See also: – **findImageNamed:** (NXImage)

setIconPosition:

– **setIconPosition:**(int)*aPosition*

Sets the position of the icon for this ButtonCell. *aPosition* can be one of the following constants:

NX_TITLEONLY = title only	(iconAndText = 0, iconOverlaps = 0)
NX_ICONONLY = icon only	(iconAndText = 0, iconOverlaps = 1)
NX_ICONLEFT = icon left of the text	(iconAndText = 1, iconOverlaps = 0)
NX_ICONRIGHT = right of the text	(iconAndText = 1, iconOverlaps = 0)
NX_ICONBELOW = below the text	(iconAndText = 1, iconOverlaps = 0)
NX_ICONABOVE = above the text	(iconAndText = 1, iconOverlaps = 0)
NX_ICONOVERLAPS = overlapping	(iconAndText = 1, iconOverlaps = 1)

If the position is top or bottom, the alignment of the text will be set to NX_CENTERED. This can be overridden with a subsequent **setAlignment:**. Returns **self**.

setImage:

– **setImage:***image*

Sets the ButtonCell's icon; *image* is the **id** of an image to be displayed. Returns **self**.

setIntValue:

– **setIntValue:**(int)*anInt*

Sets the ButtonCell's state to 1 if *anInt* is nonzero, 0 otherwise. Returns **self**.

setKeyEquivalent:

– **setKeyEquivalent:**(unsigned short)*charCode*

Sets the key equivalent character of the ButtonCell. The key equivalent will appear on the button only if there is no icon set (with **setIcon:** or **setAltIcon:**) and the **iconPosition** is not NX_TITLEONLY or NX_ICONONLY or NX_ICONOVERLAPS. The canonical way to put the key equivalent character on your button is to invoke **setKeyEquivalent:**, then invoke **setIconPosition:**NX_ICONRIGHT (or LEFT or ABOVE or BELOW). Menu entries (which inherit from ButtonCell) are usually the only ButtonCells with key equivalents. Returns **self**.

A ButtonCell's key equivalent can be tested by sending it a **keyEquivalent** message.

See also: **– keyEquivalent**, – **performClick:** (Matrix, Button)

setKeyEquivalentFont:

– **setKeyEquivalentFont:***fontObj*

Sets the font used to draw the **keyEquivalent**. Does nothing if there is already an icon associated with this ButtonCell. The default font is the same as that used to draw the text on the ButtonCell. Returns **self**.

setKeyEquivalentFont:size:

– **setKeyEquivalentFont:**(const char *)*fontName* **size:**(float)*fontSize*

Convenient form of **setKeyEquivalent:** that sets both the font and font size used to draw the **keyEquivalent**. Returns **self**.

setParameter:to:

– **setParameter:**(int)*aParameter* **to:**(int)*value*

Sets the most usual flags of a ButtonCell. See **getParameter:** for the list of usual flags. You do not usually invoke this method; instead use the appropriate **set...** methods to set flags. Returns **self**.

setPeriodicDelay:andInterval:

– **setPeriodicDelay:**(float)*delay* **andInterval:**(float)*interval*

This method sets two values: The amount of time (in seconds) that a continuous button will pause before starting to periodically send action messages to the target object, and the interval (also in seconds) at which those messages are sent. The maximum delay or interval is 60.0 seconds. Returns **self**.

See also: – **setContinuous:** (Cell)

setShowsStateBy:

– **setShowsStateBy:**(int)*aType*

Sets the way the button shows its alternate state. *aType* should be the logical OR of one or more of the following constants:

NX_PUSHIN	The button "pushes in" when pressed (default)
NX_NONE	No difference when highlighted
NX_CONTENTS	Use the alternate contents
NX_CHANGEGRAY	Light gray -> white, white -> light gray
NX_CHANGEBACKGROUND	Same as NX_CHANGEGRAY, but only touches background

If you specify both NX_CHANGEGRAY and NX_CHANGEBACKGROUND, then a choice will be made between the two based on whether the icon of your button (if any) has any alpha. If it does, then NX_CHANGEBACKGROUND will be used, else NX_CHANGEGRAY. If your button has no icon, then NX_CHANGEGRAY will be used. Returns **self**.

setSound:

– **setSound:***aSound*

Sets the sound that will be played when the mouse goes down in the ButtonCell. If you use a sound on your button, you must link your application against the soundkit. Returns **self**.

setStringValue:

– **setStringValue:**(const char *)*aString*

Sets the state of the ButtonCell. If *aString* is a non-null string, the state is set to 1; if *aString is* null, the state is set to 0. Returns **self**.

setStringValueNoCopy:

– **setStringValueNoCopy:**(const char *)*aString*

Same as **setStringValue:**.

setTitle:

– **setTitle:**(const char *)*aString*

Sets the text that is displayed on the button to *aString*. If there is already an icon associated with the button, then the **iconAndText** flag is set to YES. Returns **self**.

setTitleNoCopy:

– **setTitleNoCopy:**(const char *)*aString*

Similar to **setTitle:** but does not make a copy of *aString*. Returns **self**.

setTransparent:

– **setTransparent:**(BOOL)*flag*

Sets whether the ButtonCell is transparent. A transparent button never draws anything, but it does track the mouse and send its action normally. This method is useful for sensitizing an area on the screen so that an action gets sent to a target when the area receives a mouse click. Returns **self**.

setType:

– **setType:**(int)*aType*

Sets standard button types. The ButtonCell does not record the type directly; instead, this method sets the changeXXX and lightByXXX flags appropriately. The NX_SWITCH and NX_RADIOBUTTON types also set the icon to the default icon for that type of button (only if there is not already an icon set). *aType* can be one of the following constants:

NX_MOMENTARYPUSH
NX_MOMENTARYCHANGE
NX_PUSHONPUSHOFF
NX_TOGGLE
NX_SWITCH
NX_RADIOBUTTON

This method is invoked by Button's **setType:** method. It is very useful for creating prototype cells in a matrix of radio buttons. Returns **self**.

See also: – **setType:** (Button)

showsStateBy

– (int)**showsStateBy**

Returns flags reflecting the way that the button shows its alternate state. See **setShowsStateBy:** for list of appropriate flags. Returns **self**.

sound

– **sound**

Returns the sound object that is sent a **play** message on a mouse-down event in the ButtonCell.

See also: – **setSound:**

stringValue

– (const char *)**stringValue**

Returns the ButtonCell's state as a string. If the state is 1, "" (empty string) is returned, otherwise, NULL is returned. This is an unusual method to invoke (since the **stringValue** of a button doesn't make much sense) and is included only for completeness.

title

– (const char *)**title**

Returns ButtonCell's text if the receiving ButtonCell displays any text; otherwise it returns NULL.

trackMouse:inRect:ofView:

– (BOOL)**trackMouse:**(NXEvent *)*theEvent*
  inRect:(const NXRect *)*cellFrame*
  ofView:*controlView*

Tracks the mouse by starting the sound (if any) and calling
[super **trackMouse:***theEvent* **inRect:***cellFrame* **ofView:***controlView*]. Returns YES if the mouse button goes up with the cursor in the cell, NO otherwise.

See also: – **trackMouse:inRect:ofView:** (Cell)

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving ButtonCell to the typed stream *stream*. Returns **self**.

CONSTANTS AND DEFINED TYPES

```
/* Button Types */
#define NX_MOMENTARYPUSH    0
#define NX_PUSHONPUSHOFF    1
#define NX_TOGGLE           2
#define NX_SWITCH           3
#define NX_RADIOBUTTON      4
#define NX_MOMENTARYCHANGE  5
```

Cell

INHERITS FROM	Object
DECLARED IN	appkit/Cell.h

CLASS DESCRIPTION

Cell is an abstract super class that provides many useful functions needed for displaying text or icons without the overhead of a full View subclass. In particular, it provides most of the functionality of a Text class by providing access to a shared Text object that can be used by all instances of Cell in an Application. Cell is used heavily by the Control classes to implement their internal workings. Some subclasses of Control (notably Matrix) allow multiple Cells to be grouped and act together in some cooperative manner. Thus, with a Matrix, a group of radio buttons can be implemented without needing a View for each button (and without needing a Text object for the text on each button). Cells are also extremely useful for placing titles or icons at will in a custom subclass of View.

The Cell class provides primitives for displaying text or an icon, editing text, formatting floating point numbers, maintaining state, highlighting, and tracking the mouse. It has several subclasses: SelectionCell, NXBrowserCell, and ActionCell (which in turn has the subclasses ButtonCell, SliderCell, TextFieldCell, and FormCell). Cell's **trackMouse:inRect:ofView:** method supports the target object and action method used to implement controls. However, Cell implements these features abstractly, deferring the details of implementation to ActionCell.

The **initIconCell:** method is the designated initializer for Cells that display icons. The **initTextCell:** method is the designated initializer for Cells that display text. Override one of these methods if you implement a subclass of Cell that performs its own initialization.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in Cell	char	*contents;
	id	support;

```
struct _cFlags1 {
    unsigned int        state:1;
    unsigned int        highlighted:1;
    unsigned int        disabled:1;
    unsigned int        editable:1;
    unsigned int        type:2;
    unsigned int        freeText:1;
    unsigned int        alignment:2;
    unsigned int        bordered:1;
    unsigned int        bezeled:1;
    unsigned int        selectable:1;
    unsigned int        scrollable:1;
    unsigned int        entryType:3;
}                   cFlags1;
struct _cFlags2 {
    unsigned int        continuous:1;
    unsigned int        actOnMouseDown:1;
    unsigned int        floatLeft:4;
    unsigned int        floatRight:4;
    unsigned int        autoRange:1;
    unsigned int        actOnMouseDragged:1;
    unsigned int        noWrap:1;
    unsigned int        dontActOnMouseUp:1;
    }               cFlags2;
```

contents	String for a TextCell, name of the icon for an IconCell.
support	Font for TextCell, NXImage for IconCell.
cFlags1.state	Current state of the Cell (0 or 1).
cFlags1.highlighted	Whether Cell is highlighted.
cFlags1.disabled	Whether Cell is disabled.
cFlags1.editable	Whether text in the Cell is editable.
cFlags1.type	NULLCELL, TEXTCELL, or ICONCELL.
cFlags1.freeText	Whether to free contents when freeing the Cell.
cFlags1.alignment	Text justification.

cFlags1.bordered	Whether the Cell has a border.
cFlags1.bezeled	Whether the Cell has a bezeled border.
cFlags1.selectable	Whether the text is selectable.
cFlags1.scrollable	Whether the text is scrollable.
cFlags1.entryType	Type of data accepted.
cFlags2.continuous	Sends action continuously to target while control is active.
cFlags2.actOnMouseDown	Sends action on the mouse-down (rather than the mouse-up).
cFlags2.floatLeft	Digits to left of decimal when text is floating-point number.
cFlags2.floatRight	Digits to right of decimal when text is floating-point number.
cFlags2.autoRange	Autorange decimal when text is floating point number.
cFlags2.actOnMouseDragged	Send action every time the mouse changes position.
cFlags2.noWrap	0 = word wrap, 1 = character wrap.
cFlags2.dontActOnMouseUp	Don't send the action on the mouse-up event.

METHOD TYPES

Copying, initializing, and freeing a Cell	– copy – copyFromZone: – init – initIconCell: – initTextCell: – free
Determining component sizes	– calcCellSize: – calcCellSize:inRect: – calcDrawInfo: – getDrawRect: – getIconRect: – getTitleRect:

Setting the Cell's type	– setType: – type
Setting the Cell's state	– incrementState – setState: – state
Enabling and disabling the Cell	– isEnabled – setEnabled:
Modifying the Icon	– icon – setIcon:
Setting Cell values	– doubleValue – floatValue – intValue – setDoubleValue: – setFloatValue: – setIntValue: – setStringValue: – setStringValueNoCopy: – setStringValueNoCopy:shouldFree: – stringValue
Modifying text attributes	– alignment – font – isEditable – isScrollable – isSelectable – setAlignment: – setEditable: – setFont: – setScrollable: – setSelectable: – setTextAttributes: – setWrap:
Editing text	– edit:inView:editor:delegate:event: – endEditing: – select:inView:editor:delegate:start:length:
Validating input	– entryType – isEntryAcceptable: – setEntryType:
Formatting data	– setFloatingPointFormat:left:right:

Modifying graphic attributes	– isBezeled
	– isBordered
	– isOpaque
	– setBezeled:
	– setBordered:
Setting parameters	– getParameter:
	– setParameter:to:
Interacting with other Cells	– takeDoubleValueFrom:
	– takeFloatValueFrom:
	– takeIntValueFrom:
	– takeStringValueFrom:
Displaying	– controlView
	– drawInside:inView:
	– drawSelf:inView:
	– highlight:inView:lit:
	– isHighlighted
Target and action	– action
	– getPeriodicDelay:andInterval:
	– isContinuous
	– sendActionOn:
	– setAction:
	– setContinuous:
	– setTarget:
	– target
Assigning a tag	– setTag:
	– tag
Handling keyboard alternatives	– keyEquivalent
Tracking the mouse	– continueTracking:at:inView:
	– mouseDownFlags
	+ prefersTrackingUntilMouseUp
	– startTrackingAt:inView:
	– stopTracking:at:inView:mouseIsUp:
	– trackMouse:inRect:ofView:
Managing the cursor	– resetCursorRect:inView:
Archiving	– awake
	– read:
	– write:

CLASS METHODS

prefersTrackingUntilMouseUp

+ (BOOL)**prefersTrackingUntilMouseUp**

Returns NO by default. Override this method to return YES if the Cell should, after a mouse-down event, track mouse-dragged and mouse-up events even if they occur outside the Cell's frame. This method is overridden to ensure that a SliderCell in a matrix doesn't stop responding to user input (and its neighbor start responding) just because the knob isn't dragged in a perfectly straight line.

INSTANCE METHODS

action

– (SEL)**action**

Returns a null selector. This method is overridden by Action Cell and its subclasses, which actually implement the target object and action method.

alignment

– (int)**alignment**

Returns the alignment of text in the Cell. The return value can be one of three constants: NX_LEFTALIGNED, NX_CENTERED, or NX_RIGHTALIGNED.

awake

– **awake**

Used during unarchiving; initializes static variables for the Cell class. Returns **self**.

calcCellSize:

– **calcCellSize:**(NXSize *)*theSize*

Returns **self** and, by reference, the minimum width and height required for displaying the Cell. It's implemented by calling **calcCellSize:inRect:** with the rectangle argument set to a rectangle with very large width and height. This should be overridden if that is not the proper way to calculate the minimum width and height required for displaying the Cell (SliderCell overrides this method for that reason).

calcCellSize:inRect:

– **calcCellSize:**(NXSize *)*theSize* **inRect:**(const NXRect *)*aRect*

Returns **self** and, by reference, the minimum width and height required for displaying the Cell in a given rectangle. If it's not possible to fit, the width and/or height could be bigger than the ones of the rectangle. The computation is done by trying to size the Cell so that it fits in the rectangle argument (by wrapping the text for instance). If a choice must be made between extending the width or height of *aRect* to fit the text, the height will be extended.

calcDrawInfo:

– **calcDrawInfo:**(const NXRect *)*aRect*

Objects using Cells generally maintain a flag that informs them if any of their Cells has been modified in such a way that the location or size of the Cell should be recomputed. If so a method (usually named **calcSize**) is automatically invoked before displaying the Cell; this method invokes Cell's **calcDrawInfo:** for each Cell. Subclasses of Cell can override **calcDrawInfo:** to cache some information that could speed up the drawing of the Cell. In Cell, this method does nothing and returns **self**.

See also: **– calcSize** (Matrix)

continueTracking:at:inView:

– (BOOL)**continueTracking:**(const NXPoint *)*lastPoint*
    at:(const NXPoint *)*currentPoint*
    inView:*controlView*

Returns YES if it's OK to keep tracking. This method is invoked by **trackMouse:inRect:ofView:** as the mouse is dragged around inside the Cell. By default, this method returns YES when the **cFlags2.continuous** or **cFlags2.actOnMouseDragged** is set to YES. This method is often overridden to provide more sophisticated tracking behavior.

controlView

– **controlView**

Returns **nil**. This method is implemented abstractly, since Cell doesn't have an instance variable for the view in which an instance is drawn. It's overridden by ActionCell and its subclasses, which use the controlView's **id** as the only argument in the action message when it's sent to the target.

See also: – **controlView** (ActionCell)

copy

– **copy**

Allocates and returns a copy of the receiving Cell. The copy is allocated from the default zone and is assigned the **contents** of the receiver.

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Allocates and returns a copy of the receiving Cell. The copy is allocated from *zone* and is assigned the **contents** of the receiver. When you subclass Cell, override this method to send the message [super **copyFromZone:**], then copy each of the subclass's unique instance variables separately.

doubleValue

– (double)**doubleValue**

Returns the receiver's double value by converting its **contents** to a double using the C function **atof**(). Returns 0 if the cell type is not NX_TEXTCELL.

drawInside:inView:

– **drawInside:**(const NXRect *)*cellFrame* **inView:***controlView*

Draws the inside of the Cell; it's the same as **drawSelf:inView:** except that it does not draw the bezel or border if there is one. All subclasses of Cell which implement **drawSelf:inView:** *must* implement **drawInside:inView:**. **drawInside:inView:** should never invoke **drawSelf:inView:**, but **drawSelf:inView:** can invoke **drawInside:inView:** (in fact, it often does). **drawInside:inView:** is invoked from the Control class's **drawCellInside:** method and is used to cause minimal drawing to be done in order to update the value displayed by the Cell when the **contents** is changed. This becomes more important in more complex Cells such as ButtonCell and SliderCell. The passed *cellFrame* should be the frame of the Cell (i.e., the same *cellFrame* passed to **drawSelf:inView:**), *not* the rectangle returned by **getDrawRect:**! Be sure to lock focus on the *controlView* before invoking this method. If **cFlags1.highlighted** is YES, then the Cell is highlighted (by changing light gray to white and white to light gray throughout *cellFrame*). Returns **self**.

drawSelf:inView:

– **drawSelf:**(const NXRect *)*cellFrame* **inView:***controlView*

Displays the contents of a Cell in a given rectangle of a given view. Lock the focus on the *controlView* before invoking this method. It draws the border or bezel (if any), then invokes **drawInside:inView:**. A text Cell displays its text in the rectangle by using a global Text object, an icon Cell displays its icon centered in the rectangle if it fits in the rectangle, by setting the icon origin on the rectangle origin if it does not fit. Nothing is displayed for NX_NULLCELL. You can override this method if you want a display that is specific to your own subclass of Cell. Returns **self**.

See also: **– drawInside:inView:**

edit:inView:editor:delegate:event:

– **edit:**(const NXRect *)*aRect*
  inView:*controlView*
  editor:*textObj*
  delegate:*anObject*
  event:(NXEvent *)*theEvent*

Use this method to edit the text of a Cell by using the Text object *textObj* in response to an NX_MOUSEDOWN event. The *aRect* argument must be the one you have used when displaying the Cell. *theEvent* is the NX_MOUSEDOWN event. *anObject* is made the delegate of the Text object *textObj* used for the editing: it will receive the methods such as **textDidEnd:endChar:**, **textWillEnd**, **textDidResize**, **textWillResize**, and others sent by the Text object while editing. If the cell type is not equal to NX_TEXTCELL no editing is performed, otherwise the Text object is sized to *aRect* and its superview is set to *controlView*, so that it exactly covers the Cell. Then it's activated and editing begins. It's the responsibility of the delegate to end the editing, remove any data from the *textObj* and invoke **endEditing:** on the Cell in the **textDidEnd:endChar:** method. Returns **self**.

endEditing:

– **endEditing:***textObj*

Use this method to end the editing you began with **edit:inView:editor:delegate:event:** or **select:inView:editor:delegate:start:length:**. Usually this method is called by the **textDidEnd:endChar:** method of the object you are using as the delegate for the Text object (most often a Matrix or TextField). It removes the Text object from the view hierarchy and sets its delegate to **nil**. Returns **self**.

entryType

– (int)**entryType**

Returns the type of data allowed in the Cell. See **setEntryType:** for the list of valid types.

floatValue

– (float)**floatValue**

Returns the receiver's **float** value by converting its **contents** to a float using the C function **atof()**. Returns 0.0 if the cell type is not NX_TEXTCELL.

font

– **font**

Returns the font used to display text in the Cell. Returns **nil** if the Cell is not of type NX_TEXTCELL.

free

– **free**

Frees all disposable storage used by the Cell. If **cFlags1.freeText** is YES, then the **contents** instance variable is freed. Returns **nil**.

getDrawRect:

– **getDrawRect:**(NXRect *)*theRect*

Returns **self** and, by reference, the rectangle into which the Cell will draw its "insides." In other words, this method usually returns the rectangle which is touched by **drawInside:inView:**. Pass the bounds of the Cell in *theRect.*

getIconRect:

– **getIconRect:**(NXRect *)*theRect*

Returns **self** and, by reference, the rectangle into which the icon will be drawn. Pass the bounds of the Cell in *theRect*. If this Cell does not draw an icon, *theRect* is untouched.

getParameter:

– (int)**getParameter:**(int)*aParameter*

Returns the most usual flags of a Cell. The following constants corresponds to the different flags:

NX_CELLDISABLED
NX_CELLSTATE
NX_CELLHIGHLIGHTED
NX_CELLEDITABLE

It is, in general, much better to invoke the "is" methods (**isEnabled**, **isHighlighted**, **isEditable**) rather than use **getParameter:**.

getPeriodicDelay:andInterval:

– **getPeriodicDelay:**(float*)*delay* **andInterval:**(float*)*interval*

Sets two values: the amount of time (in seconds) that a continuous button will pause before starting to periodically send action messages to the target object, and the interval (also in seconds) at which those messages are sent. Periodic messaging behavior is controlled by Cell's **sendActionOn:** and **setContinuous:** methods. (By default, Cell sends the action message on mouse up events.) The default values returned by this method are 0.2 seconds delay and 0.025 seconds interval. Can be overridden. Returns **self**.

getTitleRect:

– **getTitleRect:**(NXRect *)*theRect*

Returns **self** and, by reference, the rectangle into which the text will be drawn. Pass the bounds of the Cell in *theRect*. If this Cell does not draw any text, *theRect* is untouched.

highlight:inView:lit:

– **highlight:**(const NXRect *)*cellFrame*
    inView:*controlView*
    lit:(BOOL)*flag*

If **cFlags1.highlighted** is not equal to *flag*, it's set to *flag* and the rectangle *cellFrame* is highlighted in *controlView*. (You must **lockFocus** on *controlView* before calling this method.) The default is simply to composite with NX_HIGHLIGHT inside the bounds of the *cellFrame*. Override this method if you want a more sophisticated highlighting behavior in a Cell subclass. Note that the highlighting that the base Cell class does will *not* appear when printed (although subclasses like TextFieldCell, SelectionCell, and ButtonCell can print themselves highlighted). This is due to the fact that the base Cell class is transparent, and there is no concept of transparency in printed output. Returns **self**.

icon

– (const char *)**icon**

Returns the name of the icon currently used by the Cell. Returns NULL if the cell type is not NX_ICONCELL.

incrementState

– **incrementState**

Adds 1 to the state of the Cell, wrapping around to 0 from maximum value (for the base Cell class, 1 wraps to 0). Subclasses may want to change the meaning of this method (for multistate Cells, for example). Remember that if you want the visual appearance of the Cell to reflect a change in state, you must invoke **drawSelf:inView:** after altering the state (and your **drawSelf:inView:** must draw the different states in different ways—the default implementation of the Cell class does *not* visually distinguish differences in state). Returns **self**.

init

– **init**

Initializes and returns the receiver, a new Cell instance, as type NX_NULLCELL. This method is the designated initializer for null cells.

initIconCell:

– **initIconCell:**(const char *)*iconName*

Initializes and returns the receiver, a new Cell instance, as type NX_ICONCELL. The icon is set to *iconName*. This method is the designated initializer for icon Cells.

See also: **– findImageFor:** (NXImage), **– name** (NXImage)

initTextCell:

– **initTextCell:**(const char *)*aString*

Initializes and returns the receiver, a new Cell instance, as type NX_TEXTCELL. The string value is set to *aString*. This method is the designated initializer for text Cells.

intValue

– (int)**intValue**

Returns the Cell's integer value by converting its **contents** to an integer using the C function **atoi**(). Returns 0 if the cell type is not NX_TEXTCELL.

isBezeled

– (BOOL)**isBezeled**

Returns YES if the Cell has a bezeled border, NO otherwise.

isBordered

– (BOOL)**isBordered**

Returns YES if the Cell is surrounded by a 1-pixel black frame, NO otherwise. The default is NO.

isContinuous

– (BOOL)**isContinuous**

Returns YES if the Cell continuously sends its action message to the target object when tracking. This usually has meaning only for subclasses of Cell that implement **target** and **action** instance variables (ActionCell and its subclasses), although some Control subclasses will send a default action to a default target even if the Cell does not itself have a **target** and **action**.

isEditable

– (BOOL)**isEditable**

Returns YES if the text in the Cell is editable, NO otherwise. The default is NO.

isEnabled

– (BOOL)**isEnabled**

Returns YES if the Cell is enabled, NO otherwise. The default is YES.

isEntryAcceptable:

– (BOOL)**isEntryAcceptable:**(const char *)*aString*

Tests whether *aString* matches the Cell's entry type, set by the **setEntryType:** method. Returns YES if it *aString* is acceptable by the receiving Cell, NO otherwise. This method is invoked by Form, Matrix, and other Controls to see if a new text string is acceptable for this Cell. This method doesn't check for overflow. It can be overridden to enforce specific restrictions on what the user can type into the Cell. If *aString* is NULL or empty, this method returns YES.

See also: **– setEntryType:**

isHighlighted

– (BOOL)**isHighlighted**

Returns YES if the Cell is currently highlighted, NO otherwise. The Cell can be highlighted by calling **highlight:inView:lit:**.

isOpaque

– (BOOL)**isOpaque**

Returns YES if the Cell is opaque (i.e., it touches every pixel in its bounds), NO otherwise. The base Cell class is opaque if and only if it has a bezel. Subclasses which draw differently should override this appropriately.

isScrollable

– (BOOL)**isScrollable**

Returns YES if typing past the end of the text in the Cell will cause the Cell to scroll to follow the typing. The default return value is NO.

isSelectable

– (BOOL)**isSelectable**

Returns YES if the text in the Cell is selectable, NO otherwise. The default return value is NO.

keyEquivalent

– (unsigned short)**keyEquivalent**

Returns 0. Should be overridden by subclasses to return a key equivalent for the receiver.

mouseDownFlags

– (int)**mouseDownFlags**

Returns the flags (e.g., NX_SHIFTMASK) that were set when the mouse went down to start the current tracking session. This is useful if you want to use these flags, but don't want the overhead of having to add NX_MOUSEDOWNMASK to the **sendActionOn:** mask just to get those flags. This method is only valid during tracking and does not work if the target of the Cell initiates another Cell tracking loop as part of its action method (for example, like PopUpLists do).

read:

– **read:**(NXTypedStream *)*stream*

Reads the Cell from the typed stream *stream*.

resetCursorRect:inView:

– **resetCursorRect:**(const NXRect *)*cellFrame* **inView:***controlView*

If the type of the Cell is NX_TEXTCELL, then a cursor rectangle is added to *controlView* (via **addCursorRect:cursor:**).

See also: **– addCursorRect:cursor:** (View, Control)

select:inView:editor:delegate:start:length:

– **select:**(const NXRect *)*aRect*
    inView:*controlView*
    editor:*textObj*
    delegate:*anObject*
    start:(int)*selStart*
    length:(int)*selLength*

Similar to **edit:inView:editor:delegate:event:** but you can invoke it in any situation, not only on a mouse-down event. You must specify the beginning and the length of the selection.

sendActionOn:

– (int)**sendActionOn:**(int)*mask*

Resets flags to determine when the action is sent to the target while tracking. Can be any combination of:

NX_MOUSEUPMASK
NX_MOUSEDOWNMASK
NX_MOUSEDRAGGEDMASK
NX_PERIODICMASK

The default is NX_MOUSEUPMASK. You can use the **setContinuous:** method to turn on the bit in the NX_PERIODICMASK or the NX_MOUSEDRAGGEDMASK (whichever is appropriate to the given subclass of Cell) in the current mask.

Returns the old mask.

setAction:

– **setAction:**(SEL)*aSelector*

Does nothing. Should be overridden by subclasses that implement **target** and **action** instance variables (ActionCell and its subclasses). Returns **self**.

setAlignment:

– **setAlignment:**(int)*mode*

Sets the alignment of text in the Cell and returns **self**. *mode* should be one of three constants: NX_LEFTALIGNED, NX_CENTERED, or NX_RIGHTALIGNED.

setBezeled:

– **setBezeled:**(BOOL)*flag*

If *flag* is YES, then the Cell is surrounded by a bezel, otherwise it's not. **setBordered:** and **setBezeled:** are mutually exclusive options. Returns **self**.

setBordered:

– **setBordered:**(BOOL)*flag*

If *flag* is YES, then the Cell is surrounded by a 1-pixel black frame, otherwise it's not. **setBordered:** and **setBezeled:** are mutually exclusive options. Returns **self**.

setContinuous:

– **setContinuous:**(BOOL)*flag*

Sets whether a Cell continuously sends its action message to the target object when tracking. Normally, this method will simply add NX_PERIODICMASK or NX_MOUSEDRAGGEDMASK to the mask set with **sendActionOn:**, depending on which setting is appropriate to the subclass implementing it. In the base Cell class, this method adds NX_PERIODICMASK to the mask. These settings usually have meaning only for ActionCell and its subclasses which implement instance variables for the target object and action method. However, some Control subclasses will send a default action to a default target when the Cell itself doesn't define **target** and **action** instance variables.

See also: **– sendActionOn:**

setDoubleValue:

– **setDoubleValue:**(double)*aDouble*

Sets the receiver to represent *aDouble*, by replacing the **contents** with the character string representing *aDouble*. Does nothing if the cell type is not NX_TEXTCELL. Returns **self**.

setEditable:

– **setEditable:**(BOOL)*flag*

Sets the editable state of the Cell. If *flag* is YES, then the text is also set to be selectable. If *flag* is NO, then the text is set not selectable. Returns **self**.

See also: **– edit:inView:editor:delegate:event:**

setEnabled:

– **setEnabled:**(BOOL)*flag*

Sets the enabled state of the Cell. Returns **self**.

setEntryType:

– **setEntryType:**(int)*aType*

This method sets the type of data allowed in the Cell. *aType* is one of these four constants:

NX_ANYTYPE
NX_(POS)INTTYPE
NX_(POS)FLOATTYPE
NX_(POS)DOUBLETYPE

If the Cell is not of type NX_TEXTCELL, it's automatically converted, in which case its **support** instance variable is set to the default font (Helvetica 12.0), and its string value is set to "Cell" (the default).

The entry type is checked by the **isEntryAcceptable:** method. That method is used by Controls that contain editable text (such as Matrix and TextField) to validate that what the user has typed is correct. If you want to have a custom Cell accept some specific type of data (other than those listed above), you can override the **isEntryAcceptable:** method to check for the validity of the data the user has entered.

See also: – **isEntryAcceptable:**, – **setFloatingPointFormat:left:right:**

setFloatingPointFormat:left:right:

– **setFloatingPointFormat:**(BOOL)*autoRange*
 left:(unsigned)*leftDigits*
 right:(unsigned)*rightDigits*

Sets whether floating-point numbers are autoranged, and sets the size of the fields to the left and right of the decimal point. *leftDigits* must be between 0 and 10. *rightDigits* must be between 0 and 14. If *leftDigits* is 0, then the number is not formatted. If *rightDigits* is 0, then the fractional part of the floating-point number is truncated (i.e., the floating-point number is printed as if it were an integer). Otherwise, *leftDigits* specifies the number of digits to the left of the decimal point, and *rightDigits* specifies the number of digits to the right. If *autoRange* is YES, the number will be fit into a field that's *leftDigits* + *rightDigits* + 1 spaces wide and the decimal point will be autoranged to fit that field (the field will also be padded with zeros). To turn off formatting, simply invoke this routine with *leftDigits* = 0. If the **entryType** of the Cell is not already NX_FLOATTYPE, NX_POSFLOATTYPE, NX_DOUBLETYPE, or NX_POSDOUBLETYPE, it's set to NX_FLOATTYPE. Returns **self**.

setFloatValue:

– **setFloatValue:**(float)*aFloat*

Sets cell-specific float value, by replacing its contents by the character string representing the float. Does nothing if the cell type is not NX_TEXTCELL. Returns **self**.

setFont:

– **setFont:***fontObj*

Sets the font to be used when displaying text in the Cell. Does nothing if the Cell is not of type NX_TEXTCELL. Returns **self**.

setIcon:

– **setIcon:**(const char *)*iconName*

Invoke this method to set the icon of the Cell to the icon represented by *iconName* (an icon is a named NXImage—see the NXImage class). If the Cell was not an NX_ICONCELL, it's automatically converted. Sets the **support** instance variable to *iconName*, and sets the **contents** instance variable to the result of sending the **name** message to that NXImage. If you specify an invalid NXImage name, you will get a default icon (you can verify that the NXImage you requested was valid by checking the result of sending the **icon** message to the Cell to be sure it matches the *iconName* you supplied). Returns **self**.

See also: **– findImageNamed** (NXImage), **– name** (NXImage)

setIntValue:

– **setIntValue:**(int)*anInt*

Sets cell-specific integer value by replacing its contents by the character string representing *anInt*. Does nothing if the cell type is not NX_TEXTCELL. Returns **self**.

setParameter:to:

– **setParameter:**(int)*aParameter* **to:**(int)*value*

Sets the most usual flags of a Cell. Calling this method could result in unpredictable results in subclasses. It's much safer to invoke the appropriate **set...** method to set a specific flag. Returns **self**.

See also: – **getParameter:**, – **highlightInView:lit:**, – **setEditable:**, – **setEnabled:**, – **setState:**

setScrollable:

– **setScrollable:**(BOOL)*flag*

Sets whether, while editing, the Cell will scroll to follow typing. Returns **self**.

See also: **– edit:inView:editor:delegate:event:**

setSelectable:

– **setSelectable:**(BOOL)*flag*

If *flag* is YES, then the text is selectable but not editable. If NO, then the text is static (not editable or selectable). Returns **self**.

See also: **– edit:inView:editor:delegate:event:**

setState:

– **setState:**(int)*value*

Sets the state of the Cell to 0 if *value* is 0, to 1 otherwise. Returns **self**.

See also: **– incrementState**

setStringValue:

– **setStringValue:**(const char *)*aString*

Invoke this method to set the **contents** instance variable to a copy of *aString*. If the Cell was not of type NX_TEXTCELL, it's automatically converted, in which case its **support** instance variable is set to the default font (Helvetica 12.0). If floating point parameters have been set (via **setFloatingPointParameters:left:right:**) and the type of the Cell is NX_(POS){FLOAT,DOUBLE}TYPE, then the string will be tested for being a **float** or a **double**. If it's a **float** or a **double**, then the appropriate parameterization will be applied; otherwise, the string will be copied directly. Returns **self**.

setStringValueNoCopy:

– **setStringValueNoCopy:**(const char *)*aString*

Similar to **setStringValue:** but does not make a copy of *aString*. The Cell records that it does not have to dispose of its **contents** instance variable when it receives the **free** message. Note that if you set a string this way, then the floating-point parameters will *not* be applied (since no copy of the string is being made). Returns **self**.

setStringValueNoCopy:shouldFree:

– **setStringValueNoCopy:**(char *)*aString* **shouldFree:**(BOOL)*flag*

Similar to **setStringValueNoCopy:**, but the caller can specify if the **contents** instance variable will be freed when the Cell receives the **free** message. Note that if you set a string this way, then the floating-point parameters will *not* be applied (since no copy of the string is being made). If *aString* == contents, then if *flag* is NO, **cFlags1.freeText** will be set to NO. Returns **self**.

setTag:

– **setTag:**(int)*anInt*

Does nothing. This method is overridden by ActionCell and its subclasses to support multiple-Cell controls (Matrix and Form). Returns **self**.

setTarget:

– **setTarget:***anObject*

Does nothing. This method is overridden by ActionCell and its subclasses that implement the target object and action method. Returns **self**.

setTextAttributes:

– **setTextAttributes:***textObj*

Invoked just before any drawing or editing occurs in the Cell. It's intended to be overridden. If you do override this method you must invoke [super **setTextAttributes:***textObj*] first. If you do not, you risk inheriting drawing attributes from the last Cell which drew any text. You should invoke only the following two Text instance methods:

```
setBackgroundGray:
setTextGray:
```

Do not set any other parameters in the Text object.

You should return *textObj* as the return value of this method. Therefore, if you want to substitute some other Text object to draw with (but not edit, editing always uses the window's field editor), you can return that object instead of *textObj* and it will be used for the draw that caused **setTextAttributes:** to be called.

TextFieldCell, a subclass of ActionCell, allows you to set the grays without creating your own subclass of Cell. You only need to subclass Cell to control the gray values if you don't want all of the functionality (and instance variable usage) of an ActionCell.

Defaults: If the Cell is disabled, its text gray will be NX_DKGRAY, otherwise it will be NX_BLACK. If the Cell has a bezel, then its background gray will be NX_WHITE, otherwise it will be NX_LTGRAY. The Text object does *not* paint the background gray before drawing; it only uses the background gray to erase characters while editing. The Cell class does paint the NX_WHITE background when it draws a bezeled Cell, but does not paint any background (i.e., it's transparent) otherwise.

Note that most of the other text object attributes can be set via Cell methods (**setFont**:, **setAlignment:**, **setWrap:**) so you need only override this method if you need to set the gray values. Returns **self**.

setType:

– **setType:**(int)*aType*

Sets the type of the Cell. It should be NX_TEXTCELL, NX_ICONCELL, or NX_NULLCELL. If *aType* is NX_TEXTCELL and the current type is not NX_TEXTCELL, then the font is set to the default font (Helvetica 12.0), and the string value of the Cell is set to the default string, "Cell". If *aType* is NX_ICONCELL and the current type is not NX_ICONCELL, then the icon for the Cell is set to be the default icon, "square16".

setWrap:

– **setWrap:**(BOOL)*flag*

If *flag* is YES, then the text (when displaying, not editing) will be wrapped to word breaks. Otherwise, it will not. The default is YES.

startTrackingAt:inView:

– (BOOL)**startTrackingAt:**(const NXPoint *)*startPoint* **inView:***controlView*

This method returns YES if and only if the Cell is continuous, that is, if **cFlags2.continuous** or **cFlags2.actOnMouseDragged** is YES. Called via **trackMouse:inRect:ofView:** the first time the mouse appears in the Cell needing to be tracked. Default is to do nothing. Should return YES if it's OK to track based on this starting point, otherwise it returns NO. This method is often overridden to provide more sophisticated tracking behavior.

state

– (int)**state**

Returns the state of the Cell (0 or 1). The default is 0.

stopTracking:at:inView:mouseIsUp:

– **stopTracking:**(const NXPoint *)*lastPoint*
at:(const NXPoint *)*stopPoint*
inView:*controlView*
mouseIsUp:(BOOL)*flag*

Invoked via **trackMouse:inRect:ofView:** when the mouse has left the bounds of the Cell, or the mouse button has gone up. *flag* is YES if the mouse button went up to cause this method to be invoked. The default method does nothing and returns **self**. This method is often overridden to provide more sophisticated tracking behavior. Returns **self**.

stringValue

– (const char *)**stringValue**

Returns a pointer to the **contents** instance variable.

tag

– (int)**tag**

Returns –1. Overridden by subclasses such as ActionCell to provide a way to identify Cells in a multiple-Cell Control such as Matrix or Form.

takeDoubleValueFrom:

– **takeDoubleValueFrom:***sender*

Sets the receiving Cell's double-precision floating point value to the value returned by *sender*'s **doubleValue** method. Returns **self**.

This method can be used in action messages between Cells. It permits one Cell (*sender*) to affect the value of another Cell (the receiver). For example, a TextFieldCell can be made the target of a SliderCell, which will send it **takeDoubleValueFrom:** action message. The TextFieldCell will get the SliderCell's **double** value, turn it into a text string, and display it.

See also: **– takeDoubleValueFrom:** (Control), **– setDoubleValue:**, **– doubleValue**

takeFloatValueFrom:

– **takeFloatValueFrom:***sender*

Sets the receiving Cell's single-precision floating-point value to the value returned by *sender*'s **floatValue** method. Returns **self**.

This is the same as **takeDoubleValueFrom:** except it works with floats rather than doubles.

See also: **– takeFloatValueFrom:** (Control), **– setFloatValue:**, **– floatValue**

takeIntValueFrom:

– **takeIntValueFrom:***sender*

Sets the receiving Cell's integer value to the value returned by *sender*'s **intValue** method. Returns **self**.

This is the same as **takeDoubleValueFrom:** except it works with **int**s rather than **double**s.

See also: **– takeIntValueFrom:** (Control), **– setIntValue:**, **– intValue**

takeStringValueFrom:

– **takeStringValueFrom:***sender*

Sets the receiving Cell's string value to the value returned by *sender*'s **stringValue** method. Returns **self**.

This is the same as **takeDoubleValueFrom:** except it works with strings rather than doubles.

See also: **– takeStringValueFrom:** (Control), **– stringValue**, **– setStringValue:**

target

– **target**

Returns **nil**. This method is overridden by ActionCell and its subclasses that implement **target** and **action** instance variables. Returns **self**.

trackMouse:inRect:ofView:

– (BOOL)**trackMouse:**(NXEvent *)*theEvent*
  inRect:(const NXRect *)*cellFrame*
  ofView:*controlView*

This method is called by Controls to implement the tracking behavior of a Cell. It's generally not overridden since the default implementation provides a simple interface to some other, simpler, tracking routines:

```
(BOOL)startTrackingAt:(NXPoint *)startPoint
    inView:controlView
(BOOL)continueTracking:(NXPoint *)lastPoint
    at:(NXPoint *)currentPoint
    inView:controlView
stopTracking:(NXPoint *)lastPoint
    at:(NXPoint *)endPoint
    inView:controlView
    mouseIsUp:(BOOL)flag
```

This method invokes **startTrackingAt:inView:** first, then, as mouse-dragged events are intercepted, **continueTracking:at:inView:** is called, and, finally, when the mouse leaves the bounds (if *cellFrame* is NULL, then the bounds are considered infinitely large), or if the mouse button goes up, **stopTracking:at:inView:mouseIsUp:** is called. If this interface is insufficient for the needs of your Cell, you may override **trackMouse:inRect:ofView:** directly. It's this method's responsibility to invoke the controlView's **sendAction:to:** method when appropriate (before, during, or after tracking) and to return YES if and only if the mouse goes up within the Cell during tracking. If the Cell's action is sent on mouse down, then **startTrackingAt:inView:** is called *before* the action is sent and the mouse is tracked until it goes up or out of bounds. If the Cell sends its action periodically, then the action is sent periodically to the target even if the mouse is not moving (although **continueTracking:at:inView:** is only called when the mouse changes position). If the Cell's action is sent on mouse dragged, then **continueTracking:at:inView:** is called *before* the action is sent. The state of the Cell is incremented (via **incrementState**) *before* the action is sent and *after* **stopTracking:at:inView:** is called when the mouse goes up. Returns **self**.

type

– (int)**type**

Returns the type of the Cell. Can be one of NX_NULLCELL, NX_ICONCELL or NX_TEXTCELL.

write:

– **write:**(NXTypedStream *)*stream*

Writes the Cell to the typed stream *stream*. Returns **self**.

CONSTANTS AND DEFINED TYPES

```
/* Cell Data Types */
#define NX_ANYTYPE                 0
#define NX_INTTYPE                 1
#define NX_POSINTTYPE              2
#define NX_FLOATTYPE               3
#define NX_POSFLOATTYPE            4
#define NX_DATETYPE                5
#define NX_DOUBLETYPE              6
#define NX_POSDOUBLETYPE           7

/* Cell Types */
#define NX_NULLCELL                0
#define NX_TEXTCELL                1
#define NX_ICONCELL                2

/* Cell & ButtonCell */
#define NX_CELLDISABLED            0
#define NX_CELLSTATE               1
#define NX_CELLEDITABLE            3
#define NX_CELLHIGHLIGHTED         5
#define NX_LIGHTBYCONTENTS         6
#define NX_LIGHTBYGRAY             7
#define NX_LIGHTBYBACKGROUND       9
#define NX_ICONISKEYEQUIVALENT    10
#define NX_HASALPHA               11
#define NX_BORDERED               12
#define NX_OVERLAPPINGICON        13
#define NX_ICONHORIZONTAL         14
#define NX_ICONONLEFTORBOTTOM     15
#define NX_CHANGECONTENTS         16

/* ButtonCell icon positions */
#define NX_TITLEONLY               0
#define NX_ICONONLY                1
#define NX_ICONLEFT                2
#define NX_ICONRIGHT               3
#define NX_ICONBELOW               4
#define NX_ICONABOVE               5
#define NX_ICONOVERLAPS            6
```

```
/* ButtonCell highlightsBy and showsStateBy mask */
#define NX_NONE                  0
#define NX_CONTENTS              1
#define NX_PUSHIN                2
#define NX_CHANGEGRAY            4
#define NX_CHANGEBACKGROUND      8

/* Cell whenActionIsSent mask flag */
#define NX_PERIODICMASK (1 << (NX_LASTEVENT+1))
```

ClipView

INHERITS FROM	View : Responder : Object
DECLARED IN	appkit/ClipView.h

CLASS DESCRIPTION

The ClipView class provides basic scrolling behavior by displaying a portion of a document that may be larger than the ClipView's frame rectangle. It also provides clipping to ensure that its document is not drawn outside the ClipView's frame. The ClipView has one subview, the *document view*, which is the view to be scrolled. Since a subview's coordinate system is positioned relative to its superview's origin, the ClipView changes the displayed portion of the document by translating the origin of its own bounds rectangle.

When the ClipView is instructed to scroll its document view, it copies as much of the previously visible document as possible, unless it received a **setCopyOnScroll:NO** message. The ClipView then sends its document view a message to either display or mark as invalidated the newly exposed region(s) of the ClipView. By default it will invoke the document view's **display::** method, but if the ClipView received a **setDisplayOnScroll:NO** message, it will invoke the document view's **invalidate::** method.

The ClipView sends its superview (usually a ScrollView) a **reflectScroll:** message to notify it whenever the relationship between the ClipView and the document view has changed. This allows the superview to update any controls it manages to reflect the change. You don't normally use the ClipView class directly; it is used by ScrollView which provides standard controls to allow the user to perform scrolling. However, you might use the ClipView class to implement a class similar to ScrollView.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from View	NXRect	frame;
	NXRect	bounds;
	id	superview;
	id	subviews;
	id	window;
	struct __vFlags	vFlags;
Declared in ClipView	float	backgroundGray;
	id	docView;
	id	cursor;

backgroundGray	The gray value used to fill the area of the ClipView not covered by the opaque portions of the document view.
docView	The ClipView's document view.
cursor	The cursor that's used inside the ClipView's frame rectangle.

METHOD TYPES

Initializing the class object	+ initialize
Initializing and freeing a ClipView	– initFrame: – free
Modifying the frame rectangle	– moveTo:: – rotateTo: – sizeTo::
Modifying the coordinate system	– rotate: – scale:: – setDrawOrigin:: – setDrawRotation: – setDrawSize:: – translate::
Managing component Views	– docView – setDocView: – getDocRect: – getDocVisibleRect: – resetCursorRects – setDocCursor:
Modifying graphic attributes and displaying	– backgroundGray – setBackgroundGray: – backgroundColor – setBackgroundColor: – drawSelf::
Scrolling	– autoscroll: – constrainScroll: – rawScroll: – setCopyOnScroll: – setDisplayOnScroll:
Coordinating with other Views	– descendantFlipped: – descendantFrameChanged:

Archiving	– awake – read: – write:

CLASS METHODS

initialize

+ initialize

Sets the current version of the ClipView class. You never send an **initialize** message; it's sent for you when the application starts. Returns **self**.

INSTANCE METHODS

autoscroll:

– **autoscroll:**(NXEvent *)*theEvent*

Performs automatic scrolling of the document. This message is sent to the document view when the mouse is dragged from a position within the ClipView to a position outside it. The document view then sends this message to its ClipView. You never send an **autoscroll:** message directly to a ClipView. Returns **nil** if no scrolling occurs; otherwise returns **self**.

See also: – **autoscroll:** (View)

awake

– **awake**

Overrides View's **awake** method to ensure additional initialization. After a ClipView has been read from an archive file, it will receive this message. You should not invoke this method directly. Returns **self**.

backgroundColor

– (NXColor)**backgroundColor**

Returns the color of the ClipView's background. If the background gray value has been set but no color has been set, the color equivalent of the background gray value is returned. If neither value has been set, the background color of the ClipView's window is returned.

See also: – **backgroundGray**, – **setBackgroundColor:**, – **setBackgroundGray:**, – **backgroundColor** (Window), **NXConvertGrayToColor()**

backgroundGray

– (float)**backgroundGray**

Returns the gray value of the ClipView's background. If no value has been set, the gray value of the ClipView's window is returned.

See also: – **backgroundColor**, – **setBackgroundGray:**,
– **backgroundGray** (Window)

constrainScroll:

– **constrainScroll:**(NXPoint *)*newOrigin*

Ensures that the document view is not scrolled to an undesirable position. This method is invoked by the private method that all scrolling messages go through before it invokes **rawScroll:** or **scrollClip:to:**. The default implementation keeps as much of the document view visible as possible. You may want to override this method to provide alternate constraining behavior. *newOrigin* is the desired new origin of the ClipView's bounds rectangle and is given in ClipView coordinates. Returns **self**.

See also: – **rawScroll:**

descendantFlipped:

– **descendantFlipped:***sender*

Notifies the ClipView that the orientation of the coordinate system of its document view has changed (from unflipped to flipped, or vice versa). The orientation of the ClipView is changed to match the orientation of its document view. You should not invoke this method directly, or override it. Returns **self**.

See also: – **notifyWhenFlipped:** (View), – **setDocView:**

descendantFrameChanged:

– **descendantFrameChanged:***sender*

Notifies the ClipView that its document view has been resized or moved. The ClipView may then scroll and/or display the document view, and the Clipview may also notify its superview to reflect the changes in the scroll position. You should not invoke this method directly, or override it. Returns **self**.

See also: – **moveTo::** (View), – **sizeTo::** (View), – **reflectScroll:** (ScrollView),
– **notifyAncestorWhenFrameChanged:** (View), – **setDocView:**

docView

– **docView**

Returns the ClipView's document view.

See also: – **setDocView:**

drawSelf::

– **drawSelf:**(const NXRect *)*rects* **:**(int)*rectCount*

Overrides View's **drawSelf::** method to fill the portions of the ClipView that are not covered by opaque portions of the document view. If a color value has been set and the ClipView is drawing itself on a color screen, the ClipView draws its background with the color value, otherwise it draws its background using its background gray value. Returns **self**.

See also: – **backgroundColor:**, – **backgroundGray:**, – **drawSelf::** (View)

free

– **free**

Deallocates the memory used by the receiving ClipView. The ClipView is removed from the view hierarchy, and all its subviews are also freed.

getDocRect:

– **getDocRect:**(NXRect *)*aRect*

Places the ClipView's document rectangle into the structure specified by *aRect*. The origin of this rectangle is equal to the origin of the document view's frame rectangle. The document rectangle's height and width are set to the maximum corresponding values from the document view's frame size and the content view's bounds size. The document rectangle is used in conjunction with the ClipView's bounds rectangle to determine values for any indicators of relative position and size between the ClipView and the document view. The ScrollView uses these rectangles to set the size and position of the Scrollers' knobs. Returns **self**.

See also: – **reflectScroll:** (ScrollView)

getDocVisibleRect:

– **getDocVisibleRect:**(NXRect *)*aRect*

Gets the portion of the document view that's visible within the ClipView. The ClipView's bounds rectangle, transformed into the document view's coordinates, is placed in the structure specified by *aRect*. This rectangle represents the portion of the document view's coordinate space that's visible through the ClipView. However, this rectangle doesn't reflect the effects of any clipping that may occur above the ClipView itself. Thus, if the ClipView is itself clipped by one of its superviews, this method returns a different rectangle than the one returned by the **getVisibleRect:** method, because the latter reflects the effects of all clipping by superviews. Returns **self**.

See also: – **getVisibleRect:** (View)

initFrame:

– **initFrame:**(const NXRect *)*frameRect*

Initializes the ClipView, which must be a newly allocated ClipView instance. The ClipView's frame rectangle is made equivalent to that pointed to by *frameRect*. This method is the designated initializer for the ClipView class, and can be used to initialize a ClipView allocated from your own zone. By default, clipping is enabled, and the ClipView is set to opaque. A ClipView is initialized without a document view. Returns **self**.

See also: – **setDocView:**, – **initFrame:** (View), + **alloc** (Object), + **allocFromZone:** (Object)

moveTo::

– **moveTo**:(NXCoord)*x* :(NXCoord)*y*

Moves the origin of the ClipView's frame rectangle to (*x*, *y*) in its superview's coordinates. Returns **self**.

See also: – **moveTo::** (View)

rawScroll:

– **rawScroll:**(const NXPoint *)*newOrigin*

Performs scrolling of the document view. This method sets the ClipView's bounds rectangle origin to *newOrigin*. Then, it copies as much of the previously visible document as possible, unless it received a **setCopyOnScroll:NO** message. It then sends its document view a message to either display or mark as invalidated the newly exposed region(s) of the ClipView. By default it will invoke the document view's **display::** method, but if the ClipView received a **setDisplayOnScroll:NO** message, it will invoke the document view's **invalidate::** method. The **rawScroll:** method does not send a **reflectScroll:** message to its superview; that message is sent by the method that invokes **rawScroll:**. Note also that while the ClipView provides clipping to its frame, it does not clip to the update rectangles.

This method is used by a private method through which all scrolling passes, and is invoked if the ClipView's superview does not implement the **scrollClip:to:** method. If the ClipView's superview does implement **scrollClip:to:**, that method should invoke **rawScroll:**. The ClipView's typical superview (Scrollview) doesn't implement the **scrollClip:to:** method. This mechanism is provided so that the ClipView's superview can coordinate scrolling of multiple tiled ClipViews. Returns **self**.

read:

– **read:**(NXTypedStream *)*stream*

Reads the ClipView and its document view from the typed stream *stream*. Returns **self**.

See also: – **write:**

resetCursorRects

– **resetCursorRects**

Resets the cursor rectangle for the document view to the bounds of the ClipView. Returns **self**.

See also: – **setDocCursor:**, – **addCursorRect:cursor:** (View)

rotate:

– **rotate:**(NXCoord)*angle*

Disables rotation of the ClipView's coordinate system. You also should not rotate the ClipView's document view, nor should you install a ClipView as a subview of a rotated view. The proper way to rotate objects in the document view is to perform the rotation in your document view's **drawSelf::** method. Returns **self**.

rotateTo:

– **rotateTo:**(NXCoord)*angle*

Disables rotation of the ClipView's frame rectangle. This method also disables ClipView's inherited **rotateBy:** method. Returns **self**.

See also: – **rotate:**

scale::

– **scale**:(NXCoord)*x* :(NXCoord)*y*

Rescales the ClipView's coordinate system by a factor of *x* for its x axis, and by a factor of *y* for its y axis. Since the document view's coordinate system is measured relative to the ClipView's coordinate system, the document view is redisplayed and a **reflectScroll:** message may be sent to the ClipView's superview. Returns **self**.

See also: – **reflectScroll:** (ScrollView)

setBackgroundColor:

– **setBackgroundColor:**(NXColor)*color*

Sets the color of the ClipView's background. This color is used to fill the area inside the ClipView that's not covered by opaque portions of the document view. If no background gray has been set for the ClipView, this method sets it to the gray component of the color. Returns **self**.

See also: – **backgroundColor**, – **backgroundGray**, – **setBackgroundGray**, **NXGrayComponent()**

setBackgroundGray:

– **setBackgroundGray:**(float)*value*

Sets the gray value of the ClipView's background. This gray is used to fill the area inside the ClipView that's not covered by opaque portions of the document view. *value* must lie in the range from 0.0 (black) to 1.0 (white). Returns **self**.

See also: – **backgroundColor**, – **backgroundGray**, – **setBackgroundColor**

setCopyOnScroll:

– **setCopyOnScroll:**(BOOL)*flag*

Determines whether the buffered bits will be copied when scrolling occurs. If *flag* is YES, scrolling will copy as much of the ClipView's bitmap as possible to scroll the view, and the document view is responsible only for updating the newly exposed portion of itself. If *flag* is NO, the document view is responsible for updating the entire ClipView. This should only rarely be changed from the default value (YES). Returns **self**.

setDisplayOnScroll:

– **setDisplayOnScroll:**(BOOL)*flag*

Determines whether the results of scrolling will be immediately displayed. If *flag* is YES, the results of scrolling will be immediately displayed. If *flag* is NO, the ClipView is marked as invalid but is not displayed. In either case, when a scroll occurs, the ClipView first copies as much of its buffered bitmap as possible, assuming the default case where **setCopyOnScroll:YES** was sent. This should only rarely be changed from the default value (YES). Returns **self**.

See also: – **rawScroll:**, – **display::** (View), – **invalidate**:: (View)

setDocCursor:

– **setDocCursor:***anObj*

Sets the cursor to be used inside the ClipView's bounds. *anObj* should be a NXCursor object. Returns the old cursor.

setDocView:

– **setDocView:***aView*

Sets *aView* as the ClipView's document view. There is one document view per ClipView, so if there was already a document view for this ClipView it is replaced. This method initializes the document view with **notifyAncestorWhenFrameChanged:YES** and **notifyWhenFlipped:YES** messages. The origin of the document view's frame is initially set to be coincident with the origin of the ClipView's bounds. If the ClipView is contained within a ScrollView, you should send the ScrollView the **setDocView:** message and have the ScrollView pass this message on to the ClipView. Returns the old document view, or **nil** if there was none.

See also: – **setDocView:** (ScrollView)

setDrawOrigin::

– **setDrawOrigin**:(NXCoord)*x* :(NXCoord)*y*

Overrides the View method so that changes in the ClipView's coordinate system are reflected in the displayed document view. This method may redisplay the document view, and a **reflectScroll:** message may be sent to the ClipView's superview. Returns **self**.

See also: – **setDrawOrigin::** (View)

setDrawRotation:

– **setDrawRotation**:(NXCoord)*angle*

Disables rotation of the ClipView's coordinate system. The proper way to rotate objects in the document view is to perform the rotation in your document view's **drawSelf::** method. Returns **self**.

See also: – **rotate:**

setDrawSize::

– **setDrawSize**:(NXCoord)*width* :(NXCoord)*height*

Overrides the View method so that rescaling of the ClipView's coordinate system is reflected in the displayed document view. This method may redisplay the document view, and a **reflectScroll:** message may be sent to the ClipView's superview. Returns **self**.

See also: – **setDrawSize::** (View)

sizeTo::

– **sizeTo**:(NXCoord)*width* :(NXCoord)*height*

Overrides the View method so that resizing of the ClipView's frame rectangle is reflected in the displayed document view. This method may redisplay the document view, and a **reflectScroll:** message may be sent to the ClipView's superview. Returns **self**.

See also: – **sizeTo::** (View)

translate::

– **translate**:(NXCoord)*x* :(NXCoord)*y*

Overrides the View method so that translation of the ClipView's coordinate system is reflected in the displayed document view. This method may redisplay the document view, and a **reflectScroll:** message may be sent to the ClipView's superview. Returns **self**.

See also: – **translate::** (View)

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving ClipView and its document view to the typed stream *stream*. Returns **self**.

See also: – **write:**

METHODS IMPLEMENTED BY CLIPVIEW'S SUPERVIEW

reflectScroll:

– **reflectScroll:***aClipView*

Notifies the ClipView's superview that either the ClipView's bounds rectangle or the document view's frame rectangle has changed, and that any indicators of the scroll position need to be adjusted. ScrollView implements this method to update its Scrollers.

scrollClip:to:

– **scrollClip:***aClipView* **to:**(const NXPoint *)*newOrigin*

Notifies the ClipView's superview that the ClipView needs to set its bounds rectangle origin to *newOrigin*. The ClipView's superview should then send the ClipView the **rawScroll:** message. This mechanism is provided so that the ClipView's superview can coordinate scrolling of multiple tiled ClipViews. Note that the default ScrollView class does not implement this method.

See also: – **rawScroll:** (ClipView)

Control

INHERITS FROM	View : Responder : Object
DECLARED IN	appkit/Control.h

CLASS DESCRIPTION

Control is an abstract superclass that provides three fundamental features for implementing user interface devices. First, as a subclass of View, Control has a bounds rectangle in which to draw the on-screen representation of the device. Second, it provides a **mouseDown:** method and a position in the responder chain; together these features enable Control to receive and respond to user-generated events within its bounds. Third, it implements the **sendAction:to:** method through which Control sends an action message to its target object. Subclasses of Control defined in the Application Kit are Button, Form, Matrix, NXBrowser, NXColorWell, Slider, Scroller, and TextField.

Target objects and action methods provide the mechanism by which Controls interact with other objects in an application. A target is an object that a Control has affect over. An action method is defined by the target class to enable its instances to respond to user input; the **id** of the Control is the only argument to the action method. When it receives an action message, a target can use the **id** to send a message requesting additional information from the Control about its status. Targets and actions can be set explicitly by application code. You can also set the target to **nil** and allow it to be determined at run time. When the target is **nil**, the Control that's about to send an action message must look for an appropriate receiver. It conducts its search in a prescribed order:

- It begins with the first responder in the current key window and follows **nextResponder** links up the responder chain to the Window object. After the Window object, it tries the Window's delegate.
- If the main window is different from the key window, it then starts over with the first responder in the main window and works its way up the main window's responder chain to the Window object and its delegate.
- Next, it tries the Application object, NXApp, and finally the Application object's delegate. NXApp and its delegate are the receivers of last resort.

Control provides methods for setting and using the target object and action method. However, these methods require that Control's **cell** instance variable be set to some subclass of Cell that provides the instance variables **target** and **action**, such as ActionCell and its subclasses.

Target objects and action methods demonstrate the close relationship between Controls and Cells. In most cases, a user interface device consists of an instance of a Control subclass paired with one or more instances of a Cell subclass. Each implements specific details of the user interface mechanism. For example, Control's **mouseDown:**

method sends a **trackMouse:inRect:ofView:** message to Cell, which handles subsequent mouse and keyboard events; Cell sends Control a **sendAction:to:** message in response to particular events. Control's **drawSelf::** method is implemented by sending a **drawSelf:inView:** message to Cell. As another example, Control provides methods for setting and formatting its contents; these methods send corresponding messages to Cell, which owns the **contents** instance variable.

A Control subclass doesn't have to use a Cell subclass to implement itself; Scroller and NXColorWell don't. However, such subclasses have to take care of details that Cell would otherwise handle. Specifically, they have to overwrite methods designed to work with a Cell. What's more, they cannot be used in a Matrix—a subclass of Control designed specifically for managing multiple Cell arrays such as radio buttons. You usually implement a unique user interface device by creating a subclass of Cell or ActionCell rather than Control.

In general, Interface Builder is the easiest way to add both kit-defined and your own subclasses of Control to an application.

The **initFrame:** method is the designated initializer for the Control class. Override this method if you create a subclass of Control that performs its own initialization.

See also: ActionCell, Cell

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from View	NXRect	frame;
	NXRect	bounds;
	id	superview;
	id	subviews;
	id	window;
	struct __vFlags	vFlags;
Declared in Control	int	tag;
	id	cell;
	struct _conFlags{	
	unsigned int	enabled:1;
	unsigned int	editingValid:1;
	unsigned int	ignoreMultiClick:1;
	unsigned int	calcSize:1;
	}	conFlags;

tag	An integer that identifies the Control; can be used by View's **findViewWithTag:** method to find a Control in a view hierarchy.

cell	The **id** of the Control's cell (if it has only one).
conFlags.enabled	True if the Control is enabled; relevant for multi-cell controls only.
conFlags.editingValid	True if editing has been validated.
conFlags.ignoreMultiClick	True if the Control ignores double- or triple-clicks.
conFlags.calcSize	True if the cell should recalculate its size and location before drawing.

METHOD TYPES

Initializing and freeing a Control	– initFrame: – free
Setting the Control's Cell	– cell – setCell: + setCellClass:
Enabling and disabling the Control	– isEnabled – setEnabled:
Identifying the selected Cell	– selectedCell – selectedTag
Setting the Control's value	– setFloatValue: – floatValue – setDoubleValue: – doubleValue – setIntValue: – intValue – setStringValue: – setStringValueNoCopy: – setStringValueNoCopy:shouldFree: – stringValue
Formatting text	– setFont: – font – setAlignment: – alignment – setFloatingPointFormat:left:right:
Managing the field editor	– abortEditing – currentEditor – validateEditing

Managing the cursor	– resetCursorRects
Interacting with other Controls	– takeDoubleValueFrom: – takeFloatValueFrom: – takeIntValueFrom: – takeStringValueFrom:
Resizing the Control	– calcSize – sizeTo:: – sizeToFit
Displaying the Control and Cell	– drawCell: – drawCellInside: – drawSelf:: – selectCell: – update – updateCell: – updateCellInside:
Target and action	– action – isContinuous – sendAction:to: – sendActionOn: – setAction: – setContinuous: – setTarget: – target
Assigning a tag	– setTag: – tag
Tracking the mouse	– ignoreMultiClick: – mouseDown: – mouseDownFlags
Archiving	– read: – write:

CLASS METHODS

setCellClass:

+ **setCellClass:***classId*

This abstract method does nothing and returns the **id** of the receiver. It's implemented by subclasses of Control, which use this method to set their **cell** instance variable.

INSTANCE METHODS

abortEditing

– **abortEditing**

Terminates and discards any editing of text displayed by the receiving Control. Returns **self** or, if no editing was going on in the receiving Control, **nil**. Does not redisplay the old value of the Control—you must explicitly do that.

See also: – **endEditingFor:** (Window), – **validateEditing**

action

– (SEL)**action**

Returns the action message sent by the Control. To get the action message, this method sends an **action** message to the Control's **cell**.

See also: – **setAction:**

alignment

– (int)**alignment**

Returns the justification mode. The return value can be one of three constants: NX_LEFTALIGNED, NX_CENTERED or NX_RIGHTALIGNED.

calcSize

– **calcSize**

Recomputes any internal sizing information for the Control, if necessary, by invoking **calcDrawInfo:** on its cell. This can be useful for caching any information needed to make the drawing of a cell faster. Does not draw. Can be used for more sophisticated sizing operations as well (for example, Form). This is automatically invoked whenever the Control is displayed and something has changed (as recorded by the **calcSize** flag).

See also: – **calcSize** (Matrix, Form), – **sizeToFit**

cell

– **cell**

Returns the Control's cell. Should not be used by the action method of the target of the Control (use **selectedCell**).

currentEditor

– **currentEditor**

If the receiving Control is being edited (that is, the user is typing or selecting text in the Control), this method returns the editor (the Text object) being used to perform that editing. If the Control is not being edited, this method returns **nil**.

doubleValue

– (double)**doubleValue**

Returns the value of the Control as a double-precision floating point number. If the Control contains many cells (for example, Matrix), then the value of the currently **selectedCell** is returned. If the Control is in the process of editing the affected cell, then **validateEditing** is invoked before the value is extracted and returned.

See also: – **setDoubleValue:**

drawCell:

– **drawCell:***aCell*

If *aCell* is the cell used to implement this Control, then the Control is displayed. This is provided primarily in support of a consistent interface with a multiple cell Control's **drawCell:**. Returns **self**.

See also: – **drawCell:** (Matrix), – **updateCell:**

drawCellInside:

– **drawCellInside:***aCell*

Same as **drawCell:** except that only the "inside" of the Control is drawn (using the cell's **drawInside:inView:** method). This method is used by **setStringValue:** and similar content-setting methods to provide a minimal update of the Control when its value is changed. Returns **self**.

See also: – **drawInside:inView:** (Cell), – **drawCellInside:** (Matrix),
– **updateCellInside:**

drawSelf::

– **drawSelf:**(const NXRect *)*rects* **:**(int)*rectCount*

Draws the Control. It simply invokes the Control's cell's **drawSelf:inView:** method. Must override if you have a multi-cell control. Returns **self**.

floatValue

– (float)**floatValue**

Returns the value of the Control as a single-precision floating point number (see **doubleValue** for more details).

See also: **– setFloatValue:**

font

– font

Returns the font object used to draw the text (if any) of the Control.

free

– free

Frees the memory used by the Control and its cells. Aborts editing if the text of the Control was currently being edited. Returns **nil**.

ignoreMultiClick:

– **ignoreMultiClick:**(BOOL)*flag*

Sets the Control to ignore multiple clicks if *flag* is **YES**. By default, double-clicks (and higher order clicks) are treated the same as single clicks. You can use this method to "debounce" a control.

initFrame:

– **initFrame:**(const NXRect *)*frameRect*

Initializes and returns the receiver, a new instance of Control, by setting *frameRect* as its frame rectangle. Sets the new instance as opaque. Since Control is an abstract class, messages to perform this method should appear only in subclass methods. **initFrame:** is the designated initializer for the Control class.

intValue

– (int)**intValue**

Returns the value of the Control as an integer (see **doubleValue** for more details).

See also: – **setIntValue:**

isContinuous

– (BOOL)**isContinuous**

Returns **YES** if the Control continuously sends its action to its target during mouse tracking.

See also: – **setContinuous:**

isEnabled

– (BOOL)**isEnabled**

Returns **YES** if the Control is enabled, **NO** otherwise.

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

Invoked when the mouse button goes down while the cursor is within the bounds of the Control. The Control is highlighted and the Control's Cell tracks the mouse until it goes outside the bounds, at which time the Control is unhighlighted. If the cursor goes back into the bounds, then the Control highlights again and its Cell starts tracking again. This behavior continues until the mouse button goes up. If it goes up with the cursor in the Control, the state of the Control is changed, and the action is sent to the target. If the mouse button goes up with the cursor outside the Control, no action is taken.

mouseDownFlags

– (int)**mouseDownFlags**

Returns the event flags (for example, NX_SHIFTMASK) that were in effect at the beginning of tracking. The flags are valid only in the action method that is sent to the Control's target.

See also: – **mouseDownFlags** (Cell), – **sendAction:to:**

read:

– **read:**(NXTypedStream *)*stream*

Reads the Control from the specified typed stream *stream*.

resetCursorRects

– **resetCursorRects**

If the Control's cell is editable, then **resetCursorRect:inView:** is sent to the cell (which will, in turn, send **addCursorRect:cursor:** back to the Control). Causes the cursor to be an I-beam when the mouse is over the editable portion of the cell.

selectCell

– **selectCell:***aCell*

If *aCell* is the receiving Control's cell and is currently unselected, this method selects *aCell* and redraws the Control. Returns **self**.

selectedCell

– **selectedCell**

This method should be used by the target of the Control when it wants to get the cell of the sending Control. Note that even though the **cell** method will return the same value for single cell Controls, it's strongly suggested that this method be used since it will work both for multiple and single cell Controls.

See also: – **sendAction:to:**, – **selectedCell** (Matrix)

selectedTag

– (int)**selectedTag**

The action method in the target of the Control should use this method to get the identifying tag of the sending Control's cell. You should use the **setTag:** and **tag** methods in conjunction with **findViewWithTag:**. This is equivalent to [[self **selectedCell**] **tag**]. Returns –1 if there is no currently **selectedCell**. The cell's tag can be set with ActionCell's **setTag:** method. When you set a single-cell Control's tag in Interface Builder, it sets both the Control's and the cell's tag (as a convenience).

See also: – **sendAction:to:**

sendAction:to:

– **sendAction:**(SEL)*theAction* **to:***theTarget*

Sends a **sendAction:to:** message to NXApp, which in turn sends a message to *theTarget* to perform *theAction*. This method adds the Control's **id** as the action method's only argument. If *theAction* is NULL, no message is sent.

This method is invoked primarily by Cell's **trackMouse:inRect:ofView:**.

If *theTarget* is nil, NXApp looks for an object that can respond to the message—that is, for an object that implements a method matching the *theAction* selector. It begins with the first responder of the key window. If the first responder can't respond, it tries the first responder's next responder and continues following next responder links up the responder chain. If none of the objects in the key window's responder chain can handle the message and if the main window is different from the key window, it begins again with the first responder in the main window. If objects in neither the key window nor the main window can respond, NXApp tries to handle the message itself. If NXApp cannot respond, then the message is sent to NXApp's delegate.

Returns **nil** if the message could not be delivered; otherwise returns **self**.

See also: – **setAction:**, – **setTarget:**, – **trackMouse:inRect:ofView:** (Cell)

sendActionOn:

– (int)**sendActionOn:**(int)*mask*

Sets a mask of the events that cause **sendAction:to:** to be invoked during tracking of the mouse (done in Cell's **trackMouse:inRect:ofView:**). Returns the old event mask.

See also: – **sendActionOn:** (Cell), – **trackMouse:inRect:ofView:** (Cell)

setAction:

– **setAction:**(SEL)*aSelector*

Makes *aSelector* the Control's action method.

See also: – **sendAction:to:**

setAlignment:

– **setAlignment:**(int)*mode*

Sets the justification mode. The *mode* should be one of: NX_LEFTALIGNED, NX_CENTERED or NX_RIGHTALIGNED.

setCell:

– **setCell:***aCell*

Sets the cell of the Control to be *cell.* Use this method with great care as it can irrevocably damage your Control. Returns the old cell.

setContinuous:

– **setContinuous:**(BOOL)*flag*

Sets whether the Control should continuously send its action to its target as the mouse is tracked.

See also: – **setContinuous:** (ButtonCell, SliderCell), – **sendActionOn:**

setDoubleValue:

– **setDoubleValue:**(double)*aDouble*

Sets the value of the Control to be *aDouble* (a double-precision floating point number). If the Control contains many cells, then the currently **selectedCell**'s value is set to *aDouble.* If the affected cell is currently being edited, then that editing is aborted and the value being typed is discarded in favor of *aDouble.* If autodisplay is on, then the cell's "inside" is redrawn.

See also: – **doubleValue**, – **abortEditing**, – **drawInside:inView:** (Cell)

setEnabled:

– **setEnabled:**(BOOL)*flag*

Sets the flag determining whether the Control is active or not. Redraws the entire Control if autodisplay is on. Subclasses may want to override this to redraw only a portion of the Control when the enabled state changes (Button and Slider do this).

setFloatValue:

– **setFloatValue:**(float)*aFloat*

Same as **setDoubleValue:**, but sets the value as a single-precision floating point number.

See also: – **floatValue**

setFloatingPointFormat:left:right:

– **setFloatingPointFormat:**(BOOL)*autoRange*
left:(unsigned)*leftDigits*
right:(unsigned)*rightDigits*

Sets the floating point number format of the Control. Does not redraw the cell. Affects only subsequent settings of the value using **setFloatValue:**.

See also: – **setFloatPointFormat:left:right:** (Cell)

setFont:

– **setFont:***fontObj*

Sets the font used to draw the text (if any) in the Control. You only need to invoke this method if you do not want to use the default font (Helvetica 12.0). If autodisplay is on, then the inside of the cell is redrawn.

setIntValue:

– **setIntValue:**(int)*anInt*

Same as **setDoubleValue:**, but sets the value as an integer.

See also: – **intValue**

setStringValue:

– **setStringValue:**(const char *)*aString*

Same as **setDoubleValue:**, but sets the value as a string.

See also: – **stringValue**, – **setStringValueNoCopy:**, – **setIntValue:**

setStringValueNoCopy:

– **setStringValueNoCopy:**(const char *)*aString*

Like **setStringValue:**, but doesn't copy the string.

See also: – **stringValue**, – **setStringValue:**, – **setStringValueNoCopy:shouldFree:**

setStringValueNoCopy:shouldFree:

– **setStringValueNoCopy:**(char *)*aString* **shouldFree:**(BOOL)*flag*

Like **setStringValueNoCopy:**, but lets you specify whether the string should be freed when the Control is freed.

See also: – **stringValue**, – **setStringValueNoCopy:**

setTag:

– **setTag:**(int)*anInt*

Makes *anInt* the receiving Control's tag.

See also: – **tag**, – **selectedTag**, – **findViewWithTag:** (View)

setTarget:

– **setTarget:***anObject*

Sets the target for the Control's action message.

See also: – **sendAction:to:**

sizeTo::

– **sizeTo:**(NXCoord)*width* **:**(NXCoord)*height*

Changes the width and the height of the Control's frame. Redisplays the Control if autodisplay is on.

sizeToFit

– **sizeToFit**

Changes the width and the height of the Control's frame so that they get the minimum size to contain the cell. If the Control has more than one cell, then you must override this method.

See also: – **sizeToFit** (Matrix), – **sizeToCells** (Matrix)

stringValue

– (const char *)**stringValue**

Returns the value of the Control as a string (see **doubleValue** for more details).

See also: – **setStringValue:**, – **setStringValueNoCopy:**

tag

– (int)**tag**

Returns the receiving Control's tag (not the Control's cell's tag).

See also: – **setTag:**, – **selectedTag**

takeDoubleValueFrom:

– **takeDoubleValueFrom:***sender*

Sets the receiving Control's double-precision floating-point value to the value obtained by sending a **doubleValue** message to *sender.*

This method can be used in action messages between Controls. It permits one Control (*sender*) to affect the value of another Control (the receiver) by sending this method in an action message to the receiver. For example, a TextField can be made the target of a Slider. Whenever the Slider is moved, it will send a **takeDoubleValueFrom:** message to the TextField. The TextField will then get the Slider's floating-point value, turn it into a text string, and display it, thus tracking the value of the Slider.

See also: – **setDoubleValue:**, – **doubleValue**

takeFloatValueFrom:

– **takeFloatValueFrom:***sender*

Sets the receiving Control's single-precision floating-point value to the value obtained by sending a **floatValue** message to *sender.*

See **setDoubleValue:** for an example.

See also: – **setFloatValue:**, – **floatValue**

takeIntValueFrom:

– **takeIntValueFrom:***sender*

Sets the receiving Control's integer value to the value returned by sending an **intValue** message to *sender.*

See **setDoubleValue:** for an example.

See also: – **setIntValue:**, – **intValue**

takeStringValueFrom:

– **takeStringValueFrom:***sender*

Sets the receiving Control's character string to a string obtained by sending a **stringValue** message to *sender.*

See **setDoubleValue:** for an example.

See also: – **stringValue**, – **setStringValue:**

target

– **target**

Returns the target for the Control's action message.

See also: – **setTarget:**

update

– **update**

Same as View's **update**, but also makes sure that **calcSize** gets performed.

updateCell:

– **updateCell:***aCell*

If *aCell* is a cell used to implement this Control, and if autodisplay is on, then draws the Control; otherwise, sets the **needsDisplay** and **calcSize** flags to YES.

updateCellInside:

– **updateCellInside:***aCell*

If *aCell* is a cell used to implement this Control, and if autodisplay is on, draws the inside portion of the cell; otherwise sets the **needsDisplay** flag to YES.

validateEditing

– **validateEditing**

Causes the value of the field currently being edited (if any) to be absorbed as the value of the Control. Invoked automatically from **stringValue**, **intValue**, and others, so that partially edited field's values will be reflected in the values returned by those methods.

This method doesn't end editing; to do that, invoke Window's **endEditingFor:** or **abortEditing**.

See also: – **endEditingFor:** (Window)

write:

– **write:**(NXTypedStream *)*stream*

Writes the Control to the specified typed stream *stream.*

Font

INHERITS FROM	Object
DECLARED IN	Font.h

CLASS DESCRIPTION

The Font class provides objects that correspond to PostScript fonts. Each Font object records a font's name, size, style, and matrix. When a Font object receives a **set** message, it establishes its font as the current font in the Window Server's current graphics state.

For a given application, only one Font object is created for a particular PostScript font. When the Font class object receives a message to create a new object for a particular font, it first checks whether one has already been created for that font. If so, it returns the **id** of that object; otherwise, it creates a new object and returns its **id**. This system of sharing Font objects minimizes the number of objects created. It also implies that no one object in your application can know whether it has the only reference to a particular Font object. Thus, Font objects shouldn't be freed; Font's **free** method simply returns **self**.

A Font object's **fontNum** instance variable stores a number (a PostScript user object) that refers to the actual font dictionary within the Window Server. You shouldn't change the value of this variable.

INSTANCE VARIABLES

Inherited from Object

```
Class               isa;
```

Declared in Font

```
char                *name;
float               size;
int                 style;
float               *matrix;
int                 fontNum;
NXFaceInfo          *faceInfo;
id                  otherFont;
struct _fFlags {
    unsigned int        usedByWS:1;
    unsigned int        usedByPrinter:1;
    unsigned int        isScreenFont:1;
}                   fFlags;
```

name	The font's name.
size	The font's size.

style	The font's style.
matrix	The font's matrix.
fontNum	The user object referring to this font.
faceInfo	The font's face information.
otherFont	The associated screen font for this font.
fFlags.usedByWS	True if the font is stored in the Window Server.
fFlags.usedByPrinter	True if the font is stored in the printer.
fFlags.isScreenFont	True if the font is a screen font.

METHOD TYPES

Initializing the Class object	+ initialize + useFont:
Creating and freeing a Font object	+ newFont:size: + newFont:size:matrix: + newFont:size:style:matrix: – free
Querying the Font object	– fontNum – getWidthOf: – hasMatrix – matrix – metrics – name – pointSize – readMetrics: – screenFont – style
Setting the font	– set – setStyle:
Archiving	– awake – finishUnarchiving – read: – write:

CLASS METHODS

alloc

Disables the inherited **alloc** method to prevent multiple Font objects from being created for a single PostScript font. Create Font objects by using **newFont:size:style:matrix:**, **newFont:size:matrix:**, or **newFont:size:**. These methods ensure that no more than one Font object is created for any PostScript font. Returns an error message.

See also: + **newFont:size:style:matrix:**, + **newFont:size:matrix:**, + **newFont:size:**

allocFromZone:

Disables the inherited **allocFromZone:** method to prevent multiple Font objects from being created for a single PostScript font. Create Font objects by using **newFont:size:style:matrix:**, **newFont:size:matrix:**, or **newFont:size:**. These methods ensure that no more than one Font object is created for any PostScript font. Returns an error message.

See also: + **newFont:size:style:matrix:**, + **newFont:size:matrix:**, + **newFont:size:**

initialize

+ **initialize**

Initializes the Font class object. The class object receives an **initialize** message before it receives any other message. You never send an **initialize** message directly.

See also: + **initialize** (Object)

newFont:size:

+ **newFont:**(const char *)*fontName* **size:**(float)*fontSize*

Returns a Font object for font *fontName* of size *fontSize*. This method invokes the **newFont:size:style:matrix:** method with the style set to 0 and the matrix set to NX_FLIPPEDMATRIX.

See also: + **newFont:size:style:matrix:**, + **newFont:size:matrix:**

newFont:size:matrix:

+ **newFont:**(const char *)*fontName*
    size:(float)*fontSize*
    matrix:(const float *)*fontMatrix*

Returns a Font object for font *fontName* of size *fontSize*. This method invokes the **newFont:size:style:matrix:** method with the style set to 0.

See also: + **newFont:size:style:matrix:**, + **newFont:size:**

newFont:size:style:matrix:

+ **newFont:**(const char *)*fontName*
    size:(float)*fontSize*
    style:(int)*fontStyle*
    matrix:(const float *)*fontMatrix*

Returns a Font object for font *fontName*, of size *fontSize*, and matrix *fontMatrix*. *fontStyle* is currently ignored. If an appropriate Font object was previously created, it's returned; otherwise, a new one is created and returned. If an error occurs, this method returns **nil**. This is the designated **new...** method for the Font class.

There are two constants available for the *fontMatrix* parameter:

- NX_IDENTITYMATRIX. Use the identity matrix.
- NX_FLIPPEDMATRIX. Use a flipped matrix. (Appropriate for a flipped View like the Text object.)

The *fontStyle* parameter is stored in the Font object, and is preserved by the FontManager's **convertFont:** method, but is not used by the Application Kit. It can be used to store application-specific font information.

Note: If this method is invoked from a subclass (through a message to **super**), a new object is always created. Thus, your subclass should institute its own system for sharing Font objects.

See also: + **newFont:size:matrix:**, + **newFont:size:**

useFont:

+ **useFont:**(const char *)*fontName*

Registers that the font identified by *fontName* is used in the document. Returns **self**.

The Font class object keeps track of the fonts that are being used in a document. It does this by registering the font whenever a Font object receives a **set** message. When a document is called upon to generate a conforming PostScript language version of its text (such as during printing), the Font class provides the list of fonts required for the **%%DocumentFonts** comment. (See *Document Structuring Conventions* by Adobe Systems Inc.)

The **useFont:** method augments this system by providing a way to register fonts that are included in the document but not set using Font's **set** method. Send a **useFont:** message to the class object for each font of this type. Returns **self**.

See also: – **set**

INSTANCE METHODS

awake

– **awake**

Reinitializes the Font object after it's been read in from a stream. This method makes sure that the Font object doesn't assume it has data cached in the Window Server.

An **awake** message is automatically sent to each object of an application after all objects of that application have been read in. You never send **awake** messages directly. The **awake** message gives the object a chance to complete any initialization that **read:** couldn't do. If you override this method in a subclass, the subclass should send this message to its superclass:

```
[super awake];
```

Returns **self**.

See also: – **read:**, – **write:**, – **finishUnarchiving**

finishUnarchiving

– **finishUnarchiving**

A **finishUnarchiving** message is sent after the Font object has been read in from a stream. This method checks if a Font object for the particular PostScript font already exists. If so, **self** is freed and the existing object is returned.

See also: – **read:**, – **write:**, – **awake**

fontNum

– (int)**fontNum**

Returns the PostScript user object that corresponds to this font. The Font object must set the font in the Window Server before this method will return a valid user object. Sending a Font object the **set** message sets the font in the Window Server. The **fontNum** method returns 0 if the Font object hasn't previously received a **set** message or if the font couldn't be set.

See also: – **set**, **DPSDefineUserObject()**

free

– **free**

Has no effect. Since only one Font object is allocated for a particular font, and since you can't be sure that you have the only reference to a particular Font object, a Font object shouldn't be freed.

getWidthOf:

– (float)**getWidthOf:**(const char *)*string*

Returns the width of *string* using this font. This method has better performance than the Window Server routine **PSstringwidth()**.

hasMatrix

– (BOOL)**hasMatrix**

Returns YES if the Font object's matrix is different from the identity matrix, NX_IDENTITYMATRIX; otherwise, returns NO.

See also: + **newFont:size:style:matrix:**, – **matrix**

matrix

– (const float *)**matrix**

Returns a pointer to the matrix for this font.

See also: – **hasMatrix**

metrics

– (NXFontMetrics *)**metrics**

Returns a pointer to the NXFontMetrics record for the font. See the header file **appkit/afm.h** for the structure of an NXFontMetrics record.

See also: – **readMetrics:**

name

– (const char *)**name**

Returns the name of the font.

pointSize

– (float)**pointSize**

Returns the size of the font in points.

read:

– **read:**(NXTypedStream *)*stream*

Reads the Font object's instance variables from *stream*. A **read:** message is sent in response to archiving; you never send this message.

See also: – **write:**, – **read:** (Object)

readMetrics:

– (NXFontMetrics *)**readMetrics:**(int)*flags*

Returns a pointer to the NXFontMetrics record for this font. The *flags* argument determines which fields of the record will be filled in. *flags* is built by ORing together constants such as NX_FONTHEADER, NX_FONTMETRICS, and NX_FONTWIDTHS. See the header file **appkit/afm.h** for the complete list of constants and for the structure of the NXFontMetrics record.

See also: – **metrics**

screenFont

– **screenFont**

Provides the screen font corresponding to this font. If the receiver represents a printer font, this method returns the Font object for the associated screen font (or **nil** if one doesn't exist). If the receiver represents a screen font, it simply returns **self**.

set

– **set**

Makes this font the current font in the current graphics state. Returns **self**.

When a Font object receives a **set** message, it registers with the Font class object that its PostScript font has been used. In this way, the Application Kit, when called upon to generate a conforming PostScript language document file, can list the fonts used within a document. (See *Document Structuring Conventions* by Adobe Systems Inc.) If the application uses fonts without sending set messages (say through including an EPS file), such fonts must be registered by sending the class object a **useFont:** message.

See also: + **useFont:**

setStyle:

– **setStyle:**(int)*aStyle*

Sets the Font's style. Setting a style isn't recommended but is minimally supported—a Font object's style isn't interpreted in any way by the Application Kit. You can use it for your own non-PostScript language font styles (a drop-shadow style, for example).

Be very careful using this method since it causes the Font to stop being shared. You must reassign the pointer to the Font to the return value of **setStyle:**.

```
font = [font setStyle:12];
```

Returns **self**.

See also: – **style**

style

– (int)**style**

Returns the style of the font. For Font objects created by the Application Kit, this method returns 0.

See also: – **setStyle:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the Font object's instance variables to *stream*. A **write:** message is sent in response to archiving; you never send this message directly.

See also: – **read:**, – **write:** (Object)

CONSTANTS AND DEFINED TYPES

```
/* Flipped matrix */
#define NX_IDENTITYMATRIX  ((float *)  0)
#define NX_FLIPPEDMATRIX   ((float *) -1)

/* Space characters */
#define NX_FIGSPACE        ((unsigned short)0x80)

/* Font information */
typedef struct _NXFaceInfo {
    NXFontMetrics *fontMetrics;   /* Information from afm file. */
    int           flags;          /* Which font info is present. */
    struct _fontFlags {           /* Keeps track of font usage for
                                     Conforming PS */
        unsigned int usedInDoc:1; /* Has font been used in document?*/
        unsigned int usedInPage:1;   /* Has font been used in page? */
        unsigned int usedInSheet:1;  /* Has font been used in sheet?
                                        (There can be more than one
                                        page printed on a sheet of
                                        paper.) */
        unsigned int _PADDING:13;
    } fontFlags;
    struct _NXFaceInfo *nextFInfo;   /* Next record in list. */
} NXFaceInfo;
```

FontManager

INHERITS FROM	Object
DECLARED IN	FontManager.h

CLASS DESCRIPTION

The FontManager is the center of activity for font conversion. It accepts messages from font conversion user-interface objects (such as the Font menu or the Font panel) and appropriately converts the current font in the selection by sending a **changeFont:** message up the responder chain. When an object receives a **changeFont:** message, it should query the FontManager (by sending it a **convertFont:** message), asking it to convert the font in whatever way the user has specified. Thus, any object containing a font that can be changed should respond to the **changeFont:** message by sending a **convertFont:** message back to the FontManager for each font in the selection.

To use the FontManager, you simply insert a Font menu into your application's menu. This is most easily done with Interface Builder, but, alternatively, you can send a **getFontMenu:** message to the FontManager and then insert the menu that it returns into the application's main menu. Once the Font menu is installed, your application automatically gains the functionality of both the Font menu and the Font panel.

The FontManager can be used to convert a font or find out the attributes of a font. It can also be overridden to convert fonts in some application-specific manner. The default implementation of font conversion is very conservative: The font isn't converted unless all traits of the font can be maintained across the conversion.

INSTANCE VARIABLES

```
Inherited from Object      Class               isa;

Declared in FontManager    id                  panel;
                           id                  menu;
                           SEL                 action;
                           int                 whatToDo;
                           NXFontTraitMask     traitToChange;
                           id                  selFont;
                           struct _fmFlags {
                               unsigned int        multipleFont:1;
                               unsigned int        disabled:1;
                           }                   fmFlags;
```

panel	The Font panel.
menu	The Font menu.

action	The action to send.
whatToDo	What to do when a **convertFont:** message is received.
traitToChange	The trait to change if **whatToDo** == NX_CHANGETRAIT.
selFont	The font of the current selection.
fmFlags.multipleFont	True if the current selection has multiple fonts.
fmFlags.disabled	True if the Font panel and menu are disabled.

METHOD TYPES

Creating the FontManager	+ new
Converting fonts	– convertFont: – convertWeight:of: – convert:toFamily: – convert:toHaveTrait: – convert:toNotHaveTrait: – findFont:traits:weight:size: – getFamily:traits:weight:size:ofFont:
Setting parameters	– setAction: + setFontPanelFactory: – setSelFont:isMultiple: – setEnabled:
Querying parameters	– action – availableFonts – getFontMenu: – getFontPanel: – isMultiple – selFont – isEnabled
Target and action methods	– modifyFont: – addFontTrait: – removeFontTrait: – modifyFontViaPanel: – orderFrontFontPanel: – sendAction
Archiving the FontManager	– finishUnarchiving

CLASS METHODS

alloc

Disables the inherited **alloc** method to prevent multiple FontManagers from being created. There's only one FontManager object for each application; you access it using the **new** method. Returns an error message.

See also: + **new**

allocFromZone:

Disables the inherited **allocFromZone** method to prevent multiple FontManagers from being created. There's only one FontManager object for each application; you access it using the **new** method. Returns an error message.

See also: + **new**

new

+ **new**

Returns a FontManager object. An application has no more than one FontManager object, so this method either returns the previously created object (if it exists) or creates a new one. This is the designated **new** method for the FontManager class.

setFontPanelFactory:

+ **setFontPanelFactory:***factoryId*

Sets the class object that will be used to create the FontPanel object when the user chooses Font Panel from the Font menu and no Font panel has yet been created. Unless you use this method to specify another class, the FontPanel class will be used.

INSTANCE METHODS

action

– (SEL)**action**

Returns the action that's sent to the first responder when the user selects a new font from the Font panel or from the Font menu.

See also: – **setAction:**

addFontTrait:

– **addFontTrait:***sender*

Causes the FontManager's action message (by default, **changeFont:**) to be sent up the responder chain. When the responder replies with a **convertFont:** message, the font is converted to add the trait specified by *sender.*

Before the action message is sent up the responder chain, the FontManager sets its **traitToChange** variable to the value returned by sending *sender* a **selectedTag** message. The FontManager also sets its **whatToDo** variable to NX_ADDTRAIT. When the **convertFont:** message is received, the FontManager converts the supplied font by sending itself a **convert:toHaveTrait:** message.

See also: – **removeFontTrait:**, – **convertFont:**, – **convert:toHaveTrait:**, – **selectedTag** (Control)

availableFonts

– (char **)**availableFonts**

Returns by reference a NULL-terminated list of NULL-terminated PostScript font names of all the fonts available for use by the Window Server. The returned names are suitable for creating new Fonts using the **newFont:size:** class method of the Font class. The fonts are not in any guaranteed order, but no font name is repeated in the list. It's the sender's responsibility to free the list when finished with it.

See also: + **newFont:size:** (Font)

convert:toFamily:

– **convert:***fontObj* **toFamily:**(const char *)*family*

Returns a Font object whose traits are the same as those of *fontObj* except as specified by *family.* If the conversion can't be made, the method returns *fontObj* itself. This method can be used to convert a font, or it can be overridden to convert fonts in a different manner.

See also: – **convert:toHaveTrait:**, **convertWeight:of:**

convert:toHaveTrait:

– **convert:***fontObj* **toHaveTrait:**(NXFontTraitMask)*traits*

Returns a Font object whose traits are the same as those of *fontObj* except as altered by the addition of the traits specified by *traits*. Of course, conflicting traits (such as NX_CONDENSED and NX_EXPANDED) have the effect of turning each other off. If the conversion can't be made, the method returns *fontObj* itself. This method can be overridden to convert fonts in a different manner.

See also: – **convert:toNotHaveTrait:**, – **convert:toFamily:**, – **convertWeight:of:**

convert:toNotHaveTrait:

– **convert:***fontObj* **toNotHaveTrait:**(NXFontTraitMask)*traits*

Returns a Font object whose traits are the same as those of *fontObj* except as altered by the removal of the traits specified by *traits*. If the conversion can't be made, the method returns *fontObj* itself. This method can be overridden to convert fonts in a different manner.

See also: – **convert:toHaveTrait:**, – **convert:toFamily:**, – **convertWeight:of:**

convertFont:

– **convertFont:***fontObj*

Converts *fontObj* according to the user's selections from the Font panel or menu. Whenever you receive a **changeFont:** message from the FontManager, you should send a **convertFont:** message for each font in the selection.

This method determines what to do to the *fontObj* by checking the **whatToDo** instance variable and applying the appropriate conversion method. Returns the converted font.

convertWeight:of:

– **convertWeight:**(BOOL)*upFlag* **of:***fontObj*

Attempts to increase (if *upFlag* is YES) or decrease (if *upFlag* is NO) the weight of the font specified by *fontObj*. If it can, it returns a new font object with the higher (or lower) weight. If it can't, it returns *fontObj* itself. By default, this method converts the weight only if it can maintain all of the traits of the original *fontObj*. This method can be overridden to convert fonts in a different manner.

See also: – **convert:toHaveTrait:**, – **convert:toNotHaveTrait:**, – **convert:toFamily:**

findFont:traits:weight:size:

– **findFont:**(const char *)*family*
  traits:(NXFontTraitMask)*traits*
  weight:(int)*weight*
  size:(float)*size*

If there's a font on the system with the specified *family*, *traits*, *weight*, and *size*, then it's returned; otherwise, **nil** is returned. If NX_BOLD or NX_UNBOLD is one of the traits, *weight* is ignored.

finishUnarchiving

– **finishUnarchiving**

Finishes the unarchiving task by instantiating the one application-wide instance of the FontManager class if necessary.

getFamily:traits:weight:size:ofFont:

– **getFamily:**(const char **)*family*
  traits:(NXFontTraitMask *)*traits*
  weight:(int *)*weight*
  size:(float*)*size*
  ofFont:*fontObj*

For the given font object *fontObj*, copies the font family, traits, weight, and point size information into the storage referred to by this method's arguments.

getFontMenu:

– **getFontMenu:**(BOOL)*create*

Returns a menu suitable for insertion in an application's menu. The menu contains an item that brings up the Font panel as well as some common accelerators (such as Bold and Italic). If the *create* flag is YES, the menu is created if it doesn't already exist.

See also: – **getFontPanel:**

getFontPanel:

– **getFontPanel:**(BOOL)*create*

Returns the FontPanel that will be used when the user chooses the Font Panel command from the Font menu. If the *create* flag is YES, the FontPanel is created if it doesn't already exist.

Unless you've specified a different class (by sending a **setFontPanelFactory:** message to the FontManager class before creating the FontManager object), an object of the FontPanel class is returned.

See also: – **getFontMenu:**

isEnabled

– (BOOL)**isEnabled**

Reports whether the controls in the Font panel and the commands in the Font menu are enabled or disabled.

See also: – **setEnabled:**

isMultiple

– (BOOL)**isMultiple**

Returns whether the current selection has multiple fonts.

modifyFont:

– **modifyFont:***sender*

Causes the FontManager's action message (by default, **changeFont:**) to be sent up the responder chain. When the responder replies with a **convertFont:** message, the font is converted in a way specified by the **selectedTag** of the *sender* of this message. The Larger, Smaller, Heavier, and Lighter commands in the Font menu invoke this method.

See also: – **addFontTrait:**, – **removeFontTrait:**

modifyFontViaPanel:

– **modifyFontViaPanel:***sender*

Causes the FontManager's action message (by default, **changeFont:**) to be sent up the responder chain. When the receiver replies with a **convertFont:** message, the FontManager sends a **panelConvertFont:** message to the FontPanel to complete the conversion.

This message is almost always sent by a Control in the Font panel itself. Usually, the panel uses the FontManager's convert routines to do the conversion based on the choices the user has made.

See also: – **panelConvertFont:** (FontPanel)

orderFrontFontPanel:

– **orderFrontFontPanel:***sender*

Sends **orderFront:** to the FontPanel. If there's no Font panel yet, a **new** message is sent to the FontPanel class object, or to the object you specified with the FontManager's **setFontPanelFactory:** class method.

removeFontTrait:

– **removeFontTrait:***sender*

Causes the FontManager's action message (by default, **changeFont:**) to be sent up the responder chain. When the responder replies with a **convertFont:** message, the font is converted to remove the trait specified by *sender.*

Before the action message is sent up the responder chain, the FontManager sets its **traitToChange** variable to the value returned by sending *sender* a **selectedTag** message. The FontManager also sets its **whatToDo** variable to NX_REMOVETRAIT. When the **convertFont:** message is received, the FontManager converts the supplied font by sending itself a **convert:toNotHaveTrait:** message.

See also: – **convertFont:**, – **convert:toHaveTrait:**, – **selectedTag** (Control)

selFont

– **selFont**

Returns the last font set with **setSelFont:isMultiple:**.

If you receive a **changeFont:** message from the FontManager and want to find out what font the user has selected from the Font panel, use the following (assuming **theFontManager** is the application's FontManager object):

```
selectedFont = [theFontManager convertFont:[theFontManager selFont]]
```

See also: – **setSelFont:isMultiple:**, – **modifyFont:**

sendAction

– **sendAction**

Sends the FontManager's action message (by default, **changeFont:**) up the responder chain. The sender is always the FontManager object regardless of which user-interface object initiated the sending of the action. The **whatToDo** and possibly **traitToChange** variables should be set appropriately before sending a **sendAction** message.

You rarely, if ever, need to send a **sendAction** message or to override this method. The message is sent by the target/action messages sent by different user-interface objects that allow users to manipulate the font of the current text selection (for example, the Font panel and the Font menu).

See also: – **setAction:**

setAction:

– **setAction:**(SEL)*aSelector*

Sets the action that's sent when the user selects a new font from the Font panel or from the Font menu. The default is **changeFont:**.

See also: – **sendAction**

setEnabled:

– **setEnabled:**(BOOL)*flag*

Sets whether the controls in the Font panel and the commands in the Font menu are enabled or disabled. By default, these controls and commands are enabled. Even when disabled, the Font panel allows the user to preview fonts. However, when the Font panel is disabled, the user can't apply the selected font to text in the application's main window.

You can use this method to disable the user interface to the font selection system when its actions would be inappropriate. For example, you might disable the font selection system when your application has no document window.

See also: – **isEnabled**

setSelFont:isMultiple:

– **setSelFont:***fontObj* **isMultiple:**(BOOL)*flag*

Sets the font that the Font panel is currently manipulating. An object containing a document should send this message every time its selection changes. If the selection contains multiple fonts, *flag* should be YES.

An object shouldn't send this message as part of its handling of a **changeFont:** message, since doing so will cause subsequent **convertFont:** messages to have no effect. This is because if you are converting a font based on what is set in the Font panel and you reset what's in the panel (by sending a **setSelFont:isMultiple:** message), the FontManager can no longer sensibly convert the font since the information necessary to convert it has been lost.

See also: – **selFont**

CONSTANTS AND DEFINED TYPES

```
typedef unsigned int NXFontTraitMask;

/*
 * Font Traits.  This list should be kept small since the more traits
 * that are assigned to a given font, the harder it will be to map it
 * to some other family.  Some traits are mutually exclusive, such as
 * NX_EXPANDED and NX_CONDENSED.
 */
#define NX_ITALIC              0x00000001
#define NX_BOLD                0x00000002
#define NX_UNBOLD              0x00000004
#define NX_NONSTANDARDCHARSET  0x00000008
#define NX_NARROW              0x00000010
#define NX_EXPANDED            0x00000020
#define NX_CONDENSED           0x00000040
#define NX_SMALLCAPS           0x00000080
#define NX_POSTER              0x00000100
#define NX_COMPRESSED          0x00000200

/* whatToDo values */
#define NX_NOFONTCHANGE        0
#define NX_VIAPANEL            1
#define NX_ADDTRAIT            2
#define NX_SIZEUP              3
#define NX_SIZEDOWN            4
#define NX_HEAVIER             5
#define NX_LIGHTER             6
#define NX_REMOVETRAIT         7
```

FontPanel

INHERITS FROM	Panel : Window : Responder : Object
DECLARED IN	FontPanel.h

CLASS DESCRIPTION

The FontPanel is a user-interface object that lets the user preview fonts and change the font of the text that's selected in the application's main window. The actual changes are made through conversion messages sent to the FontManager. There is only one FontPanel object for each application.

In general, you add the facilities of the FontPanel (and of the other components of the font conversion system: the FontManager and the Font menu) to your application through Interface Builder. You do this by dragging a Font menu into one of your application's menus. At runtime, when the user chooses the Font Panel command for the first time, the FontPanel object will be created and hooked into the font conversion system. You can also create (or access) the FontPanel through either of the **new...** methods.

A FontPanel can be customized by adding an additional View object or hierarchy of View objects (see **setAccessoryView:**). If you want the FontManager to instantiate a panel object from some class other than FontPanel, use the FontManager's **setFontPanelFactory:** method.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from Window	NXRect	frame;
	id	contentView;
	id	delegate;
	id	firstResponder;
	id	lastLeftHit;
	id	lastRightHit;
	id	counterpart;
	id	fieldEditor;
	int	winEventMask;
	int	windowNum;
	float	backgroundGray;
	struct _wFlags	wFlags;
	struct _wFlags2	wFlags2;
Inherited from Panel	(none)	

Declared in FontPanel

```
id                  faces;
id                  families;
id                  preview;
id                  current;
id                  size;
id                  sizes;
id                  manager;
id                  selFont;
NXFontMetrics       *selMetrics;
int                 curTag;
id                  accessoryView;
id                  setButton;
id                  separator;
id                  sizeTitle;
char                *lastPreview;
struct _fpFlags {
    unsigned int        multipleFont:1;
    unsigned int        dirty:1;
}                   fpFlags;
```

faces	The Typeface browser.
families	The Family browser.
preview	The preview field.
current	The current font field.
size	The Size field.
sizes	The Size browser.
manager	The FontManager object.
selFont	The font of the current selection.
selMetrics	The metrics of **selFont**.
curTag	The tag of the currently displayed font.
accessoryView	The application-customized area.
currentBox	The box displaying the current font.
setButton	The Set button.
separator	The line separating buttons from upper part of panel.
sizeTitle	The title over the Size field and Size browser.

lastPreview	The last font previewed.
fpFlags.multipleFont	True if selection has multiple fonts.
fpFlags.dirty	True if panel was updated while not visible.

METHOD TYPES

Creating a FontPanel	+ new + newContent:style:backing:buttonMask:defer:
Setting the font	– panelConvertFont: – setPanelFont:isMultiple:
Configuring the FontPanel	– accessoryView – setAccessoryView: – setEnabled: – isEnabled – worksWhenModal
Editing the FontPanel's fields	– textDidGetKeys:isEmpty: – textDidEnd:endChar:
Displaying the FontPanel	– orderWindow:relativeTo:
Resizing the FontPanel	– windowDidResize: – windowWillResize:toSize:

CLASS METHODS

alloc

Disables the inherited **alloc** method to prevent multiple FontPanels from being created. There's only one FontPanel object for each application; you access it through either of the **new...** methods. Returns an error message.

See also: + **new**, + **newContent:style:backing:buttonMask:defer:**

allocFromZone:

Disables the inherited **allocFromZone** method to prevent multiple FontPanels from being created. There's only one FontPanel object for each application; you access it through either of the **new...** methods. Returns an error message.

See also: + **new**, + **newContent:style:backing:buttonMask:defer:**

new

+ **new**

Returns a FontPanel object by invoking the **newContent:style:backing:buttonMask:defer:** method. An application has no more than one Font panel, so this method either returns the previously created object (if it exists) or creates a new one.

See also: + **new**

newContent:style:backing:buttonMask:defer:

+ **newContent:**(const NXRect *)*contentRect*
  style:(int)*aStyle*
  backing:(int)*bufferingType*
  buttonMask:(int)*mask*
  defer:(BOOL)*flag*

Returns a FontPanel object. An application has no more than one Font panel, so this method either returns the previously created object (if it exists) or creates a new one. The arguments are ignored. This is the designated **new...** method of the FontPanel class.

See also: + **new**

INSTANCE METHODS

accessoryView

– **accessoryView**

Returns the application-customized View set by **setAccessoryView:**.

See also: – **setAccessoryView:**

isEnabled

– (BOOL)**isEnabled**

Reports whether the Font panel's Set button is enabled.

See also: – **setEnabled:**

orderWindow:relativeTo:

– **orderWindow:**(int)*place* **relativeTo:**(int)*otherWin*

Repositions the panel in the screen list and updates the panel if it was changed while not visible. *place* can be one of:

NX_ABOVE
NX_BELOW
NX_OUT

If it's NX_OUT, the panel is removed from the screen list and *otherWin* is ignored. If it's NX_ABOVE or NX_BELOW, *otherWin* is the window number of the window that the Font Panel is to be placed above or below. If *otherWin* is 0, the panel will be placed above or below all other windows.

See also: – **orderWindow:relativeTo:** (Window), – **makeKeyAndOrderFront:** (Window)

panelConvertFont:

– **panelConvertFont:***fontObj*

Returns a Font object whose traits are the same as those of *fontObj* except as specified by the users choices in the Font Panel. If the conversion can't be made, the method returns *fontObj* itself. The FontPanel makes the conversion by using the FontManager's methods that convert fonts. A **panelConvertFont:** message is sent by the FontManager whenever it needs to convert a font as a result of user actions in the Font panel.

setAccessoryView:

– **setAccessoryView:***aView*

Customizes the Font panel by adding *aView* above the action buttons at the bottom of the panel. The FontPanel is automatically resized to accommodate *aView.*

aView should be the top View in a view hierarchy. If *aView* is **nil**, any existing accessory view is removed. If *aView* is the same as the current accessory view, this method does nothing. Returns the previous accessory view or **nil** if no accessory view was previously set.

See also: – **accessoryView**

setEnabled:

– **setEnabled:**(BOOL)*flag*

Sets whether the Font panel's Set button is enabled (the default state). Even when disabled, the Font panel allows the user to preview fonts. However, when the Font panel is disabled, the user can't apply the selected font to text in the application's main window.

You can use this method to disable the user interface to the font selection system when its actions would be inappropriate. For example, you might disable the font selection system when your application has no document window.

See also: – **isEnabled**

setPanelFont:isMultiple:

– **setPanelFont:***fontObj* **isMultiple:**(BOOL)*flag*

Sets the font that the FontPanel is currently manipulating. This message should *only* be sent by the FontManager. Do not send a **setPanelFont:isMultiple:** message directly.

textDidEnd:endChar:

– **textDidEnd:***textObject* **endChar:**(unsigned short)*endChar*

A **textDidEnd:endChar:** message is sent to the FontPanel object when editing is completed in the Size field. This method updates the Size browser and the preview field.

See also: – **textDidGetKeys:isEmpty:**, – **textDidEnd:endChar:** (Text)

textDidGetKeys:isEmpty:

– **textDidGetKeys:***textObject* **isEmpty:**(BOOL)*flag*

A **textDidGetKeys:isEmpty:** message is sent to the FontPanel object whenever the Size field is typed in or emptied.

See also: – **textDidEnd:endChar:**, – **textDidGetKeys:isEmpty:** (Text)

windowDidResize:

– **windowDidResize:***sender*

Adjusts the width of the browser columns and the accessory view in response to window resizing.

See also: – **windowDidResize:** (Window)

windowWillResize:toSize:

– **windowWillResize:***sender* **toSize:**(NXSize *)*frameSize*

Keeps the FontPanel from being sized too small to accommodate the browser columns and accessory view.

See also: – **windowWillResize:toSize:** (Window)

worksWhenModal

– (BOOL)**worksWhenModal**

Returns whether the FontPanel will operate while a modal panel is displayed within the application. By default, this method returns YES.

See also: – **worksWhenModal** (Panel)

CONSTANTS AND DEFINED TYPES

```
/* Tags of View objects in the FontPanel */
#define NX_FPPREVIEWFIELD    128
#define NX_FPSIZEFIELD       129
#define NX_FPREVERTBUTTON    130
#define NX_FPPREVIEWBUTTON   131
#define NX_FPSETBUTTON       132
#define NX_FPSIZETITLE       133
#define NX_FPCURRENTFIELD    134
```

Form

INHERITS FROM	Matrix : Control : View : Responder : Object
DECLARED IN	appkit/Form.h

CLASS DESCRIPTION

A Form is a Control that contains titled entries into which a user can type data values. An example:

Name: ....
Address: ....
Telephone: ....

These entries are indexed starting with zero as the topmost entry. A mouse click event in an entry starts text editing in that entry. A mouse click event outside the Form or a RETURN key event while editing an entry causes the action of the entry to be sent to the target of the entry if there is such an action; otherwise the action of the Form is sent to the target of the Form. If the user presses the Tab key, the next entry is selected.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from View	NXRect	frame;
	NXRect	bounds;
	id	superview;
	id	subviews;
	id	window;
	struct __vFlags	vFlags;
Inherited from Control	int	tag;
	id	cell;
	struct _conFlags	conFlags;

Inherited from Matrix	id	cellList;
	id	target;
	SEL	action;
	id	selectedCell;
	int	selectedRow;
	int	selectedCol;
	int	numRows;
	int	numCols;
	NXSize	cellSize;
	NXSize	intercell;
	float	backgroundGray;
	float	cellBackgroundGray;
	id	font;
	id	protoCell;
	id	cellClass;
	id	nextText;
	id	previousText;
	SEL	doubleAction;
	SEL	errorAction;
	id	textDelegate;
	struct _mFlags	mFlags;
Declared in Form	(none)	

METHOD TYPES

Setting the Cell Class	+ setCellClass:
Initializing a Form Object	– initFrame:
Laying Out the Form	– addEntry:
	– addEntry:tag:target:action:
	– insertEntry:at:
	– insertEntry:at:tag:target:action:
	– removeEntryAt:
	– setInterline:
Resizing the Form	– calcSize
	– setEntryWidth:
	– sizeTo::
	– sizeToFit

Setting Form Values	– doubleValueAt: – floatValueAt: – intValueAt: – setDoubleValue:at: – setFloatValue:at: – setIntValue:at: – setStringValue:at: – stringValueAt:
Returning the Index	– findIndexWithTag: – selectedIndex
Modifying Text Attributes	– setFont: – setTextAlignment: – setTextFont: – setTitle:at: – setTitleAlignment: – setTitleFont: – titleAt:
Editing Text	– selectTextAt:
Modifying Graphic Attributes	– setBezeled: – setBordered:
Displaying	– drawCellAt:
Target and Action	– setAction:at: – setTarget:at:
Assigning a Tag	– setTag:at:

CLASS METHODS

setCellClass:

+ **setCellClass:***classId*

This method initializes the subclass of Cell used in the Form. The default is FormCell. Use this when you subclass FormCell to modify the behavior of a **Form:**, by sending this method with the class **id** of your subclass as the argument.

INSTANCE METHODS

addEntry:

– **addEntry:**(const char *)*title*

Adds a new entry with the given *title* at the bottom of the Form. Returns the FormCell used to implement the entry. Does not redraw the Form even if autodisplay is on.

addEntry:tag:target:action:

– **addEntry:**(const char *)*title*
tag:(int)*anInt*
target:*anObject*
action:(SEL)*aSelector*

Adds a new entry with the given *title* at the bottom of the Form. The tag, target, and action of the corresponding entry are set to the given values. Returns the FormCell used to implement the entry. Does not redraw the Form even if autodisplay is on.

calcSize

– **calcSize**

Invoke this method before drawing after you have modified any of the cells in the Form in such a way that the size of the cells or the size of the title part of the cells has changed. Automatically invoked before any drawing is done after a **setTitle:at:**, **setFont:**, **setBezeled:** or some other similar method has been invoked.

See also: – **validateSize:** (Matrix)

doubleValueAt:

– (double)**doubleValueAt:**(int)*index*

Returns the entry at position *index*, converted to a float by the C function **atof()** then cast as a double.

drawCellAt:

– **drawCellAt:**(int)*index*

Displays the entry at the specified *index* in the Form.

findIndexWithTag:

– (int)**findIndexWithTag:**(int)*aTag*

Returns the index which has the corresponding tag, –1 otherwise.

See also: – **findCellWithTag:** (Matrix)

floatValueAt:

– (float)**floatValueAt:**(int)*index*

Returns the entry at position *index*, converted to a float by the C function **atof()**.

initFrame

– **initFrame:**(const NXRect *)*frameRect*

Initializes and returns the receiver, a new instance of Form, with default parameters in the given frame. The default Form has no entries. Newly created entries will have the following default characteristics: Titles will be right justified, text will be left justified with bezeled border, background colors will be white, text color black, fonts will be the system font 12.0, the interline spacing will be 1.0, and the action selectors will be NULL. This method is the designated initializer for Form; override it if you create a subclass of Form that performs its own initialization.

Note that Form doesn't override the Matrix class's designated initializers **initFrame:mode:cellClass:numRows:numCols:** or **initFrame:mode:prototype:numRows:numCols:**. Don't use those methods to initialize a new instance of Form.

insertEntry:at:

– **insertEntry:**(const char *)*title* **at:**(int)*index*

Inserts a new entry with the given *title* at position *index*. Returns the FormCell used to implement the entry. Does not redraw the Form even if autodisplay is on.

insertEntry:at:tag:target:action:

– **insertEntry:**(const char *)*title*
  at:(int)*index*
  tag:(int)*anInt*
  target:*anObject*
  action:(SEL)*aSelector*

Inserts a new entry with the given *title* at position *index*. The tag, target, and action of the corresponding entry are set to the given values. Returns the FormCell used to implement the entry. Does not redraw the Form even if autodisplay is on.

intValueAt:

– (int)**intValueAt:**(int)*index*

Returns the entry at position *index* converted to an integer by the C function **atoi()**.

removeEntryAt:

– **removeEntryAt:**(int)*index*

Removes the entry at the given *index* and disposes of the associated memory. Note that if you use Matrix's **removeRowAt:andFree:** method to remove an entry, the widths of the titles in the entries will not be readjusted, so use this method instead. Does not redraw the Form even if autodisplay is on. Returns **self**.

selectTextAt:

– **selectTextAt:**(int)*index*

Enters text editing on the entry at *index* and selects all of its contents. Do not invoke this function before inserting your Form in a view hierarchy with a window at the root; it will have no effect. Returns the **id** of the Cell located at *index*.

selectedIndex

– (int)**selectedIndex**

Returns the index of the currently selected entry if any, –1 otherwise. The currently selected entry is the one being edited or, if none of the entries is being edited, then it's the last edited entry.

setAction:at:

– **setAction:**(SEL)*aSelector* **at:**(int)*index*

Sets the action of the FormCell associated with the entry at position *index* in the Form to *aSelector*. Returns **self**.

setBezeled:

– **setBezeled:**(BOOL)*flag*

Sets whether to draw a bezeled frame around the text in the Form (YES is the default). Redraws the Form if autodisplay is on. Returns **self**.

setBordered:

– **setBordered:**(BOOL)*flag*

Sets whether to draw a 1-pixel black frame around the text in the Form (rather than the default bezel). Redraws the Form if autodisplay is on. Returns **self**.

setDoubleValue:at:

– **setDoubleValue:**(double)*aDouble* **at:**(int)*index*

Sets the text of the entry at position *index* to *aDouble*. Redraws the entry. Returns **self**.

setEntryWidth:

– **setEntryWidth:**(NXCoord)*width*

Sets the width of all the entries (including the title part). You should invoke **sizeToFit** after invoking this method. Returns **self**.

setFloatValue:at:

– **setFloatValue:**(float)*aFloat* **at:**(int)*index*

Sets the text of the entry at position *index* to *aFloat*. Redraws the entry. Returns **self**.

setFont:

– **setFont:***fontObj*

Sets the font used to draw both the titles and the editable text in the Form. It's generally best to keep the title font and the text font the same (or at least the same size); therefore, this method is preferred to **setTitleFont:** and **setTextFont:**. Redraws the Form if autodisplay is on. Returns **self**.

setIntValue:at:

– **setIntValue:**(int)*anInt* **at:**(int)*index*

Sets the text of the entry at position *index* to *anInt*. Returns **self**.

setInterline:

– **setInterline:**(NXCoord)*spacing*

Changes the value of the interline spacing. Does not redraw the matrix even if autodisplay is on. Returns **self**.

setStringValue:at:

– **setStringValue:**(const char *)*aString* **at:**(int)*index*

Sets the text of the entry at position *index* to a copy of *aString*. The entry is redrawn. Returns **self**.

setTag:at:

– **setTag:**(int)*anInt* **at:**(int)*index*

Sets the tag of the FormCell associated with the entry at position *index* in the Form to *anInt*. Returns **self**.

setTarget:at:

– **setTarget:***anObject* **at:**(int)*index*

Sets the target of the FormCell associated with the entry at position *index* in the Form to *anObject*.

setTextAlignment:

– **setTextAlignment:**(int)*mode*

Sets the justification mode for the editable text in the Form. *mode* can be one of three constants: NX_LEFTALIGNED, NX_CENTERED or NX_RIGHTALIGNED. Redraws the Form if autodisplay is on, and returns **self**.

setTextFont:

– **setTextFont:***fontObj*

Sets the font used to draw the editable text in the Form. Redraws the Form if autodisplay is on, and returns **self**.

See also: – **setFont:**

setTitle:at:

– **setTitle:**(const char *)*aString* **at:**(int)*index*

Changes the title of the entry at position *index* to *aString*.

setTitleAlignment:

– **setTitleAlignment:**(int)*mode*

Sets the justification mode for titles in the Form. *mode* can be one of three constants: NX_LEFTALIGNED, NX_CENTERED or NX_RIGHTALIGNED. Redraws the Form if autodisplay is on, and returns **self**.

setTitleFont:

– **setTitleFont:***fontObj*

Sets the font used to draw the titles in the Form. Redraws the Form if autodisplay is on and returns **self**.

See also: – **setFont:**

sizeTo::

– **sizeTo:**(NXCoord)*width* **:**(NXCoord)*height*

Resizes the entry width to reflect *width*, then resizes the Form to *width* and *height*. Returns **self**.

sizeToFit

– **sizeToFit**

Adjusts the width of the Form so that it s the same as the width of the entries. Adjusts the height of the Form so that it will just contain all of the cells. Returns **self**.

See also: – **setEntryWidth:**

stringValueAt:

– (const char *)**stringValueAt:**(int)*index*

Returns a pointer to the text (contents) of the entry at position *index*.

titleAt:

– (const char *)**titleAt:**(int)*index*

Returns a pointer to the title of the entry at position *index*.

FormCell

INHERITS FROM	ActionCell : Cell : Object
DECLARED IN	appkit/FormCell.h

CLASS DESCRIPTION

This class is used to implement the details of the Form class. Form is a subclass of Matrix, and this is the cell which goes in that Matrix. The **titleCell** is used to draw the title of the FormCell. The **titleWidth** is the width of the title (in pixels). If it's –1.0, then the title is autosized to the width of the **titleCell**. The **titleEndPoint** is the coordinate at which the title ends and the editable text begins.

If you want to change the look of a Form, then you should subclass FormCell. When you do so, remember to implement both **drawSelf:inView:** *and* **drawInside:inView:**. The **initTextCell:** method is the designated initializer for FormCell; override this method if your subclass performs its own initialization.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Cell	char	*contents;
	id	support;
	struct _cFlags1	cFlags1;
	struct _cFlags2	cFlags2;
Inherited from ActionCell	int	tag;
	id	target;
	SEL	action;
Declared in FormCell	NXCoord	titleWidth;
	id	titleCell;
	NXCoord	titleEndPoint;

titleWidth	The width of the title field.
titleCell	The cell used to draw the title.
titleEndPoint	The coordinate that separates the title from the text area.

METHOD TYPES

Copying, Initializing, and Freeing a FormCell	
	– copy
	– init
	– initTextCell:
	– free
Determining the FormCell's Size	– calcCellSize:inRect:
Enabling and Disabling the FormCell	
	– setEnabled:
Modifying the Title	– setTitle:
	– setTitleAlignment:
	– setTitleFont:
	– setTitleWidth:
	– title
	– titleAlignment
	– titleFont
	– titleWidth
	– titleWidth:
Modifying Graphic Attributes	– isOpaque
Displaying	– drawInside:inView:
	– drawSelf:inView:
Managing the Cursor	– resetCursorRect:inView:
Tracking the Mouse	– trackMouse:inRect:ofView:
Archiving	– read:
	– write:

INSTANCE METHODS

calcCellSize:inRect:

– **calcCellSize:**(NXSize *)*theSize* **inRect:**(const NXRect *)*aRect*

Calculates the size of the FormCell assuming it's constrained to fit within *aRect*. Returns the size in *theSize*.

copy

– **copy**

Creates and returns a copy of the receiving FormCell instance.

drawInside:inView:

– **drawInside:**(const NXRect *)*cellFrame* **inView:***controlView*

Draws only the text inside the FormCell (not the bezels or the title of the FormCell). This is called from the Control method **drawCellInside:** (which is called from Cell **set***Type***Value:** methods). If you subclass FormCell and override **drawSelf:inView:** you MUST implement this method as well. Returns **self**.

drawSelf:inView:

– **drawSelf:**(const NXRect *)*cellFrame* **inView:***controlView*

Draws the FormCell by sending **drawSelf:inView:** to the **titleCell** with the frame width set to the **titleWidth** (if the **titleWidth** is –1.0, then the width is calculated), and then sending **drawSelf:inView:** to super. Does *not* invoke [super **drawSelf:inView:**] nor does it invoke [self **drawInside:inView:**] (it does, however, invoke [super **drawInside:inView:**]). Returns **self**.

free

– **free**

Frees the storage used by the FormCell (the **titleCell**) and returns **nil**.

init

– **init**

Initializes and returns the receiver, a new instance of FormCell, with its contents set to the empty string ("") and its title set to "Field."

initTextCell:

– **initTextCell:**(const char *)*aString*

Initializes and returns the receiver, a new instance of FormCell, with its contents set to the empty string ("") and its title set to *aString*. This method is the designated initializer for FormCell.

isOpaque

– (BOOL)**isOpaque**

Returns YES if the FormCell is opaque, NO otherwise. If the FormCell has a title, then it's NOT opaque (since the title field is not opaque).

read:

– **read:**(NXTypedStream *)*stream*

Reads the FormCell from the typed stream *stream.*

resetCursorRect:inView:

– **resetCursorRect:**(const NXRect *)*cellFrame* **inView:***controlView*

Sets up an appropriate cursor rectangle in *controlView.*

setEnabled:

– **setEnabled:**(BOOL)*flag*

Enables or disables the FormCell.

setTitle:

– **setTitle:**(const char *)*aString*

Sets the title of the FormCell.

setTitleAlignment:

– **setTitleAlignment:**(int)*mode*

Sets the alignment of the title. *mode* can be one of three constants: NX_LEFTALIGNED, NX_CENTERED, or NX_RIGHTALIGNED.

setTitleFont:

– **setTitleFont:***fontObj*

Sets the font used to draw the title of the FormCell.

setTitleWidth:

– **setTitleWidth:**(NXCoord)*width*

Sets the width of the title field. Can be "unset"by providing –1.0 as the *width.*

title

– (const char *)**title**

Returns the title of the FormCell.

titleAlignment

– (int)**titleAlignment**

Returns the alignment of the title. The return value will match one of three constants: NX_LEFTALIGNED, NX_CENTERED, or NX_RIGHTALIGNED.

titleFont

– **titleFont**

Returns the font used to draw the title of the FormCell.

titleWidth

– (NXCoord)**titleWidth**

If the width of the title has already been set (i.e., it's not –1.0), then that value is returned. Otherwise, it's calculated (not constrained to any rectangle) and returned.

titleWidth:

– (NXCoord)**titleWidth:**(const NXSize *)*aSize*

If the title width is already set (i.e., it's not –1.0), then it's returned. Otherwise, the width is calculated constrained to *aSize*.

trackMouse:inRect:ofView:

– (BOOL)**trackMouse:**(NXEvent*)*event*
inRect:(const NXRect*)*aRect*
ofView:*controlView*

Does nothing since clicking in a FormCell causes editing to occur.

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving FormCell to the typed stream *stream* and returns **self**.

Listener

INHERITS FROM	Object
DECLARED IN	appkit/Listener.h

CLASS DESCRIPTION

The Listener class, with the Speaker class, supports communication between applications (tasks) through Mach messaging. Mach messages are the standard way of performing remote procedure calls (RPCs) in the Mach operating system. The Listener class implements the receiving end of a remote message, and the Speaker class implements the sending end.

Remote messages are sent to ports, which act something like mailboxes for the tasks that have the right to receive the messages delivered there. Each Listener corresponds to a single Mach port to which its application has receive rights. Since a port has a fixed size—usually there's room for only five messages in the port queue—when the port is full, a new message must wait for the Listener to take an old message from the queue.

To initiate a remote message, you send an Objective-C message to a Speaker instance. The Speaker method that responds to the message translates it into the proper Mach message protocol and dispatches it to the port of the receiving task. The Mach message is received by the Listener instance associated with the port. The Listener verifies that it understands the message, that the Speaker has sent the correct parameters for the message, and that all data values are well formed—for example, that character strings are null-terminated. The Listener translates the Mach message back into an Objective-C message, which it sends to itself. It's as if an Objective-C message sent to a Speaker in one task is received by a Listener in another task.

Delegation

The Listener methods that receive remote Objective-C messages simply pass those messages on to a delegate. The Listener's job is just to get the message and find another object to respond to it.

The **setDelegate:** method assigns a delegate to the Listener. There's no default delegate, but before the Application object gets its first event, it registers a Listener for the application and makes itself the Listener's delegate. You can register your own Listener (with Application's **setAppListener:** method) in start-up code, but when you send the Application object a **run** message, it will become the Listener's delegate.

If an object has its own delegate when it becomes the Listener's delegate, the Listener looks first to its delegate's delegate and only then to its own delegate when searching for an object to entrust with a remote message. This means that you can implement the methods that respond to remote messages in either the Application object's delegate or in the Application object. (You can also implement the methods directly in a Listener subclass, or in another object you make the Listener's delegate.)

Setting Up a Listener

Two methods, **checkInAs:** and **usePrivatePort**, allocate a port for the Listener:

- With the **checkInAs:** method, the Listener's port is given a name (usually the name of the application) and is registered with the network name server. This makes the port publicly available so that other applications can find it. Applications get send rights to a public port through the **NXPortFromName()** function.
- Alternatively, the Listener's port can be kept private (with the **usePrivatePort** method). Send rights to the port can then be doled out only to selected applications.

Once allocated, the port must be added (with the **addPort** method) to the list of those that the client library monitors. A procedure will automatically be called to read Mach messages from the port queue and begin the Listener's process of transforming the Mach message back into an Objective-C message. The procedure is called between events, provided the priority of getting remote messages is at least as high as the priority of getting the next event.

A Listener is typically set up as follows:

```
myListener = [[Listener alloc] init];
[myListener setDelegate:someOtherObject];
            /*
             * Sets the object responsible for handling
             * messages received.
             */
[myListener checkInAs:"portname"];
            /* or [myListener usePrivatePort] */
[myListener addPort];
            /*
             * Now, between events, the client library
             * will check to see if a message has arrived
             * in the port queue.
             */
. . .
[myListener free];
            /* When we no longer need the Listener. */
```

An application may have more than one Listener and Speaker, but it must have at least one of each to communicate with the Workspace Manager and other applications. If your application doesn't create them, a default Listener and Speaker are created for you at start-up before Application's **run** method gets the first event.

If a Listener is created for you, it will be checked in automatically under the name returned by Application's **appListenerPortName** method. Normally, this is the name assigned to the application at compile time. The port will also be added to the list of those the client library monitors, so the Listener will be scheduled to receive messages asynchronously.

Remote Methods

The Listener and Speaker classes implement a number of methods that can be used to send and receive remote messages. You can add other methods in Listener and Speaker subclasses. The **msgwrap** program can be used to generate subclass definitions from a list of method declarations. Most programmers will use **msgwrap** instead of manually subclassing the Listener class. See the man page for **msgwrap** for details.

The Listener class declares the same set of remote methods as the Speaker class. However, applications will use some of these methods only in their Speaker versions to send messages and others only in their Listener versions to receive messages. For example, **launchProgram:ok:** messages are normally sent by applications to the Workspace Manager, which has the responsibility for launching applications, so in general only the Speaker version of the method will be used. On the other hand, **unmounting:ok:** messages are received by applications when the Workspace Manager is ready to unmount an optical disk. Since the Workspace Manager is in charge of mounting and unmounting disks, applications won't send this message but will use the Listener version of the method to receive it.

Some remote methods, especially those with the prefix "msg", are designed to allow an application to run under program control rather than user control. By implementing these methods, you'll permit a controlling application to run your application in conjunction with others as part of a script.

Argument Types

Remote messages take two kinds of arguments—input arguments, which pass values from the Speaker to the Listener, and output arguments, which are used to pass values back from the Listener to the Speaker. The Listener sends return information back to the Speaker in a separate Mach message to a port provided by the Speaker. The Speaker reformats this information so that it's returned by reference in variables specified in the original Objective-C message.

A method can take up to NX_MAXMSGPARAMS arguments. Arguments are constrained to a limited set of permissible types. Internally, the Listener and Speaker identify each permitted type with a unique character code. Input argument types and their identifying codes are listed below. Note that an array of bytes counts as a single argument, even though two Objective-C parameters are used to refer to it—a pointer to the array and an integer that counts the number of bytes in the array. A character string must be null-terminated.

Category	**Type**	**Character Code**
integer	(int)	i
double	(double)	d
character string	(char *)	c
byte array	(char *), (int)	b
receive rights (port)	(port_t)	r
send rights (port)	(port_t)	s

There's a matching output argument for each of these categories. Since output arguments return information by reference, they're declared as pointers to the respective input types:

Category	**Type**	**Character Code**
integer	(int *)	I
double	(double *)	D
character string	(char **)	C
byte array	(char **), (int *)	B
receive rights (port)	(port_t *)	R
send rights (port)	(port_t *)	S

The validity of all input parameters is guaranteed for the duration of the remote message. The memory allocated for a character string or a byte array is freed automatically after the Listener method returns. If you want to save a string or an array, you must copy it. When the amount of input data is large, you can use the **NXCopyInputData()** function to take advantage of the out-of-line data feature of Mach messaging. This function is passed the index of the argument to be copied (the combination of a pointer and an integer for a byte array counts as a single argument) and returns a pointer to an area obtained through the **vm_allocate()** function. This pointer must be freed with **vm_deallocate()**, rather than **free()**. Note that the size of the area allocated is rounded up to the next page boundary, and so will be at least one page. Consequently, it is more efficient to **malloc()** and copy amounts up to about half the page size.

The application is responsible for deallocating all port parameters received with the **port_deallocate()** function when they're no longer needed.

Return Values

All remote methods return an **int** that indicates whether or not the message was successfully transmitted. A return of 0 indicates success.

The Listener methods that receive remote messages use the return value to signal whether they're able to delegate a message to another object. If a method can't entrust its message to the delegate (or the delegate's delegate), it returns a value other than 0. If, on the other hand, it's successful in delegating the message, it passes on the delegate's return value as its own. In general, delegate methods should always return 0.

The Listener doesn't pass the return value back to the Speaker that initiated the remote message. However, if the Speaker is expecting return information from the Listener—that is, if the remote message has output arguments—a nonzero return causes the Listener to send an immediate message back to the Speaker indicating its failure to find a delegate for the remote message. The Speaker method then returns –1.

Note that the return value indicates only whether the message got through; it doesn't say anything about whether the action requested by the message was successfully carried out. To provide that information, a remote message must include an output argument.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in Listener	char	*portName;
	port_t	listenPort;
	port_t	signaturePort;
	id	delegate;
	int	timeout;
	int	priority;

portName	The name under which the port is registered.
listenPort	The port where the Listener receives remote messages.
signaturePort	The port used to authenticate registration.
delegate	The object responsible for responding to remote messages received by the Listener.
timeout	How long, in milliseconds, that the Listener will wait for its return results to be placed in the port queue of the sending application.
priority	The priority level at which the Listener will receive messages.

METHOD TYPES

Initializing the class	+ initialize
Initializing a new Listener instance	– init
Freeing a Listener	– free
Setting up a Listener	– addPort
	– removePort
	– checkInAs:
	– usePrivatePort
	– checkOut
	– listenPort
	– signaturePort
	– portName
	– setPriority:
	– priority
	– setTimeout:
	– timeout
	+ run

Standard remote methods	– openFile:ok:
	– openTempFile:ok:
	– launchProgram:ok:
	– powerOffIn:andSave:
	– extendPowerOffBy:actual:
	– unhide
	– unmounting:ok:
Handing off an icon	– iconEntered:at::iconWindow:iconX:iconY: iconWidth:iconHeight:pathList:
	– iconMovedTo::
	– iconReleasedAt::ok:
	– iconExitedAt::
	– registerWindow:toPort:
	– unregisterWindow:
Providing for program control	– msgCalc:
	– msgCopyAsType:ok:
	– msgCutAsType:ok:
	– msgDirectory:ok:
	– msgFile:ok:
	– msgPaste:
	– msgPosition:posType:ok:
	– msgPrint:ok:
	– msgQuit:
	– msgSelection:length:asType:ok:
	– msgSetPosition:posType:andSelect:ok:
	– msgVersion:ok:
Getting file information	– getFileInfoFor:app:type:ilk:ok:
	– getFileIconFor:TIFF:TIFFLength:ok:
Receiving remote messages	– messageReceived:
	– performRemoteMethod:paramList:
	– remoteMethodFor:
Assigning a delegate	– setDelegate:
	– delegate
	– setServicesDelegate:
	– servicesDelegate
Archiving	– read:
	– write:

CLASS METHODS

initialize

+ initialize

Sets up a table that instances of the class use to recognize the remote messages they understand. The table lists the methods that can receive remote messages and specifies the number of parameters for each and their types. An **initialize** message is sent to the class the first time it's used; you should never invoke this method.

run

+ run

Sets up the necessary conditions for Listener objects to receive remote messages if they're used in applications that don't have an Application object and a main event loop. In other words, if an application doesn't send a **run** message to the Application object,

```
[NXApp run];
```

it will need to send a **run** message to the Listener class

```
[Listener run];
```

for instances of the class to work. This method never returns, so your application will probably need to be dispatched by messages to its Listener instances.

INSTANCE METHODS

addPort

– addPort

Enables the Listener to receive messages by adding its port to the list of those that the client library monitors. The Listener will then be scheduled to receive messages between events. Returns **self**.

See also: **– removePort**, **DPSAddPort()**

checkInAs:

– (int)**checkInAs:**(const char *)*name*

Allocates a port for the Listener, and registers that port as *name* with the Mach network name server. This method also allocates a signature port that's used to protect the right to remove *name* from the name server. This method returns 0 if it successfully checks in the application with the name server, and a Mach error code if it doesn't. The Mach error code is most likely to be one of those defined in the header files **netname_defs.h** and **sys/kern_return.h**

See also: – **usePrivatePort**, – **checkOut**

checkOut

– (int)**checkOut**

Removes the Listener's port from the list of those registered with the network name server. This makes the port private. This method will always be successful and therefore always returns 0.

See also: – **checkInAs:**

delegate

– **delegate**

Returns the Listener's delegate. The default delegate is **nil**, but just before the first event is received, the Application object is made the delegate of the Listener registered as the Application object's Listener. The delegate is expected to respond to the remote messages received by the Listener, although it may do this by sending messages to its own delegate. Here is an example of how this can work: When the Application object's Listener receives an **openFile:ok:** message, it passes this message to its delegate, which is the Application object. The Application object, in turn, queries its delegate to see if it accepts another file, and if it does, the Application object sends its delegate a **app:openFile:type:** message.

See also: – **setDelegate:**, – **setAppListener:** (Application)

extendPowerOffBy:actual:

– (int)**extendPowerOffBy:**(int)*requestedMs* **actual:**(int *)*actualMs*

Receives a remote message requesting the Workspace Manager for more time before logging out or turning the power off. Other applications use the Speaker version of this method to send the Workspace Manager **extendPowerOffBy:actual:** requests.

See also: – **extendPowerOffBy:actual:** (Speaker), – **powerOffIn:andSave:**, – **app:powerOffIn:andSave:** (Application delegate)

free

– **free**

Frees the Listener object and deallocates its listen port and its signature port. If the Listener's port is registered with the network name server, it is unregistered.

See also: – **allocFromZone:** (Object), – **init**

getFileIconFor:TIFF:TIFFLength:ok:

– (int)**getFileIconFor:**(char *)*fullPath*
  TIFF:(char **)*tiff*
  TIFFLength:(int *)*length*
  ok:(int *)*flag*

Receives a remote message to obtain information about an icon. The Workspace Manager implements a method that responds to this message. For information on how to use **getFileIconFor:TIFF:TIFFlength:ok:** messages to get information from the Workspace Manager, see the Speaker class.

See also: – **getFileIconFor:TIFF:TIFFLength:ok:** (Speaker)

getFileInfoFor:app:type:ilk:ok:

– (int)**getFileInfoFor:**(char *)*fullPath*
  app:(char **)*appName*
  type:(char **)*aType*
  ilk:(int *)*anIlk*
  ok:(int *)*flag*

Receives a remote message to obtain information about a file. The Workspace Manager implements a method that can respond to this message. For information on how to use **getFileInfoFor:app:type:ilk:ok:** messages to get information from the Workspace Manager, see the Speaker class.

See also: – **getFileInfoFor:app:type:ilk:ok:** (Speaker)

iconEntered:at::iconWindow:iconX:iconY:iconWidth:iconHeight:pathList:

– (int)**iconEntered:**(int)*windowNum*
  at:(double)*x*
  :(double)*y*
  iconWindow:(int)*iconWindowNum*
  iconX:(double)*iconX*
  iconY:(double)*iconY*
  iconWidth:(double)*iconWidth*
  iconHeight:(double)*iconHeight*
  pathList:(const char *)*pathList*

Receives a remote message from the Workspace Manager that the user has dragged an icon into the *windowNum* window. This message is received when the icon first enters the window, but only if *windowNum* was previously registered through a **registerWindow:toPort:** message to the Workspace Manager:

```
unsigned int windowNum;
id speaker = [NXApp appSpeaker];

NXConvertWinNumToGlobal([myWindow windowNum], &windowNum);
[speaker setSendPort:NXPortFromName(NX_WORKSPACEREQUEST, NULL)];
[speaker registerWindow:windowNum toPort:[myListener listenPort]];
```

windowNum is the global window number of the window the icon entered. (The global window number is the one assigned by the Window Server, not the user object maintained within an application.)

x and *y* specify the cursor's location in screen coordinates.

iconWindowNum is the global window number of the off-screen window where the icon image is cached. The icon can be composited from that window to your own. The four arguments *iconX*, *iconY*, *iconWidth*, and *iconHeight* locate the rectangle occupied by the icon in *iconWindow*'s base coordinates.

pathList is the null-terminated pathname of the file represented by the icon. If the icon represents a number of files, *pathList* will contain a list of tab-separated paths.

You will probably want to save a copy of the file icon and/or the path list so you can use them in your **iconMovedTo::** and **iconReleasedAt::ok:** methods. The following implementation of this method saves both:

```
char *iconPathList = NULL;
NXSize size = {48.0, 48.0};
myFileIcon = [[NXImage alloc] initSize:&size];

- (int)iconEntered:(int)windowNum at:(double)winX :(double)winY
    iconWindow:(int)iconWindowNum iconX:(double)x iconY:(double)y
    iconWidth:(double)w iconHeight:(double)h
    pathList:(char *)pathList
```

```
{
    /* lock focus on the image so we can use the pswrap function */
    /* to copy the icon from the icon's window */
    [myFileIcon lockFocus];
    copyIconPicture(iconWindowNum, (float)x, (float)y,
        (float)w, (float)h);
    [myFileIcon unlockFocus];

    /* The icon now has a copy of the picture. Let's make */
    /* a copy of the path list */

    if (iconPathList) NX_FREE(iconPathList);
    /* allocate space for the path list and copy the string */
    iconPathList = NXCopyStringBuffer(pathList);

    /* Don't forget to free your copy of the path list in your */
    /* iconReleasedAt::ok: and iconExitedAt:: methods.  You will */
    /* also need to set iconPathList to NULL */
    return 0;
}
```

In order to copy the icon to your image, you'll need a **copyIconPicture()** function. Put the following pswrap in a file with an extension of **.psw**:

```
defineps copyIconPicture(int win; float x; float y; float w; float h)
    x y w h gsave win windowdeviceround gstate grestore 0 0
    Copy composite
endps
```

See also: – **registerWindow:toPort:** (Speaker), – **iconMovedTo::**, – **iconReleasedAt::ok:**, – **dragFile:fromRect:slideBack:event:** (View)

iconExitedAt::

– (int)**iconExitedAt:**(double)*x* **:**(double)*y*

Receives a remote message from the Workspace Manager that the user has dragged an icon out of a registered window. An **iconExitedAt::** message will be received only after the application has been notified that the icon entered the window. The two arguments, *x* and *y*, specify the cursor's location in screen coordinates when the icon exited the window.

See also:
– **iconEntered:at::iconWindow:iconX:iconY:iconWidth:iconHeight:pathList:**

iconMovedTo::

– (int)**iconMovedTo:**(double)*x* **:**(double)*y*

Receives a remote message from the Workspace Manager that the user has dragged an icon to the cursor location (*x*, *y*) in screen coordinates. You will probably want to use Window's **convertScreenToBase:** method to convert these points to window coordinates, and View's **convertPoint:fromView:** method to then convert them to the coordinate system of a particular View. **iconMovedTo::** messages are repeatedly received while the icon is being dragged within a registered window. They're received only after the application has been notified that the icon entered the window and before it has been notified that the icon exited the window.

See also:
– **iconEntered:at::iconWindow:iconX:iconY:iconWidth:iconHeight:pathList:**,
– **convertScreenToBase:** (Window), – **convertPoint:fromView:** (View),
– **iconReleasedAt::ok:**

iconReleasedAt::ok:

– (int)**iconReleasedAt:**(double)*x*
    :(double)*y*
    ok:(int *)*flag*

Receives a remote message from the Workspace Manager when the user releases an icon over a registered window. The Workspace Manager sends an **iconReleasedAt::ok:** message only after notifying the application that the icon entered the window. Your **iconEntered:at:...** method should save the icon's image and pathname if you need them for this method.

The first two arguments, *x* and *y*, specify the location of the cursor in screen coordinates when the user let go of the mouse button to stop dragging the icon.

The **iconReleasedAt::ok:** method you implement should set the integer referred to by *flag* to 1 if you want the Workspace Manager to hide the icon window the user was dragging, or to 0 if you want the Workspace Manager to animate the icon back to its source window (indicating to the user that your window didn't accept it).

See also:
– **iconEntered:at::iconWindow:iconX:iconY:iconWidth:iconHeight:pathList:**,
– **iconMovedTo::**

init

– **init**

Initializes the Listener which must be a newly allocated Listener instance. The new instance has no port name, its priority is set to NX_BASETHRESHOLD, its timeout is initialized to 30,000 milliseconds, its listen port and signature port are both PORT_NULL, and it has no delegate. Returns **self**.

See also: + **alloc** (Object), + **allocFromZone:** (Object), + **new** (Object), – **setPriority:**, – **setTimeout:**, – **setDelegate:**, – **checkInAs:**

launchProgram:ok:

– (int)**launchProgram:**(const char *)*name* **ok:**(int *)*flag*

Receives requests to launch an application. The Workspace Manager is the application that properly responds to these requests. See the Speaker class for information on how to send **launchProgram:ok:** messages to the Workspace Manager.

See also: – **launchProgram:ok:** (Speaker)

listenPort

– (port_t)**listenPort**

Returns the port at which the Listener receives remote messages. This port is never set directly, but is allocated by either **checkInAs:** or **usePrivatePort**. It's deallocated by the **free** method. The Listener caches this port as its **listenPort** instance variable.

See also: – **checkInAs:**, – **usePrivatePort**, – **free**

messageReceived:

– **messageReceived:**(NXMessage *)*msg*

Begins the process of translating a Mach message received at the Listener's port into an Objective-C message. This method verifies that the Mach message is well formed, that it corresponds to an Objective-C method understood by the Listener, and that the method's arguments agree in number and type with the fields of the Mach message.

messageReceived: messages are initiated whenever a Mach message is to be read from the Listener's port; you shouldn't initiate them in the code you write. Returns **self**.

See also: – **performRemoteMethod:paramList:**

msgCalc:

– (int)**msgCalc:**(int *)*flag*

Receives a remote message to perform any calculations that are necessary to bring the current window up to date. The method you implement to respond to this message should set the integer specified by *flag* to YES if the calculations will be performed, and to NO if they won't.

msgCopyAsType:ok:

– (int)**msgCopyAsType:**(const char *)*aType* **ok:**(int *)*flag*

Receives a remote message requesting the application to copy the current selection to the pasteboard as *aType* data. *aType* should be one of the standard pasteboard types defined in **appkit/Pasteboard.h**. The method you implement to respond to this request should set the integer referred to by *flag* to YES if the selection is copied, and to NO if it isn't.

msgCutAsType:ok:

– (int)**msgCutAsType:**(const char *)*aType* **ok:**(int *)*flag*

Receives a remote message requesting the application to delete the current selection and place it in the pasteboard as *aType* data. *aType* should be one of the standard pasteboard types defined in **appkit/Pasteboard.h**. The method you implement to respond to this request should set the integer referred to by *flag* to YES if the requested action is carried out, and to NO if it isn't.

msgDirectory:ok:

– (int)**msgDirectory:**(char *const *)*fullPath* **ok:**(int *)*flag*

Receives a remote message asking for the current directory. The method you implement to respond to this message should place a pointer to the full path of its current directory in the variable specified by *fullPath*. The integer specified by *flag* should be set to YES if the directory will be provided, and to NO if it won't.

The current directory is application-specific, but is probably best described as the directory the application would show in its Open panel were the user to bring it up.

msgFile:ok:

– (int)**msgFile:**(char *const *)*fullPath* **ok:**(int *)*flag*

Receives a remote message requesting the application to provide the full pathname of its current document. The current document is the file displayed in the main window.

The method you implement to respond to this request should set the pointer referred to by *fullPath* so that it points to a string containing the full pathname of the current document. The integer specified by *flag* should be set to YES if the pathname is provided, and to NO if it isn't.

msgPaste:

– (int)**msgPaste:**(int *)*flag*

Receives a remote message requesting the application to replace the current selection with the contents of the pasteboard, just as if the user had chosen the Paste command from the Edit menu. The method you implement to respond to this message should set the integer referred to by *flag* to YES if the request is carried out, and to NO if it isn't.

msgPosition:posType:ok:

– (int)**msgPosition:**(char *const *)*aString*
    posType:(int *)*anInt*
    ok:(int *)*flag*

Receives a remote message requesting a description of the current selection.

The method you implement to respond to this request should describe the selection in a character string and set the pointer referred to by *aString* so that it points the description. The integer referred to by *anInt* should be set to one of the following constants to indicate how the current selection is described:

NX_TEXTPOSTYPE	As a character string to search for
NX_REGEXPRPOSTYPE	As a regular expression to search for
NX_LINENUMPOSTYPE	As a colon-separated range of line numbers, for example "10:12"
NX_CHARNUMPOSTYPE	As a colon-separated range of character positions, for example "21:33"
NX_APPPOSTYPE	As an application-specific description

The integer referred to by *flag* should be set to YES if the requested information is provided in the other two output arguments, and to NO if it isn't.

msgPrint:ok:

– (int)**msgPrint:**(const char *)*fullPath* **ok:**(int *)*flag*

Receives a remote message requesting the application to print the document whose path is *fullPath*. The method you implement to respond to this request should set the integer referred to by *flag* to YES if the document is printed, and to NO if it isn't. The document file should be closed after it's printed.

msgQuit:

– (int)**msgQuit:**(int *)*flag*

Receives a remote message for the application to quit. The method you implement to respond to this message should set the integer specified by *flag* to YES if the application will quit, and to NO if it won't.

msgSelection:length:asType:ok:

– (int)**msgSelection:**(char *const *)*bytes*
  length:(int *)*numBytes*
  asType:(const char *)*aType*
  ok:(int *)*flag*

Receives a remote message asking the application for its current selection as *aType* data. *aType* will be one of the following standard data types for the pasteboard (or an application-specific type):

NXAsciiPboardType
NXPostScriptPboardType
NXTIFFPboardType
NXRTFPboardType
NXSoundPboardType
NXFilenamePboardType
NXTabularTextPboardType

The method you implement to respond to this request should set the pointer referred to by *bytes* so that it points to the selection and also place the number of bytes in the selection in the integer referred to by *numBytes*. The integer referred to by *flag* should be set to YES if the selection is provided, and to NO if it's not.

msgSetPosition:posType:andSelect:ok:

– (int)**msgSetPosition:**(const char *)*aString*
  posType:(int)*anInt*
  andSelect:(int)*selectFlag*
  ok:(int *)*flag*

Receives a remote message requesting the application to scroll the current document (the one displayed in the main window) so that the portion described by *aString* is visible. *aString* should be interpreted according to the *anInt* constant, which will be one of the following:

NX_TEXTPOSTYPE	*aString* is a character string to search for.
NX_REGEXPRPOSTYPE	*aString* is a regular expression to search for.
NX_LINENUMPOSTYPE	*aString* is a colon-separated range of line numbers, for example "10:12".
NX_CHARNUMPOSTYPE	*aString* is a colon-separated range of character positions, for example "21:33".
NX_APPPOSTYPE	*aString* is an application-specific description of a portion of the document.

The **msgSetPosition:posType:andSelect:ok:** method you implement should set the integer referred to by *flag* to YES if the document is scrolled, and to NO if it isn't. If *selectFlag* is anything other than 0, the portion of the document described by *aString* should also be selected.

msgVersion:ok:

– (int)**msgVersion:**(char *const *)*aString* **ok:**(int *)*flag*

Receives a remote message requesting the current version of the application. The method you implement to respond to this request should set the pointer referred to by *aString* so that it points to a string containing current version information for your application. The integer specified by *flag* should be set to YES if version information is provided, and to NO if it's not.

openFile:ok:

– (int)**openFile:**(const char *)*fullPath* **ok:**(int *)*flag*

Receives a remote message asking the application to open a file. The file is identified by an absolute pathname, *fullPath*.

The Application object, NXApp, has an **openFile:ok:** method that can respond to this message. Much of the task of opening and displaying the file is left to the application, however. This can be done by implementing an **appOpenFile:type:** method, either for NXApp's delegate or in an Application subclass, rather than a version of **openFile:ok:**.

If you implement your own version of **openFile:ok:**, it should set the output argument specified by *flag* to YES if the application will open the file, and to NO if it won't. It should return 0 to indicate that the remote message was handled.

See also: – **app:openFile:type:** (Application delegate), – **openFile:ok:** (Application)

openTempFile:ok:

– (int)**openTempFile:**(const char *)*fullPath* **ok:**(int *)*flag*

Receives a remote message asking the application to open a temporary file. The temporary file is identified by an absolute pathname, *fullPath*. The application that receives this message should delete the file when it's no longer needed.

The Application class implements a **openTempFile:ok:** method that can respond to this message.

See also: – **app:openTempFile:type:** (Application delegate),
– **openTempFile:ok:** (Application), – **openFile:ok:** (Application)

performRemoteMethod:paramList:

– (int)**performRemoteMethod:**(NXRemoteMethod *)*method*
paramList:(NXParamValue *)*params*

Matches the data received in the Mach message with the corresponding Objective-C method and initiates the Objective-C message to **self**. The Listener method that receives the message will then try to delegate it to another object. *method* is a pointer to the method structure returned by **remoteMethodFor:** and *params* is a pointer to the list of arguments.

The **msgwrap** program automatically generates a **performRemoteMethod:paramList:** method for a Listener subclass. Each Listener subclass must define its own version of the method.

performRemoteMethod:paramList: messages are initiated when the Listener reads a Mach message from its port queue.

See also: **msgwrap** (in the Unix manual)

portName

– (const char *)**portName**

Returns the name under which the Listener's port (the port returned by the **listenPort** method) is registered with the network name server.

See also: – **checkInAs:**, – **listenPort**, – **appListenerPortName** (Application)

powerOffIn:andSave:

– (int)**powerOffIn:**(int)*ms* **andSave:**(int)*aFlag*

Receives a remote message from the Workspace Manager that the machine will be powered down, or the user will be logged out, in *ms* milliseconds. The second argument, *aFlag*, should be ignored. If *ms* is insufficient time, the application can ask for additional time by sending an **extendPowerOffBy:actual:** to the Workspace Manager.

The Application class implements a **powerOffIn:andSave:** method that can respond to this message. It raises an exception that's caught by the main event loop, which then notifies the Application object's delegate with an **appPowerOffIn:andSave:** message.

See also: – **app:powerOffIn:andSave:** (Application delegate),
– **powerOffIn:andSave:** (Application)

priority

– (int)**priority**

Returns the priority level for receiving remote messages. This value is cached as the Listener's **priority** instance variable.

See also: – **setPriority:**

read:

– **read:**(NXTypedStream *)*stream*

Reads the Listener from the typed stream *stream.*

See also: – **write:**

registerWindow:toPort:

– (int)**registerWindow:**(int)*windowNum* **toPort:**(port_t)*aPort*

Receives a remote message registering *windowNum* to receive icons the user drags into the window. The Workspace Manager implements a method that responds to this message. Other applications will use the Speaker version of the method to send the Workspace Manager **registerWindow:toPort:** messages.

See also: – **registerWindow:toPort:** (Speaker), – **iconEntered:at:...**

remoteMethodFor:

– (NXRemoteMethod *)**remoteMethodFor:**(SEL)*aSelector*

Looks up *aSelector* in the table of remote messages the Listener understands and returns a pointer to the table entry. A NULL pointer is returned if *aSelector* isn't in the table.

Each Listener subclass must define its own version of this method and send a message to **super** to perform the Listener version. The **msgwrap** program produces subclass method definitions automatically. The version of the method produced by **msgwrap** uses the **NXRemoteMethodFromSel()** function to do the look up.

remoteMethodFor: messages are initiated automatically when the Listener reads a Mach message from its port queue.

See also: – **performRemoteMethod:paramList:**, **msgwrap** (in the Unix manual)

removePort

– **removePort**

Removes the Listener's port from the list of those that the client library monitors. Remote messages sent to the port will pile up in the port queue until they are explicitly read; they won't be read automatically between events.

See also: – **addPort**

servicesDelegate

– **servicesDelegate**

Returns the Listener's services delegate, the object that will respond to remote messages sent from the Services menus of other applications. The services delegate should contain the methods that a service providing application uses to provide services to other applications.

See also: – **setServicesDelegate:**

setDelegate:

– **setDelegate:***anObject*

Sets the Listener's delegate to *anObject*. The delegate is expected to respond to the remote messages received by the Listener. However, if *anObject* has a delegate of its own at the time the **setDelegate:** message is sent, the Listener will first check to see if that object can handle a remote message before checking *anObject*. In other words, the Listener recognizes a chain of delegation.

The delegate assigned by this method will be overridden if the Listener is registered as the Application object's **appListener** and the assignment is made before the Application object is sent a **run** message. Before getting the first event, the **run** method makes the Application object the **appListener**'s delegate.

See also: – **delegate**, – **setAppListener:** (Application)

setPriority:

– **setPriority:**(int)*level*

Sets the priority for receiving remote messages to *level*. Whenever the application is ready to get another event, the priority level is compared to the threshold at which the application is asking for the next event. For the Listener to be able to receive remote messages from its port queue, the priority level must be at least equal to the event threshold.

Priority values can range from 0 through 30, but three standard values are generally used:

NX_MODALRESPTHRESHOLD	10
NX_RUNMODALTHRESHOLD	5
NX_BASETHRESHOLD	1

These constants are defined in the **appkit/Application.h** header file.

- At a priority equal to NX_BASETHRESHOLD, the Listener will be able to receive messages whenever the application asks for an event in the main event loop, but not during a modal loop associated with an attention panel nor during a modal loop associated with a control such as a button or slider.

- At a priority equal to NX_RUNMODALTHRESHOLD, the Listener will receive remote messages in the main event loop and in the event loop for an attention panel, but not during a control event loop.

- At a priority equal to NX_MODALRESPTHRESHOLD, remote messages are received even during a control event loop.

The default priority level is NX_BASETHRESHOLD.

A new priority takes effect when the Listener receives an **addPort** message. To change the default, you must either set the Listener's priority before sending it an **addPort** message, or you must send it a **removePort** message then another **addPort** message.

See also: – **priority**, – **addPort**

setServicesDelegate:

– **setServicesDelegate:***anObject*

Registers *anObject* as the object within a service provider that will respond to remote messages. This method returns **self**. As an example, consider an application called **Thinker** that provides a ThinkAboutIt service that ponders the meaning of Ascii text it receives on the pasteboard. **Thinker** would need to have something like the following in the __services section of its __ICON segment in its Mach-O file:

```
Message: thinkMethod
Port: Thinker
Send Type: NXAsciiPboardType
Menu Item: ThinkAboutIt
```

To get this information in your Mach-O file you could put the above text in a file called **services.txt** and then include the following line in your **Makefile.preamble** file:

```
LDFLAGS = -segcreate __ICON __services services.txt
```

Alternatively, if the services the application can provide are not known at compile time, the application can build a services file at run time; see **NXUpdateDynamicServices()**.

Then, in order to provide the ThinkAboutIt service you must implement a **thinkMethod:userData:error:** method in an object which is the services delegate of a Listener which is listening on the Thinker port. (If the application is named "Thinker", then by default NXApp's Listener listens on this port.) Here is an example method that could be used to provide the ThinkAboutIt service:

```
- thinkMethod:(id)pb
    userData:(const char *)userData
    error:(char **)msg
{
    char *data;
    int length;
    char *const *s;  /* We use s to go through types. */
    char *const *types = [pb types];

    for (s = types; *s; s++)
        if (!strcmp(*s, NXAsciiPboardType)) break;
    if (*s && [pb readType:NXAsciiPboardType
                    data:&data length:&length])
    {
        /* doSomething is your own method... */
        [self doSomething:data :length];
        /* free the memory allocated by readType:... */
        vm_deallocate(task_self(), data, length);
    }
    /* now make msg point to an error string if */
    /* anything went wrong, and return... */
    return self;
}
```

See also: – **servicesDelegate**,
– **registerServicesMenuSendTypes:andReturnTypes:** (Application),
– **validRequestorForSendType:andReturnType:** (Responder)

setTimeout:

– **setTimeout:**(int)*ms*

Sets, to *ms* milliseconds, how long the Listener will persist in attempting to send a return message back to the Speaker that initiated the remote message. If *ms* is 0, there will be no time limit. The default is 30,000 milliseconds. Returns **self**.

See also: – **timeout**

signaturePort

– (port_t)**signaturePort**

Returns the port that's used to authenticate the Listener's port to the network name server. This port is never set directly, but is allocated by **checkInAs:** and deallocated by **free**. The Listener caches this port as its **signaturePort** instance variable.

See also: – **checkInAs:**, – **free**, **netname_check_in()**, **netname_check_out()**

timeout

– (int)**timeout**

Returns the number of milliseconds the Listener will wait for a return message to the Speaker to be successfully placed in the port designated by the Speaker. This value is cached by the Listener as its **timeout** instance variable. If it's 0, there's no time limit.

See also: – **setTimeout:**

unhide

– (int)**unhide**

Receives a remote message asking the application to unhide its windows and become the active application. When the user double-clicks a freestanding or docked icon for a running application, the Workspace Manager sends the application an **unhide** message. The Application object has an **unhide** method that can respond appropriately to this message. The Application object notifies its delegate with an **appDidUnhide** message, if its delegate can respond. Returns the delegate's return value if the Listener's delegate responds to this message, otherwise returns -1.

See also: – **unhide** (Application)

unmounting:ok:

– (int)**unmounting:**(const char *)*fullPath* **ok:**(int *)*flag*

Receives a remote message from the Workspace Manager that a disk is about to be unmounted. *fullPath* is the full pathname of a directory on the disk that will be unmounted.

The Application class implements an **unmounting:ok:** method that responds to this message. Application's method first tries to assign responsibility for the message to its delegate by sending the delegate an **appUnmounting:** message. Failing that, it tries to change the current working directory so that it's not on the disk.

If you implement your own version of **unmounting:ok:**, it should set the integer specified by *flag* to YES if it's OK for the Workspace Manager to unmount the disk, and to NO if it's not. Most applications will implement **appUnmounting:** instead of **unmounting:ok:**. Returns the delegate's return value if the Listener's delegate responds to this message, otherwise returns -1.

See also: – **unmounting:ok:** (Application)

unregisterWindow:

– (int)**unregisterWindow:**(int)*windowNum*

Receives a remote message to cancel the registration of *windowNum*. The Workspace Manager implements a method that responds to this message. Other applications will send the Workspace Manager **unregisterWindow:** messages when they no longer want to be notified of icons dragged into the window. See the Speaker class for information on sending these messages.

See also: – **unregisterWindow:** (Speaker), – **registerWindow:toPort:** (Speaker), – **iconEntered:at:...**

usePrivatePort

– (int)**usePrivatePort**

Allocates a listening port for the Listener, but doesn't register it publicly. Other tasks can send messages to this Listener only if they are explicitly given the address of the port in a message; the port is not available through the Network Name Server. This method is an alternative to **checkInAs:**. It returns 0 on success and a Mach error code if it can't allocate the port. The error code will be one of those defined in the **kern_return.h** header file in **/usr/include/sys**.

See also: – **checkInAs:**

write:

– write:(NXTypedStream *)*stream*

Writes the Listener to the typed stream *stream.*

See also: **– read:**

CONSTANTS AND DEFINED TYPES

```
/* Port for sending messages to the Workspace Manager */
#define NX_WORKSPACEREQUEST  NXWorkspaceName

/* Port for acknowledging launch by Workspace Manager */
#define NX_WORKSPACEREPLY    NXWorkspaceReplyName

/* Reserved message numbers */
#define NX_SELECTORPMSG      35555
#define NX_SELECTORFMSG      35556
#define NX_RESPONSEMSG       35557
#define NX_ACKNOWLEDGE       35558

/* RPC return result error returns. */
#define NX_INCORRECTMESSAGE -20000

/* Maximum number of remote method parameters allowed */
#define NX_MAXMSGPARAMS      20

#define NX_MAXMESSAGE       (2048-sizeof(msg_header_t)-\
                             sizeof(msg_type_t)-sizeof(int)-\
                             sizeof(msg_type_t)-8)

/* A message sent via Mach */
typedef struct _NXMessage {
    msg_header_t header;            /* every message has one */
    msg_type_t sequenceType;        /* sequence number type */
    int sequence;                   /* sequence number */
    msg_type_t actionType;          /* selector string */
    char action[NX_MAXMESSAGE];
} NXMessage;

/* A message received via Mach */
typedef struct _NXResponse {
    msg_header_t header;            /* every message has one */
    msg_type_t sequenceType;        /* sequence number type */
    int sequence;                   /* sequence number */
} NXResponse;
```

```
/* For acknowledging a message via Mach */
typedef struct _NXAcknowledge {
    msg_header_t header;            /* every message has one */
    msg_type_t sequenceType;        /* sequence number type */
    int sequence;                   /* sequence number */
    msg_type_t errorType;           /* error number type */
    int error;                      /* error number, 0 is ok */
} NXAcknowledge;
/* defines method understood by Listener */
typedef struct _NXRemoteMethod {
    SEL key;                        /* Objective-C selector */
    char *types;                    /* defines types of parameters */
} NXRemoteMethod;

/* used to pass parameters to method */
typedef union {
    int ival;
    double dval;
    port_t pval;
    struct _bval {
        char *p;
        int len;
    } bval;
} NXParamValue;

/*
 * permissible values for the second argument of
 * msgSetPosition:posType:andSelect:ok: and msgPostion:posType:ok:
 */

#define NX_TEXTPOSTYPE     0
#define NX_REGEXPRPOSTYPE  1
#define NX_LINENUMPOSTYPE  2
#define NX_CHARNUMPOSTYPE  3
#define NX_APPPOSTYPE      4
```

Matrix

INHERITS FROM	Control : View : Responder : Object
DECLARED IN	appkit/Matrix.h

CLASS DESCRIPTION

The Matrix class allows creation of matrices of Cells of the same or of different types. The main restriction is that all Cells must have the same size. You can add rows and columns to a Matrix by using **addRow**, **insertRowAt:**, **addCol**, or **insertColAt:**. Cells created by the Matrix to fill its rows and columns will be instances of the Cell subclass stored in the **cellClass** instance variable or copies of the prototype Cell stored in the **protoCell** instance variable.

There are four modes of operation for a Matrix:

NX_TRACKMODE is the most basic mode of operation. All that happens in this mode is that the Cells are asked to track the mouse via **trackMouse:inRect:ofView:** whenever the mouse is inside their bounds. No highlighting is performed. An example of this mode might be a "graphic equalizer" Matrix of Sliders. Moving the mouse around would cause the sliders to move under the mouse.

NX_HIGHLIGHTMODE is a modification of TRACKMODE. In this mode, a Cell is highlighted before it is asked to track the mouse, then unhighlighted when it is done tracking. Useful for multiple unconnected Cells which use highlighting to inform the user that they are being tracked (like buttons).

NX_RADIOMODE is used when you want no more than one Cell to be selected at a time. Used in conjunction with **allowEmptySel:**NO, it can be used to create a set of buttons of which one and only one is selected. Any time a Cell is selected, that Cell's action (if any) is sent to its target (or the Matrix's target if the Cell has none). The canonical example of this mode is a set of radio buttons.

NX_LISTMODE allows multiple Cells to be highlighted. The Cell is not given the opportunity to track the mouse; it is only highlighted. This can be used to select a range of text values, for example. The method **sendAction:to:forAllCells:**NO can be used to iterate through the highlighted Cells and perform various functions on them. Highlighting can be done in many ways including dragging to select, using the shift key to make disjoint selections, and using the alternate key to extend selections.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from View	NXRect	frame;
	NXRect	bounds;
	id	superview;
	id	subviews;
	id	window;
	struct __vFlags	vFlags;
Inherited from Control	int	tag;
	id	cell;
	struct _conFlags	conFlags;
Declared in Matrix	id	cellList;
	id	target;
	SEL	action;
	id	selectedCell;
	int	selectedRow;
	int	selectedCol;
	int	numRows;
	int	numCols;
	NXSize	cellSize;
	NXSize	intercell;
	float	backgroundGray;
	float	cellBackgroundGray;
	id	font;
	id	protoCell;
	id	cellClass;
	id	nextText;
	id	previousText;
	SEL	doubleAction;
	SEL	errorAction;
	id	textDelegate;
	struct _mFlags {	
	unsigned int	highlightMode:1;
	unsigned int	radioMode:1;
	unsigned int	listMode:1;
	unsigned int	allowEmptySel:1;
	unsigned int	autoscroll:1;
	unsigned int	reaction:1;
	}	mFlags;

cellList	The List of Cells.
target	Target of the Matrix.

action	Action of the Matrix.
selectedCell	The currently selected Cell.
selectedRow	The row number of selectedCell.
selectedCol	The column number of selectedCell.
numRows	Number of rows.
numCols	Number of columns.
cellSize	Width & height of the Cells.
intercell	Vertical and horizontal spacing between Cells.
backgroundGray	Background gray.
cellBackgroundGray	Cells background gray.
font	Font of Cells.
protoCell	Prototypical Cell.
cellClass	Factory for new Cells.
nextText	Object to select when Tab key is pressed.
previousText	Object to select when Shift-Tab is pressed.
doubleAction	Action sent on double click.
errorAction	Action to apply for edit errors.
textDelegate	Object to which **textDidEnd:endChar:**, etc. is forwarded.
mFlags.highlightMode	NX_HIGHLIGHTMODE.
mFlags.radioMode	NX_RADIOMODE.
mFlags.listMode	NX_LISTMODE.
mFlags.allowEmptySel	Whether no selection is allowed in NX_RADIOMODE.
mFlags.autoscroll	Autoscroll when in a ScrollView.
mFlags.reaction	**sendAction** caused the Cell to change.

METHOD TYPES

Initializing the Matrix Class Object	+ initialize
	+ setCellClass:
Initializing and Freeing a Matrix	– initFrame:
	– initFrame:mode:cellClass:numRows: numCols:
	– initFrame:mode:prototype:numRows: numCols:
	– free
Creating a new Cell	– makeCellAt::
	– prototype
	– setCellClass:
	– setPrototype:
Laying out the Matrix	– addCol
	– addRow
	– cellCount
	– getCellFrame:at::
	– getCellSize:
	– getIntercell:
	– getNumRows:numCols:
	– getRow:andCol:forPoint:
	– getRow:andCol:ofCell:
	– insertColAt:
	– insertRowAt:
	– removeColAt:andFree:
	– removeRowAt:andFree:
	– renewRows:cols:
	– setCellSize:
	– setIntercell:
Modifying the Matrix	– putCell:at::
	– setMode:
	– setPreviousText:
Modifying the Cells	– sendAction:to:forAllCells:
	– setEnabled:
	– setFont:
	– setIcon:at::
	– setState:at::
	– setTarget:at::
	– setTitle:at::

Editing Text	– selectAll: – selectText: – selectTextAt:: – setNextText: – setTextDelegate: – textDidGetKeys:isEmpty: – textDelegate – textDidChange: – textDidEnd:endChar: – textWillChange: – textWillEnd:
Selecting and Identifying Cells	– allowEmptySel: – cellAt:: – cellList – clearSelectedCell – findCellWithTag: – selectCell: – selectCellAt:: – selectCellWithTag: – selectedCell – selectedCol – selectedRow
Modifying Graphic Attributes	– backgroundColor – backgroundGray – cellBackgroundColor – cellBackgroundGray – font – isBackgroundTransparent – isCellBackgroundTransparent – setBackgroundColor: – setBackgroundGray: – setBackgroundTransparent: – setCellBackgroundColor: – setCellBackgroundGray: – setCellBackgroundTransparent:
Resizing the Matrix and Cells	– doesAutosizeCells – calcSize – setAutosizeCells: – sizeTo:: – sizeToCells – sizeToFit – validateSize:
Scrolling	– scrollCellToVisible:: – setAutoscroll: – setScrollable:

Displaying	– display
	– drawCell:
	– drawCellAt::
	– drawCellInside:
	– drawSelf::
	– highlightCellAt::lit:
Target and Action	– action
	– doubleAction
	– errorAction
	– sendAction
	– sendAction:to:
	– sendDoubleAction
	– setAction:
	– setAction:at::
	– setDoubleAction:
	– setErrorAction:
	– setReaction:
	– setTarget:
	– target
Assigning a Tag	– setTag:at::
	– setTag:target:action:at::
Handling Event and Action Messages	
	– acceptsFirstMouse
	– mouseDown:
	– mouseDownFlags
	– performKeyEquivalent:
Managing the Cursor	– resetCursorRects
Archiving	– read:
	– write:

CLASS METHODS

initialize

+ initialize

Sets the current version of the Matrix class.

setCellClass:

+ **setCellClass:***factoryId*

This method initializes the subclass of Cell used by the Matrix class when the **initFrame:** method is used to initialize a Matrix. You rarely need to invoke this method since you usually set the **cellClass** or a prototype Cell by invoking the methods **initFrame:mode:{prototype,cellClass}:numRows:numCols:** when the Matrix is first initialized.

See also: – **initFrame:**, – **initFrame:mode:cellClass:numRows:numCols:**, – **initFrame:mode:prototype:numRows:numCols:**

INSTANCE METHODS

acceptsFirstMouse

– (BOOL)**acceptsFirstMouse**

Returns NO if the Matrix is in NX_LISTMODE, YES if the Matrix is in any other mode. The Matrix does not accept first mouse in NX_LISTMODE.

action

– (SEL)**action**

Returns the default action of the Matrix. If a Cell which has no action receives an event which causes an action message to be sent to a target object (normally an NX_MOUSEUP event), this action is sent to the Matrix's target.

addCol

– **addCol**

Adds a new column of Cells to the right of the existing columns by invoking **insertColAt:**. New Cells are created with **makeCellAt::**. Does not redraw even if autodisplay is on. If the number of rows or columns in the Matrix has been changed via **renewRows:cols:** then **makeCellAt:** is invoked only if a new one is needed (since **renewRows:cols:** doesn't free any Cells). This fact can be used to your advantage since you can grow and shrink a Matrix without repeatedly creating and freeing the Cells.

See also: – **insertColAt:**, – **makeCellAt::**

addRow

– **addRow**

Adds a new row of Cells at the bottom of the existing rows by invoking **insertRowAt:**. New Cells are created with **makeCellAt::**. Does not redraw even if autodisplay is on. If the number of rows or columns in the Matrix has been changed via **renewRows:cols:** then **makeCellAt:** is invoked only if a new one is needed (since **renewRows:cols:** doesn't free any Cells). This fact can be used to your advantage since you can grow and shrink a Matrix without repeatedly creating and freeing the Cells.

See also: – **insertColAt:**, – **makeCellAt::**

allowEmptySel:

– **allowEmptySel:**(BOOL)*flag*

If *flag* is YES, then the Matrix will allow one or zero Cells to be selected. If *flag* is NO, then the Matrix will allow one and only one Cell (not zero Cells) to be selected. This setting has effect only in NX_RADIOMODE.

backgroundColor

– (NXColor)**backgroundColor**

Returns the background color of the matrix.

backgroundGray

– (float)**backgroundGray**

Returns the background gray. A **backgroundGray** of –1.0 implies no background gray; the Matrix is transparent.

calcSize

– **calcSize**

You never invoke this method. It is invoked automatically by the system if it has to recompute some size information about the Cells. It invokes **calcDrawInfo:** on each Cell in the Matrix. Can be overridden to do more if necessary (Form overrides **calcSize**, for example). Returns **self**.

See also: – **calcSize** (Control, Form), – **validateSize:**

cellAt::

– **cellAt:**(int)*row* **:**(int)*col*

Returns the Cell at row *row* and column *col*.

cellBackgroundColor

– (NXColor)**cellBackgroundColor**

Returns the background color used to fill the background of a Cell.

cellBackgroundGray

– (float)**cellBackgroundGray**

Returns the gray value used to fill the background of a Cell before the Cell is drawn. If –1.0, then no fill is done behind the Cell before drawing (the cell is transparent).

cellCount

– (int)**cellCount**

Returns the number of Cells in the Matrix.

cellList

– **cellList**

Returns the List object that tracks the Cells of the Matrix.

See also: the List class

clearSelectedCell

– **clearSelectedCell**

Sets **selectedCell** to be no selection. Does no drawing. Doesn't end the previous text editing if any and doesn't invoke **selectTextAt::**. Will not allow clearing of selected Cell if NX_RADIOMODE and an empty selection is not allowed. Returns whatever Cell used to be the **selectedCell**. You rarely invoke this method since **selectCellAt:**–1 **:**–1 will clear the selected Cell and redraw.

See also: – **allowEmptySel:**

display

– **display**

Invokes **displayFromOpaqueAncestor:::** if the Matrix is not opaque and either there is an interCell spacing or one or more of the Cells is not opaque. Invokes **display:::** otherwise. The Matrix is considered to be opaque if the **backgroundGray** is non-negative (or if it was **setOpaque:** explicitly). If **cellBackgroundGray** is non-negative, then all of the Cells are treated as if they were opaque.

doesAutosizeCells

– (BOOL)**doesAutosizeCells**

Determines whether Cells will automatically resize when the size of the matrix changes.

doubleAction

– (SEL)**doubleAction**

Returns the action that is sent on a double-click on a Cell in the Matrix.

drawCell:

– **drawCell:***aCell*

If *aCell* is in the Matrix, then it is drawn. Does nothing otherwise. Useful for constructs like: [matrix **drawCell:**[[matrix **cellAt:**row **:**col] **setSomething:**args]]].

drawCellAt::

– **drawCellAt:**(int)*row* **:**(int)*col*

Displays the Cell at (*row*, *col*) in the Matrix.

drawCellInside:

– **drawCellInside:***aCell*

If *aCell* is in the Matrix, then its inside is drawn (i.e., **drawInside:inView:** is invoked on the Cell).

drawSelf::

– **drawSelf:**(const NXRect *)*rects* **:**(int)*rectCount*

Displays the Cells in the Matrix which intersect any of the *rects*.

errorAction

– (SEL)**errorAction**

Returns the action that is sent to the target of the Matrix upon text editing errors.

See also: **setErrorAction:**

findCellWithTag:

– **findCellWithTag:**(int)*anInt*

Returns the Cell which has a tag matching *anInt*. If no Cell in the Matrix matches *anInt*, then nil is returned.

See also: – **setTag:** (ActionCell), – **setTag:at::**, – **setTag:target:action::**, – **selectCellWithTag:**

font

– **font**

Returns the font that will be used to display text in any Cells.

free

– **free**

Deallocates the storage for the Matrix and all its Cells and returns **nil**.

getCellFrame:at::

– **getCellFrame:**(NXRect *)*theRect*
  at:(int)*row*
  :(int)*col*

Returns the frame of the Cell at the specified *row* and *col*.

getCellSize:

– **getCellSize:**(NXSize *)*theSize*

Gets the width and the height of the Cells in the Matrix.

getIntercell:

– **getIntercell:**(NXSize *)*theSize*

Gets the vertical and horizontal spacing between Cells.

getNumRows:numCols:

– **getNumRows:**(int *)*rowCount* **numCols:**(int *)*colCount*

Returns, by reference, the number of rows and columns in the Matrix.

getRow:andCol:forPoint:

– **getRow:**(int *)*row*
  andCol:(int *)*col*
  forPoint:(const NXPoint *)*aPoint*

Returns the Cell at *aPoint* in the Matrix. If *aPoint* is outside the bounds of the Matrix or in an intercell spacing, **getRow:andCol:forPoint:** returns **nil**. Fills **row* and **col* with the row and column position of the Cell. *aPoint* must be in the Matrix's coordinate system.

getRow:andCol:ofCell:

– **getRow:**(int *)*row*
  andCol:(int *)*col*
  ofCell:*aCell*

Gets the row and column position of *aCell* in the Matrix. Fills **row* and **col* with the row and column position of the Cell. Returns the Cell (or **nil** if *aCell* is not in the Matrix).

highlightCellAt::lit:

– **highlightCellAt:**(int)*row*
  :(int)*col*
  lit:(BOOL)*flag*

Highlights or unhighlights the Cell at (*row*, *col*) in the Matrix by sending the **highlight:inView:lit:** message to the Cell. The focus must be locked on the Matrix. Returns **self**.

initFrame:

– **initFrame:**(const NXRect *)*frameRect*

Initializes and returns the receiver, a new instance of Matrix, with default parameters in the given frame. The default font is Helvetica 12.0, the default **cellSize** is 100.0-by-17.0, the default intercell is 1.0-by-1.0, the default **backgroundGray** is –1 (transparent), and the default **cellBackgroundGray** is –1. The new Matrix contains no rows or columns. The default mode is NX_RADIOMODE.

initFrame:mode:cellClass:numRows:numCols:

– **initFrame:**(const NXRect *)*frameRect*
    mode:(int)*aMode*
    cellClass:*cellId*
    numRows:(int)*numRows*
    numCols:(int)*numCols*

Initializes and returns the receiver, a new instance of Matrix, with *numRows* rows and *numCols* columns. Sets the Matrix's mode to *aMode*. *aMode* can be one of four constants:

NX_TRACKMODE	Just track the mouse inside the Cells
NX_HIGHLIGHTMODE	Highlight the Cell, then track, then unhighlight
NX_RADIOMODE	Allow no more than one selected Cell
NX_LISTMODE	Allow multiple selected Cells

These constants are described in the "CLASS DESCRIPTION." The new Matrix adds new Cells by sending **alloc** and **init** messages to the Cell subclass represented by *classId* (the value returned when sending a **class** message to Cell or a subclass of Cell).

This method is the designated initializer for Matrices that add Cells by creating instances of a Cell subclass.

initFrame:mode:prototype:numRows:numCols:

– **initFrame:**(const NXRect *)*frameRect*
    mode:(int)*aMode*
    prototype:*aCell*
    numRows:(int)*numRows*
    numCols:(int)*numCols*

Initializes and returns the receiver, a new instance of Matrix, with *numRows* rows and *numCols* columns. Sets the Matrix's mode to *aMode*. *aMode* can be one of the four constants listed in the previous method.

These constants are described in the "CLASS DESCRIPTION." The new Matrix adds new Cells by copying *aCell*, and instance of Cell or a subclass of Cell. If you do not plan to add any more Cells to this Matrix, invoke [[matrix **setPrototype:**nil] **free**] after creating the Matrix.

This method is the designated initializer for Matrices that add Cells by copying an instance of a Cell subclass.

insertColAt:

– **insertColAt:**(int)*col*

Inserts a new column of Cells before column *col*. New Cells are created with **makeCellAt::**. This method doesn't redraw even if autodisplay is on. Most of the time, you'll want to perform **sizeToCells** after performing this method to resize the Matrix View to fit the newly added Cells. Returns **self**.

insertRowAt:

– **insertRowAt:**(int)*row*

Inserts a new row of Cells before row *row*. New Cells are created with **makeCellAt::**. This method doesn't redraw even if autodisplay is on. Most of the time, you'll want to perform **sizeToCells** after performing this method to resize the Matrix View to fit the newly added Cells. Returns **self**.

isBackgroundTransparent

– (BOOL)**isBackgroundTransparent**

Returns YES if the Matrix background is transparent, NO otherwise.

isCellBackgroundTransparent

– (BOOL)**isCellBackgroundTransparent**

Returns YES if Cells in the Matrix are created with transparent backgrounds, NO otherwise.

makeCellAt::

– **makeCellAt:**(int)*row* **:**(int)*col*

If there is a **protoCell**, then it is cloned by sending it a copy message; otherwise, a new Cell is created by sending **new** to the class object referenced by the **cellClass** instance variable. You never invoke this method directly; it's invoked by **addRow** and other methods. It may be overridden if desired.

See also: – **addCol**, – **addRow**, – **insertColAt:**, – **insertRowAt:**

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

You never invoke this method but may override it to implement subclassses of the Matrix class. The response of the Matrix depends on the **mode** set when it was first initialized:

In NX_TRACKMODE, each Cell is given the opportunity to track the mouse while it is in its bounds.

In NX_HIGHLIGHTMODE, each Cell is given the opportunity to track the mouse while it is in its bounds and the currently tracking Cell is highlighted.

In NX_RADIOMODE, each Cell is given the opportunity to track the mouse while it is in its bounds, the currently tracking Cell is highlighted, and no more than one Cell can have a non-zero state at any time.

In NX_LISTMODE, Cells are not given the opportunity to track the mouse, rather, they are merely highlighted as the mouse is dragged over them. Shift-mousedown can be used to extend the selection, Command-mousedown can be used to make disjoint selections.

In any mode, a mousedown in an editable Cell immediately enters text editing mode. Also, a double-click in any Cell sends the **doubleAction** to the target in addition to the regular action.

See also: – **initFrame:mode:cellClass:numRows:numCols:**,
– **initFrame:mode:prototype:numRows:numCols:**

mouseDownFlags

– (int)**mouseDownFlags**

Returns the flags (e.g., NX_SHIFTMASK) that were in effect when the mouse went down to start the current tracking session. Use this method if you want to access these flags, but don't want the overhead of having to add NX_MOUSEDOWNMASK to the **sendActionOn:** mask in every Cell to get them. This method is valid only during tracking; it's not useful if the target of the Matrix initiates another Matrix tracking loop as part of its action method.

performKeyEquivalent:

– (BOOL)**performKeyEquivalent:**(NXEvent *)*theEvent*

Returns YES if a Cell in the matrix responds to the key equivalent in *theEvent*, NO if no Cell responds. If a Cell responds to the key equivalent, it is sent the messages **highlight:inView:lit:**YES, then **incrementState**, and finally **highlight:inView:lit:**NO. You do not send this message; it is sent when the Matrix or one of its superviews is the first responder and the user presses a key.

prototype

– **prototype**

Returns the prototype Cell set by **initFrame:mode:prototype:numRows:numCols:** or **setPrototype:**.

See also: – **initFrame:mode:prototype:numRows:numCols:**, – **setPrototype:**

putCell:at::

– **putCell:***newCell*
  at:(int)*row*
  :(int)*col*

Replaces the Cell at (*row*, *col*) by *newCell*, and returns the old Cell at that position. Draws the new Cell if autodisplay is on.

read:

– **read:**(NXTypedStream *)*stream*

Reads the Matrix from the typed stream *stream*. Returns **self**.

removeColAt:andFree:

– **removeColAt:**(int)*col* **andFree:**(BOOL)*flag*

Removes the column at position *col*. If *flag* is YES then the Cells in that column are freed. Doesn't redraw even if autodisplay is on. You normally need to invoke **sizeToCells** after invoking this method to resize the Matrix to fit the reduced Cell count. Returns **self**.

removeRowAt:andFree:

– **removeRowAt:**(int)*row* **andFree:**(BOOL)*flag*

Removes the row at position *row*. If *flag* is YES then the Cells in that column are freed. Doesn't redraw even if autodisplay is on. You normally need to invoke **sizeToCells** after invoking this method to resize the Matrix to fit the reduced Cell count. Returns **self**.

renewRows:cols:

– **renewRows:**(int)*newRows* **cols:**(int)*newCols*

Changes the number of rows and columns in the Matrix, but uses the same Cells as before (creates new Cells if the new size is larger). Since renewing the number of rows and columns often requires that the size of the Matrix itself change (by sending a **sizeToCells** message, for example), **renewRows:cols:** doesn't automatically display the Matrix even if autodisplay is on. You will normally want to invoke **sizeToCells** to resize your Matrix View after invoking this method. The **selectedCell** is cleared. Returns **self**.

resetCursorRects

– **resetCursorRects**

Cycles through the Cells asking each to **resetCursorRects:inView:**. If one of the Cells has a cursor rectangle to set up, it will send the message **addCursorRect:cursor:** back to the Matrix. Returns **self**.

scrollCellToVisible::

– **scrollCellToVisible:**(int)*row* **:**(int)*col*

If the Matrix is in a scrolling view, then the Matrix will scroll to make the Cell at (*row*, *col*) visible. Returns **self**.

selectAll:

– **selectAll:***sender*

If the mode of the Matrix is not NX_RADIOMODE, then all the Cells in the Matrix are selected. The currently selected Cell is unaffected. Editable Cells are not affected. The Matrix is redisplayed. Returns **self**.

See also: – **selectText:**, – **selectCellAt::**

selectCell:

– **selectCell:***aCell*

If *aCell* is in the Matrix, then the Cell is selected, the Matrix is redrawn, and the selected Cell is returned. Returns **nil** if the Cell is not in the Matrix.

selectCellAt::

– **selectCellAt:**(int)*row* **:**(int)*col*

Sets **selectedCell** to be the Cell at (*row*, *col*), **selectedRow** to be *row* and **selectedCol** to be *col*. Ends any editing going on in the window and invokes **selectTextAt:***row* **:***col* if the Cell at (*row*, *col*). If *row* or *col* is –1, then the current selection is cleared (unless the Matrix is in NX_RADIOMODE and does not allow empty selection). Redraws the affected Cells and returns **self**.

selectCellWithTag:

– **selectCellWithTag:**(int)*theTag*

Finds the Cell in the Matrix with the given tag and selects it. Returns the Matrix **id** or **nil** if no Cell has *theTag*.

selectedCell

– **selectedCell**

Returns the currently selected Cell.

selectedCol

– (int)**selectedCol**

Returns the column number of the currently selected Cell. If Cells in multiple columns are selected, this method returns the number of the last column in which a cell was selected. If no Cells are selected, this method returns –1.

selectedRow

– (int)**selectedRow**

Returns the row number of the currently selected Cell. If Cells in multiple rows are selected, this method returns the number of the last row in which a cell was selected. If no Cells are selected, this method returns –1.

selectText:

– **selectText:***sender*

Selects the text of an editable Cell in the Matrix, if any. If *sender* is **nextText**, the first Cell is selected; otherwise, the last Cell is selected. Don't invoke this method before inserting the receiving Matrix in a window's view hierarchy and drawing it. Returns **self**.

selectTextAt::

– **selectTextAt:**(int)*row* **:**(int)*col*

Select the text of the Cell at (*row*, *col*) in the Matrix if any. Don't invoke this method before inserting the receiving Matrix in a window's view hierarchy and drawing it. Returns **self**.

sendAction

– **sendAction**

If the selected Cell has an action and a target, its action is sent to its target. If the Cell has an action but no target, its action is sent to the Matrix's target. If the Cell doesn't have an action or target, the Matrix's action is sent to its target.

See also: – **action**, – **setAction:**, – **setTarget:**, – **target**

sendAction:to:

– **sendAction:**(SEL)*theAction* **to:***theTarget*

Sends *theAction* to *theTarget* and returns **self**. You don't normally invoke this method. It is invoked by event handling methods such as Cell's **trackMouse:inRect:ofView:** to send an action to a target in response to an event within the Matrix.

sendAction:to:forAllCells:

– **sendAction:**(SEL)*aSelector*
        to:*anObject*
        forAllCells:(BOOL)*flag*

Repeatedly sends the message [*anObject aSelector:aCell*] for each Cell in the matrix. The process begins with *aCell* being the Cell in the first row and column of the Matrix and proceeds row by row. If the *flag* is NO, then only highlighted Cells are sent in the message; this is useful for performing actions when multiple Cells are selected in an NX_LISTMODE Matrix. The method *aSelector* should return YES if it wants to continue looping for remaining cells, NO otherwise.

Note: This method is *not* invoked to send action messages to target objects in response to mouse-down events in the Matrix. Instead, you can invoke it if you want to have multiple Cells in a Matrix interact with an object.

This method returns **self**.

sendDoubleAction

– **sendDoubleAction**

You don't invoke this method; it is sent in response to a double-click event in the Matrix. The method sends an action message to a target object, depending on the actions and targets of the Matrix and the selected Cell. If the selected Cell has an action, then it sends that action to the selected cell's target. Otherwise, if the Matrix has a **doubleAction** message, it sends that message to the Matrix's target. Finally, if the Matrix doesn't have a **doubleAction**, it sends the Matrix's action to its target. Returns **self**.

setAction:

– **setAction:**(SEL)*aSelector*

Sets the default action of the Matrix. If it has an action, a Cell in the Matrix can respond to certain events (usually NX_MOUSEUP events) within its frame by sending its action to its target. If a Cell doesn't have an action, the Matrix can respond to the event by sending its action to its target (not to the Cell's target). This method sets the action sent by the Matrix in such cases. Returns **self**.

setAction:at::

– **setAction:**(SEL)*aSelector*
  at:(int)*row*
  :(int)*col*

Sets the action of the Cell at (*row*, *col*) to *aSelector*. Returns **self**.

setAutoscroll:

– **setAutoscroll:**(BOOL)*flag*

If *flag* is YES and the Matrix is in a scrolling view, it will be autoscrolled whenever a the mouse is dragged outside the Matrix after a mouse-down event within its bounds. Returns **self**.

setAutosizeCells:

– **setAutosizeCells:**(BOOL)*flag*

Sets Cells in the Matrix to automatically resize when the size of the Matrix changes. Returns **self**.

setBackgroundColor:

– **setBackgroundColor:**(NXColor)*Colorvalue*

Sets the background color for the Matrix. This is the color used to fill the space between Cells or the space behind any non-opaque Cells. If autodisplay is on, the entire Matrix is redrawn. Returns **self**.

setBackgroundGray:

– **setBackgroundGray:**(float)*value*

Sets the background gray for the Matrix (a **backgroundGray** of –1.0 means there is no background gray: the Matrix is transparent). This is the gray used to fill the spaces between Cells or the space behind any non-opaque Cell if **cellBackgroundGray** is –1.0. If autodisplay is on, the entire Matrix is redrawn. Returns **self**.

See also: – **backgroundGray**

setBackgroundTransparent:

– **setBackgroundGray:**(BOOL)*flag*

Sets the background of the Matrix to transparent. With the background transparent, the spaces between Cells are transparent, as is the space behind any non-opaque Cell. If autodisplay is on, the entire Matrix is redrawn.

See also: – **isBackgroundTransparent**

setCellBackgroundColor:

– **setCellBackgroundColor:**(NXColor)*value*

Sets the background color for the Cells. If autodisplay is on, the entire Matrix is redrawn.

setCellBackgroundGray:

– **setCellBackgroundGray:**(float)*value*

Sets the background gray for the Cells. If *value* is –1.0, then no background gray is drawn behind the Cells. If autodisplay is on, the entire Matrix is redrawn.

setCellBackgroundTransparent:

– **setCellBackgroundTransparent:**(BOOL)*flag*

Sets the background of the Cells to transparent. If autodisplay is on, the entire Matrix is redrawn.

See also: – **isCellBackgroundTransparent**

setCellClass:

– **setCellClass:***classId*

Sets the **cellClass** instance variable to *classId*, the value returned by sending a **class** message to Cell or a subclass of Cell. This class will be used by **makeCellAt::** to create new Cells if there is no prototype Cell. The default is set with the **setCellClass:** class method.

See also: + **setCellClass:**, – **setPrototype:**

setCellSize:

– **setCellSize:**(const NXSize *)*aSize*

Sets the width and the height of each of the Cells. Does not redraw the Matrix (even if autodisplay is on).

setDoubleAction:

– **setDoubleAction:**(SEL)*aSelector*

Sets *aSelector* as the action to be sent to the Matrix's target (in addition to the regular action) when the user double-clicks on a Cell. If there is no **doubleAction**, then double-clicks are treated as single-clicks. Setting a double action also sets **allowMultiClick:** to YES. Returns **self**.

See also: – **allowMultiClick:**

setEnabled:

– **setEnabled:**(BOOL)*flag*

If *flag* is YES, enables all Cells in the Matrix; if NO, disables all Cells. If autodisplay is on, this redraws the entire Matrix. Returns **self**.

setErrorAction:

– **setErrorAction:**(SEL)*aSelector*

Sets *aSelector* as the action sent to the target of the Matrix when any text editing errors occur. An error can occur when the user types something into a Cell and the value returned when **isEntryAcceptable:** is sent to the Cell is NO. This is a convenient method for enforcing some restrictions on what a user can type into a Cell. However, if you want to impose some restriction such as a range restriction (e.g., a typed number must be within some bounds), it is probably more convenient simply to check the value in your action method and, if it is not acceptable, invoke **selectTextAt::**) to notify the user that the value must be retyped. Returns **self**.

setFont:

– **setFont:***fontObj*

Sets the font of the Matrix to *fontObj*. This will cause all current Cells to have their font changed to *fontObj* as well as cause all future Cells to have that font. If autodisplay is on, this redraws the entire Matrix. Returns **self**.

setIcon:at::

– **setIcon:**(const char *)*iconName*
  at:(int)*row*
  :(int)*col*

Sets the icon of the Cell at (*row*, *col*) to *iconName*. If autodisplay is on, then the Cell is redrawn. Returns **self**.

See also: – **setIcon:** (ButtonCell, Cell)

setIntercell:

– **setIntercell:**(const NXSize *)*aSize*

Sets the width and the height of the space between Cells without redrawing the Matrix, even if autodisplay is on. Returns **self**.

setMode:

– **setMode:**(int)*aMode*

Sets the mode of the Matrix. *aMode* can be one of four constants:

NX_TRACKMODE	Just track the mouse inside the Cells
NX_HIGHLIGHTMODE	Highlight the Cell, then track, then unhighlight
NX_RADIOMODE	Allow no more than one selected Cell
NX_LISTMODE	Allow multiple selected Cells

See also: – **mouseDown:**

setNextText:

– **setNextText:***anObject*

Sets the **nextText** instance variable. When the user presses the Tab key while the last editable entry of the Matrix is being edited, the **selectText:** method is sent to the object represented by **nextText**. A backwards link is automatically created, so that pressing Shift-Tab will move backwards to the previous text via **setPreviousText:**. Returns **self**.

setPreviousText:

– **setPreviousText:***anObject*

Normally you never invoke this method. It is invoked automatically by some other object's **setNextText:** method. It sets the object which will be sent **selectText:** when Shift-Tab is pressed in the Matrix and there are no more fields. Returns **self**.

setPrototype:

– **setPrototype:***aCell*

Sets the **protoCell** instance variable to *aCell* and returns the **id** of the previous **protoCell**. As the new prototype, *aCell* is copied to make any future Cells added to the Matrix.

If you implement your own Cell subclass, then instantiate it as the prototype for your Matrix and make sure your Cell does the right thing when it receives a **copy** message. For example, remember that Object's **copy** copies only pointers, not what they point to—sometimes this is what you want, sometimes not. The best way to implement **copy** when you subclass Cell is to invoke [super **copy**], then copy instance variable values in your subclass individually. Be especially careful that freeing the prototype will not damage any of the copies that were made and put into the Matrix (for example, due to shared pointers).

To stop prototyping, invoke this method with **nil** as the argument, then free the old prototype Cell if no more Cells of that type will be created. If you want to use a prototype cell in other places in the application, it may be useful to copy your prototype when invoking this method, for example:

```
myCellPrototype = [[myCell alloc] init];
[myMatrix setPrototype:[myCellPrototype copy]];
```

This prevents your version of the prototype from being freed when the Matrix is freed.

setReaction:

– **setReaction:**(BOOL)*flag*

If *flag* is NO, prevents the cell from changing back to its previous state; if YES, allows it to revert to reflect unhighlighting. Invoke this from an action method if the action causes the Cell to change in such a way that trying to unhighlight it would be incorrect; for example, if the Cell is deleted or its visual appearance completely changes. Returns **self**.

setScrollable:

– **setScrollable:**(BOOL)*flag*

Sets all the Cells to be scrollable. Returns **self**.

See also: – **setScrollable:** (Cell)

setState:at::

– **setState:**(int)*value*
  at:(int)*row*
  :(int)*col*

Sets the state of the Cell at row *row* and column *col* to *value*. For NX_RADIOMODE Matrices, this is identical to **selectCellAt::** except that the state can be set to any arbitrary *value*. If autodisplay is on, redraws the affected Cell; if the Matrix is in NX_RADIOMODE, the Cell is redrawn regardless of the setting of autodisplay. Returns **self**.

setTag:at::

– **setTag:**(int)*anInt*
  at:(int)*row*
  :(int)*col*

Sets the tag of the Cell at (*row*, *col*) to *anInt* and returns **self**.

setTag:target:action:at::

– **setTag:**(int)*anInt*
  target:*anObject*
  action:(SEL)*aSelector*
  at:(int)*row*
  :(int)*col*

Sets the tag, target object and action method of the Cell at row *row* and column *col*. Returns **self**.

setTarget:

– **setTarget:***anObject*

Sets the target object of the Matrix. This is the target to which actions will be sent during tracking in any Cells that do not have their own target. Returns **self**.

See also: – **action**, – **setAction**, – **target**

setTarget:at::

– **setTarget:***anObject*
  at:(int)*row*
  :(int)*col*

Sets the target of the Cell at row *row* and column *col* to *anObject*. Returns **self**.

setTextDelegate:

– **setTextDelegate:***anObject*

Sets the object to which the Matrix will forward any messages from the field editor (for example, **text:isEmpty:**, **textWillEnd:**, **textDidEnd:endChar:**, **textWillChange:** and **textDidChange:**). Returns **self**.

See also: the Text class

setTitle:at::

– **setTitle:**(const char *)*aString*
  at:(int)*row*
  :(int)*col*

Invoke this method to set the title of the Cell at row *row* and column *col* to *aString*. If autodisplay is on, then the Cell is redrawn. Returns **self**.

See also: – **setTitle:** (ButtonCell)

sizeTo::

– **sizeTo:**(float)*width* **:**(float)*height*

If editing is going on in the Matrix, this aborts the editing, then, after the View is resized, reselects the text to allow editing to continue. Returns **self**.

sizeToCells

– **sizeToCells**

Changes the width and the height of the Matrix frame so that the Matrix's frame contains exactly the Cells. Does not redraw the Matrix. Returns **self**.

sizeToFit

– **sizeToFit**

Changes **cellSize** to accommodate the Cell with the largest contents in the Matrix. Then changes the width and the height of the Matrix frame so that the Matrix's frame contains exactly the Cells. Doesn't redraw the Matrix. Returns **self**.

target

– **target**

Returns the **id** of the Matrix's target object.

See also: – **setTarget:.**

textDelegate

– **textDelegate**

Returns the **id** of the Matrix's text delegate: the object that receives messages from the field editor. The field editor is the Text object used to draw text in all cells in the window. Messages are forwarded to the text delegate by the Matrix.

See also: – **getFieldEditor:for:** (Window)

textDidChange:

– **textDidChange:***textObject*

This message is forwarded to the **textDelegate** if the Matrix has one.

See also: – **textDelegate**

textDidEnd:endChar:

– **textDidEnd:***textObject* **endChar:**(unsigned short)*whyEnd*

Invoked automatically by the system when the text editing ends. If editing ends because the Return key is pressed, then the message [self **sendAction**] is sent. To get the **id** of the Cell in which editing is being performed, use the **selectedCell** method; to access its row or column, use **selectedRow** or **selectedCol**. If editing ends because the Tab key is pressed and the Cell being edited was not the last in the Matrix, then the next Cell is selected. If the Cell is the last one and the **nextText** instance variable is **nil**, the first Cell in the Matrix is selected. Otherwise the **selectText:** message is sent to the object stored in **nextText**. The **textDelegate** (if any) is sent the **textDidEnd:endChar:** message. Returns **self**.

textDidGetKeys:isEmpty:

– **textDidGetKeys:***textObject* **isEmpty:**(BOOL)*flag*

Forwarded to the **textDelegate** (if any). Returns **self**.

textWillChange:

– (BOOL)**textWillChange:***textObject*

Forwarded to the **textDelegate** (if any). Returns **self**.

textWillEnd:

– (BOOL)**textWillEnd:***textObject*

Invoked automatically by the system before text editing ends. It sends the **errorAction** to the target if **isEntryAcceptable:**. The **textDelegate** gets a chance to cancel as well. Returns **self**.

validateSize:

– **validateSize:**(BOOL)*flag*

Allows control over whether the Matrix will invoke **calcSize** the next time it draws. If *flag* is YES, then the size information in the Matrix is assumed correct and will not be recomputed. If *flag* is NO, then **calcSize** will be invoked before any further drawing is done. Returns **self**.

See also: – **calcSize:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving Matrix to the typed stream *stream*. Returns **self**.

CONSTANTS AND DEFINED TYPES

```
/* Matrix Constants */
#define NX_RADIOMODE      0
#define NX_HIGHLIGHTMODE  1
#define NX_LISTMODE       2
#define NX_TRACKMODE      3
```

Menu

INHERITS FROM	Panel : Window : Responder : Object
DECLARED IN	appkit/Menu.h

CLASS DESCRIPTION

The Menu class defines a Panel that contains a single Control object: a Matrix that displays a list of MenuCells.

There are methods for adding both command and submenu items to the Menu. The Menu window can be resized to exactly fit the matrix.

Exactly one Menu created by the application is designated as the "main menu" for the application. This Menu is displayed on top of all other windows whenever the application is active, and it has no close box.

Menus can be made submenus of other menus. A submenu is associated with a particular item in another menu, its "supermenu." Whenever the user clicks the item, the submenu it controls is brought to the screen and "attached" to the controlling supermenu. An item can control only one submenu.

Note that you can drag Menus into your application from Interface Builder's Palettes panel. Several menu items are initialized to work correctly without any additional effort on your part. You can easily set other menu items to display the commands and perform the actions associated with your specific application.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from Window	NXRect	frame;
	id	contentView;
	id	delegate;
	id	firstResponder;
	id	lastLeftHit;
	id	lastRightHit;
	id	counterpart;
	id	fieldEditor;
	int	winEventMask;
	int	windowNum;
	float	backgroundGray;
	struct _wFlags	wFlags;
	struct _wFlags2	wFlags2;

Inherited from Panel	(none)

Declared in Menu

```
id                      supermenu;
id                      matrix;
id                      attachedMenu;
NXPoint                 lastLocation;
id                      reserved;
struct _menuFlags {
    unsigned int            sizeFitted:1;
    unsigned int            autoupdate:1;
    unsigned int            attached:1;
    unsigned int            tornOff:1;
    unsigned int            wasAttached:1;
    unsigned int            wasTornOff:1;
}                       menuFlags;
```

supermenu	The Menu that this Menu is a submenu of.
matrix	The Matrix object used to hold MenuCells.
attachedMenu	The submenu that is currently attached to this Menu. When the user moves or closes a Menu, the attached submenu performs with it.
lastLocation	Last menu location.
reserved	Reserved for future use.
menuFlags.sizeFitted	Set if the menu has been sized to fit the matrix.
menuFlags.autoupdate	Set if the menu wants automatic updating.
menuFlags.attached	Set if the menu is attached to its supermenu.
menuFlags.tornOff	Set if the menu has been torn off of its supermenu.
menuFlags.wasAttached	Set if the menu was attached before tracking.
menuFlags.wasTornOff	Set if the menu was torn off before tracking.

METHOD TYPES

Creating a Menu zone	+ menuZone + setMenuZone:
Initializing a new Menu	– init – initTitle:

Setting up the commands	– addItem:action:keyEquivalent:
	– findCellWithTag:
	– itemList
	– setItemList:
	– setSubmenu:forItem:
	– submenuAction:
Managing menu windows	– close
	– getLocation:forSubmenu:
	– moveTopLeftTo::
	– sizeToFit
	– windowMoved:
Displaying the Menu	– display
	– setAutoupdate:
	– update
Handling events	– mouseDown:
	– rightMouseDown:
Archiving	– awake
	– read:
	– write:

CLASS METHODS

menuZone

+ (NXZone *)**menuZone**

Creates and returns a zone with the name "Menus" in which to allocate new Menus. After invoking this method, you should allocate new Menu instances from this zone.

See also: – **alloc** (Object)

setMenuZone

+ **setMenuZone:**(NXZone *)*aZone*

Sets the zone from which menus will be allocated to *aZone*.

See also: – **alloc** (Object)

INSTANCE METHODS

addItem:action:keyEquivalent:

– **addItem:**(const char *)*aString*
  action:(SEL)*aSelector*
  keyEquivalent:(unsigned short)*charCode*

Creates a new MenuCell, appends it to the receiving Menu, and returns the id of the new cell.

The MenuCell displays *aString* as the command name for the menu item. *aSelector* is the action method the command will invoke. The key equivalent charCode becomes the key equivalent for the cell.

The new MenuCell's target is **nil**, it's automatically enabled, and it has no tag or alternate character string to display. You can change these and other properties of the Cell, including the submenu attribute, by sending direct messages to the returned id.

This method doesn't automatically redisplay the Menu. Upon the next display message, the menu is automatically sized to fit.

See also: – **setSubmenu:forItem:**

awake

– **awake**

Reinitializes and returns a Menu as it's unarchived. Do not invoke this method directly; it's invoked by the **read:** method.

close

– **close**

Overrides Panel's **close** method. If a submenu is attached to the Menu, the attached submenu is also removed from the screen.

See also: – **close** (Window)

display

– **display**

This overrides window's **display** method to provide automatic size-to-fit of the menu window to its matrix. All changes to the matrix that go through the menu methods cause a resizing the next time the Menu is displayed.

See also: – **sizeToFit**

findCellWithTag:

– **findCellWithTag:**(int)*aTag*

Returns the MenuCell that has *aTag* as its tag; returns **nil** if no such cell can be found.

getLocation:forSubmenu:

– **getLocation:**(NXPoint *)*theLocation* **forSubmenu:***aSubmenu*

This message is sent whenever the submenu location is needed. By default, the submenu is to the right of its supermenu, with its titlebar aligned with the supermenu's. You never directly use this method, but may override it to cause the submenu to be attached with a different strategy.

init

– **init**

Initializes and returns the receiver, a new instance of Menu, displaying the title "Menu." All other features are as described in the **initTitle:** method below.

initTitle:

– **initTitle:**(const char *)*aTitle*

Initializes and returns the receiver, a new instance of Menu, displaying the title *aTitle*. The Menu is positioned in the upper left corner of the screen. The Menu's Matrix is initially empty.

The Menu is created as a buffered window initially out of the Window Server's screen list. It must be sent one message to display itself (into the buffer), and another message to move itself on-screen before it will be visible.

The Menu has a style of NX_MENUSTYLE and it has an NX_CLOSEBUTTON button mask. The button isn't shown until the Menu is torn off of its supermenu.

A default matrix is created to contain MenuCell items to display without any intervening space in a single column. The Matrix will use 12-point Helvetica by default to display the items. The matrix will be empty.

Items can be added to the Menu through the **addItem:action:keyEquivalent:** method. The action and key equivalent may both be null. To make a submenu, a **setSubmenu:forItem:** message is sent directly to the Menu.

All Menus have an event mask that excludes keyboard events; they therefore will never become the key window or main window for your application.

See also: – **addItem:action:keyEquivalent:**

itemList

– **itemList**

Returns the matrix of MenuCells used by the Menu.

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

Overrides the View method to allow MenuCell to delegate tracking control to the Menu. Returns **self**.

moveTopLeftTo::

– **moveTopLeftTo:**(NXCoord)*x* **:**(NXCoord)*y*

Repositions the Window on the screen. The arguments specify the new location of the Window's top left corner—the top left corner of its frame rectangle—in screen coordinates.

See also: – **dragFrom::eventNum:** (Window), – **moveTo::** (Window)

read:

– **read:**(NXTypedStream *)*stream*

Reads the Menu from the typed stream *stream*. Returns **self**.

rightMouseDown:

– **rightMouseDown:**(NXEvent *)*theEvent*

Saves the current state of the menu (and its submenus), and pops it up under the mouse position. The menu is tracked as normal, and then the menu's state is restored.

setAutoupdate:

– **setAutoupdate:**(BOOL)*flag*

If *flag* is YES, the menu will respond to the **update** message sent by the Application to all visible Windows after each event (if Application's autoupdating has been enabled). If NO, the Menu won't respond.

See also: – **update**

setItemList:

– **setItemList:***aMatrix*

Sets the Menu's Matrix to *aMatrix*. Subsequent display will size to fit. The previous Matrix is returned.

setSubmenu:forItem:

– **setSubmenu:***aMenu* **forItem:***aCell*

Sets *aMenu* as the submenu controlled by the MenuCell *aCell.*

sizeToFit

– **sizeToFit**

Adjusts the size of the Menu window to its Matrix subview so that they exactly encompass all the commands. Use this method after you're through adding items, modifying the strings they display, or altering the font used to display them. When the Menu is resized, its upper left corner remains fixed. After any resizing that might be necessary, this method will redisplay the Menu.

See also: – **sizeToFit** (Matrix)

submenuAction:

– **submenuAction:***sender*

This message is the action message sent to a submenu by the MenuCell attached to that submenu. If *sender* is in a visible Menu, this action message causes the receiving Menu to attach itself to the menu containing *sender.* Returns **self**.

update

– **update**

Sent to Menu to have the menu update its display. It does this by getting the **updateAction** for each cell and sending it to NXApp. If the **updateMethod** returns YES, the Menu's Matrix is told to redraw the cell using **drawCellAt::**. For this method to have any effect, you must have sent a prior **setAutoupdate:**YES message.

See also: – **setUpdate:**

windowMoved:

– **windowMoved:**(NXEvent *)*theEvent*

Overrides Window method to detach the receiving Menu from its supermenu.

See also: – **windowMoved:** (Window)

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving Menu to the typed stream *stream* and returns **self**.

METHODS IMPLEMENTED BY THE DELEGATE

submenuAction:

– **submenuAction:***sender*

MenuCell

INHERITS FROM	ButtonCell : ActionCell : Cell : Object
DECLARED IN	appkit/MenuCell.h

CLASS DESCRIPTION

MenuCell is a subclass of ButtonCells that appear in Menus. They draw their text left-justified and show an optional key equivalent or submenu arrow on the right.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Cell	char	*contents;
	id	support;
	struct _cFlags1	cFlags1;
	struct _cFlags2	cFlags2;
Inherited from ActionCell	int	tag;
	id	target;
	SEL	action;
Inherited from ButtonCell	char	*altContents;
	union _icon	icon;
	id	sound;
	struct _bcFlags1	bcFlags1;
	struct _bcFlags2	bcFlags2;
	unsigned short	periodicDelay;
	unsigned short	periodicInterval;
Declared in MenuCell	SEL	updateAction;

updateAction	Action used to keep MenuCell's enabled state in synch with the Application.

METHOD TYPES

Initializing a new MenuCell	– init – initTextCell:
Setting the Update Action	– setUpdateAction:forMenu: – updateAction
Querying the MenuCell	– hasSubmenu

Tracking the Mouse	– trackMouse:inRect:ofView:
Setting User Key Equivalents	+ useUserKeyEquivalents: – userKeyEquivalent
Archiving	– read: – write:

INSTANCE METHODS

hasSubmenu

– (BOOL)**hasSubmenu**

Return YES if the MenuCell invokes a submenu, NO otherwise.

init

– init

Initializes and returns the receiver, a new instance of MenuCell, with the default title "MenuItem."

initTextCell:

– **initTextCell:**(const char *)*aString*

Initializes and returns the receiver, a new instance of MenuCell, with *aString* as its title. This method is the designated initializer for the MenuCell class; override this method if you create a subclass of MenuCell that performs its own initialization.

read:

– **read:**(NXTypedStream *)*stream*

Reads the MenuCell from the typed stream *stream*. Returns **self**.

setUpdateAction:forMenu:

– **setUpdateAction:**(SEL)*aSelector* **forMenu:***aMenu*

Sets the **updateAction** for the MenuCell. The **updateAction** is a method that when invoked should set the MenuCell to reflect the current state of the application. This may include enabling or disabling the item, changing the string displayed, or setting the item's state. The **updateAction** takes a single argument, the id of the Cell to update.

The updateAction shouldn't redisplay the Cell itself. Rather it should return YES or NO depending upon whether the Cell needs to be redisplayed.

When an **updateAction** is set for a MenuCell, the Menu passed in *aMenu* is set so it will be automatically updated after each event is processed.

See also: **– update:** (Menu), **– updateWindows:** (Application)

trackMouse:inRect:ofView:

– (BOOL)**trackMouse:**(NXEvent *)*theEvent*
  inRect:(const NXRect *)*cellFrame*
  ofView:*controlView*

Delegates the first event it gets to the Menu. All mouse tracking is handled by Menu.

updateAction

– (SEL)**updateAction**

Returns selector for the updateAction method.

userKeyEquivalent

– **userKeyEquivalent**

Returns the user-assigned key equivalent for the receiving MenuCell.

useUserKeyEquivalents:

+ **useUserKeyEquivalents:**(BOOL)*flag*

If *flag* is YES, then MenuCells can accept user key equivalents. If NO, user key equivalents are disabled.

write:

– **write:**(NXTypedStream *)*stream*

Writes the receiving MenuCell to the typed stream *stream* and returns **self**.

NXBitmapImageRep

INHERITS FROM	NXImageRep : Object
DECLARED IN	appkit/NXBitmapImageRep.h

CLASS DESCRIPTION

An NXBitmapImageRep is an object that can render an image from bitmap data. The data can be in Tag Image File Format (TIFF), or it can be raw image data. If it's raw data, the object must be informed about the structure of the image—its size, the number of color components, the number of bits per sample, and so on—when it's first initialized. If it's TIFF data, the object can get this information from the various TIFF fields included with the data.

Although NXBitmapImageReps are often used indirectly, through instances of the NXImage class, they can also be used directly—to render bitmap images or to produce TIFF representations of them.

Setting Up an NXBitmapImageRep

A new NXBitmapImageRep is passed bitmap data for an image—or told where to find it—when it's first initialized:

- TIFF data can be read from a stream, from a file, or from a section of the __TIFF segment of the application executable. If it's stored in a section or a separate file, the object will delay reading the data until it's needed.
- Raw bitmap data is placed in buffers, and pointers to the buffers are passed to the object.

An NXBitmapImageRep can also be created from bitmap data that's read from an existing (already rendered) image. The object created from this data is able to reproduce the image.

Although the NXBitmapImageRep class inherits NXImageRep methods that set image attributes, these methods shouldn't be used. Instead, you should either allow the object to find out about the image from the TIFF fields or use methods defined in this class to supply this information when the object is initialized.

TIFF Compression

TIFF data can be read and rendered after it has been compressed using any one of the three schemes briefly described below:

LZW	Compresses and decompresses without information loss, achieving compression ratios of anywhere from 2:1 to 3:1. It may be somewhat slower to compress and decompress than the PackBits scheme.
PackBits	Compresses and decompresses without information loss, but may not achieve the same compression ratios as LZW.
JPEG	Compresses and decompresses with some information loss, but can achieve compression ratios anywhere from 10:1 to 100:1. The ratio is determined by a user-settable factor ranging from 1.0 to 255.0, with higher factors yielding greater compression. More information is lost with greater compression, but 15:1 compression is safe for publication quality. Some images can be compressed even more. JPEG compression can be used only for images that specify at least 4 bits per sample.

An NXBitmapImageRep can also produce compressed TIFF data for its image using any of these schemes.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from NXImageRep	NXSize	size;
Declared in NXBitmapImageRep	(none)	

METHOD TYPES

Initializing a new NXBitmapImageRep object

- initFromSection:
- initFromFile:
- initFromStream:
- initData:fromRect:
- initData:pixelsWide:pixelsHigh:
  bitsPerSample:samplesPerPixel:
  hasAlpha:isPlanar:colorSpace:
  bytesPerRow:bitsPerPixel:
- initDataPlanes:pixelsWide:pixelsHigh:
  bitsPerSample:samplesPerPixel:
  hasAlpha:isPlanar:colorSpace:
  bytesPerRow:bitsPerPixel:

Creating a List of NXBitmapImageReps	
	+ newListFromSection:
	+ newListFromSection:zone:
	+ newListFromFile:
	+ newListFromFile:zone:
	+ newListFromStream:
	+ newListFromStream:zone:
Reading information from a rendered image	
	+ sizeImage:
	+ sizeImage:pixelsWide:pixelsHigh: bitsPerSample:samplesPerPixel: hasAlpha:isPlanar:colorSpace:
Copying and freeing an NXBitmapImageRep	
	– copy
	– free
Getting information about the image	
	– bitsPerPixel
	– samplesPerPixel
	– bitsPerSample (NXImageRep)
	– isPlanar
	– numPlanes
	– numColors (NXImageRep)
	– hasAlpha (NXImageRep)
	– bytesPerPlane
	– bytesPerRow
	– colorSpace
	– pixelsWide (NXImageRep)
	– pixelsHigh (NXImageRep)
Getting image data	– data
	– getDataPlanes:
Drawing the image	– draw
	– drawIn:
	– drawAt: (NXImageRep)
Producing a TIFF representation of the image	
	– writeTIFF:
	– writeTIFF:usingCompression:
	– writeTIFF:usingCompression:andFactor:
Archiving	– read:
	– write:

CLASS METHODS

newListFromFile:

+ (List *)**newListFromFile:**(const char *)*filename*

Creates one new NXBitmapImageRep instance for each TIFF image specified in the *filename* file, and returns a List object containing all the objects created. If no NXBitmapImageReps can be created (for example, if *filename* doesn't exist or doesn't contain TIFF data), **nil** is returned. The List should be freed when it's no longer needed.

Each new NXBitmapImageRep is initialized by the **initFromFile:** method, which reads information about the image from *filename*, but not the image data. The data will be read when it's needed to render the image.

See also: + **newListFromFile:zone:**, – **initFromFile:**

newListFromFile:zone:

+ (List *)**newListFromFile:**(const char *)*filename* **zone:**(NXZone *)*aZone*

Returns a List of new NXBitmapImageRep instances, just as **newListFromFile:** does, except that the List object and the NXBitmapImageReps are allocated from memory located in *aZone*.

See also: + **newListFromFile:**, – **initFromFile:**

newListFromSection:

+ (List *)**newListFromSection:**(const char *)*name*

Creates one new NXBitmapImageRep instance for each TIFF image specified in the *name* section of the __TIFF segment in the executable file, and returns a List object containing all the objects created. If not even one NXBitmapImageRep can be created (for example, if the *name* section doesn't exist or doesn't contain TIFF data), **nil** is returned. The List should be freed when it's no longer needed.

Each new NXBitmapImageRep is initialized by the **initFromSection:** method, which reads information about the image from the section, but doesn't read image data. The data will be read when it's needed to render the image.

See also: + **newListFromSection:zone:**, – **initFromSection:**

newListFromSection:zone:

+ (List *)**newListFromSection:**(const char *)*name* **zone:**(NXZone *)*aZone*

Returns a List of new NXBitmapImageRep instances, just as **newListFromSection:** does, except that the List object and the NXBitmapImageReps are allocated from memory located in *aZone*.

See also: + **newListFromSection:**, – **initFromSection:**

newListFromStream:

+ (List *)**newListFromStream:**(NXStream *)*stream*

Creates one new NXBitmapImageRep instance for each TIFF image that can be read from *stream*, and returns a List object containing all the objects created. If not even one NXBitmapImageRep can be created (for example, if the *stream* doesn't contain TIFF data), **nil** is returned. The List should be freed when it's no longer needed.

The data is read and each new object initialized by the **initFromStream:** method.

See also: + **newListFromStream:zone:**, – **initFromStream:**

newListFromStream:zone:

+ (List *)**newListFromStream:**(NXStream *)*stream* **zone:**(NXZone *)*aZone*

Returns a List of new NXBitmapImageRep instances, just as **newListFromStream:** does, except that the NXBitmapImageReps and the List object are allocated from memory located in *aZone*.

See also: + **newListFromStream:**, – **initFromStream:**

sizeImage:

+ (int)**sizeImage:**(const NXRect *)*rect*

Returns the number of bytes that would be required to hold bitmap data for the rendered image bounded by the *rect* rectangle. The rectangle is located in the current window and is specified in the current coordinate system.

See also: + **sizeImage:pixelsWide:pixelsHigh:bitsPerSample:samplesPerPixel: hasAlpha:isPlanar:colorSpace:**, – **initData:fromRect:**

sizeImage:pixelsWide:pixelsHigh:bitsPerSample:samplesPerPixel:hasAlpha: isPlanar:colorSpace:

+ (int)**sizeImage:**(const NXRect *)*rect*
  pixelsWide:(int *)*width*
  pixelsHigh:(int *)*height*
  bitsPerSample:(int *)*bps*
  samplesPerPixel:(int *)*spp*
  hasAlpha:(BOOL *)*alpha*
  isPlanar:(BOOL *)*config*
  colorSpace:(NXColorSpace *)*space*

Returns the number of bytes that would be required to hold bitmap data for the rendered image bounded by the *rect* rectangle. The rectangle is located in the current window and is specified in the current coordinate system.

Every argument but *rect* is a pointer to a variable where the method will write information about the image. For an explanation of the information provided, see the description of the **initDataPlanes:**... method

See also: – **initDataPlanes:pixelsWide:pixelsHigh:bitsPerSample: samplesPerPixel:hasAlpha:isPlanar:colorSpace:bytesPerRow:bitsPerPixel:**

INSTANCE METHODS

bitsPerPixel

– (int)**bitsPerPixel**

Returns the number of bits allocated for each pixel in each plane of data. This is normally equal to the number of bits per sample or, if the data is in meshed configuration, the number of bits per sample times the number of samples per pixel. It can be explicitly set to another value (in the **initData:**... or **initDataPlanes:**...method) in case extra memory is allocated for each pixel. This may be the case, for example, if pixel data is aligned on byte boundaries.

However, in the current release, an NXBitmapImageRep cannot render an image that has empty memory separating pixel specifications.

bytesPerPlane

– (int)**bytesPerPlane**

Returns the number of bytes in each plane or channel of data. This will be figured from the number of bytes per row and the height of the image.

See also: – **bytesPerRow**

bytesPerRow

– (int)**bytesPerRow**

Returns the minimum number of bytes required to specify a scan line (a single row of pixels spanning the width of the image) in each data plane. If not explicitly set to another value (in the **initData:**... or **initDataPlanes:**... method), this will be figured from the width of the image, the number of bits per sample, and, if the data is in a meshed configuration, the number of samples per pixel. It can be set to another value to indicate that each row of data is aligned on word or other boundaries.

However, in the current release, an NXBitmapImageRep can't render an image that has empty space at the end of a scan line.

colorSpace

– (NXColorSpace)**colorSpace**

Returns one of the following enumerated values, which indicate how bitmap data is to be interpreted:

NX_OneIsBlack	A gray scale where 1 means black and 0 means white
NX_OneIsWhite	A gray scale where 0 means black and 1 means white
NX_RGBColorSpace	Red, green, and blue color values
NX_CMYKColorSpace	Cyan, magenta, yellow, and black color values

These values are defined in the header file **appkit/graphics.h**.

See also: – **numColors** (NXImageRep)

copy

– **copy**

Returns a new NXBitmapImageRep instance that's an exact copy of the receiver. The new object will have its own copy of the bitmap data, unless the receiver merely references the data. In that case, both objects will reference the same data.

The new object doesn't need to be initialized.

data

– (unsigned char *)**data**

Returns a pointer to the bitmap data. If the data is in planar configuration, this pointer will be to the first plane. To get separate pointers to each plane, use the **getDataPlanes:** method.

See also: – **getDataPlanes:**

draw

– (BOOL)**draw**

Renders the image at (0.0, 0.0) in the current coordinate system on the current device using the appropriate PostScript imaging operator. This method returns YES if successful in producing the image, and NO if not.

See also: – **drawAt:** (NXImageRep), – **drawIn:**

drawIn:

– (BOOL)**drawIn:**(const NXRect *)*rect*

Renders the image so that it fits inside the rectangle referred to by *rect*. The current coordinate system is translated and scaled so the image will appear at the right location and fit within the rectangle. The **draw** method is then invoked to render the image. This method passes through the return value of the **draw** method, which indicates whether the image was successfully drawn.

The coordinate system is not restored after it has been altered.

See also: – **draw**, – **drawAt:** (NXImageRep)

free

– **free**

Deallocates the NXBitmapImageRep. This method will not free any bitmap data that the object merely references—that is, raw data that was passed to it in a **initData:**... or **initDataPlanes:**... message.

getDataPlanes:

– **getDataPlanes:**(unsigned char **)*thePlanes*

Provides bitmap data for the image separated into planes. *thePlanes* should be an array of five character pointers. If the bitmap data is in planar configuration, each pointer will be initialized to point to one of the data planes. If there are less than five planes, the remaining pointers will be set to NULL. If the bitmap data is in meshed configuration, only the first pointer will be initialized; the others will be NULL. Returns **self**.

Color components in planar configuration are arranged in the expected order—for example, red before green before blue for RGB color. All color planes precede the coverage plane.

See also: – **data**, – **isPlanar**

init

Generates an error message. This method cannot be used to initialize an NXBitmapImageRep. Instead, use one of the methods listed under "See also" below.

See also: – **initFromSection:**, – **initFromFile:**, – **initFromStream:**, – **initDataPlanes:pixelsWide:pixelsHigh:bitsPerSample:samplesPerPixel: hasAlpha:isPlanar:colorSpace:bytesPerRow:bitsPerPixel:**, – **initData:pixelsWide:pixelsHigh:bitsPerSample:samplesPerPixel: hasAlpha:isPlanar:colorSpace:bytesPerRow:bitsPerPixel:**, – **initData:fromRect:**

initData:fromRect:

– **initData:**(unsigned char *)*data* **fromRect:**(const NXRect *)*rect*

Initializes the receiver, a newly allocated NXBitmapImageRep object, with bitmap data read from a rendered image. The image that's read is located in the current window and is bounded by the *rect* rectangle as specified in the current coordinate system.

This method uses PostScript imaging operators to read the image data into the *data* buffer; the object is then created from that data. The object is initialized with information about the image obtained from the Window Server.

If *data* is NULL, the NXBitmapImageRep will allocate enough memory to hold bitmap data for the image. In this case, the buffer will belong to the object and will be freed when the object is freed.

If *data* is not NULL, you must make sure the buffer is large enough to hold the image bitmap. You can determine how large it needs to be by sending a **sizeImage:** message for the same rectangle. The NXBitmapImageRep will only reference the data in the buffer; the buffer won't be freed when the object is freed.

If for any reason the new object can't be initialized, this method frees it and returns **nil**. Otherwise, it returns the initialized object (**self**).

See also: + **sizeImage:**

initData:pixelsWide:pixelsHigh:bitsPerSample:samplesPerPixel:hasAlpha: isPlanar:colorSpace:bytesPerRow:bitsPerPixel:

– **initData:**(unsigned char *)*data*
    pixelsWide:(int)*width*
    pixelsHigh:(int)*height*
    bitsPerSample:(int)*bps*
    samplesPerPixel:(int)*spp*
    hasAlpha:(BOOL)*alpha*
    isPlanar:(BOOL)*config*
    colorSpace:(NXColorSpace)*space*
    bytesPerRow:(int)*rowBytes*
    bitsPerPixel:(int)*pixelBits*

Initializes the receiver, a newly allocated NXBitmapImageRep object, so that it can render the image specified in *data* and described by the other arguments. If the object can't be initialized, this method frees it and returns **nil**. Otherwise, it returns the object (**self**).

data points to a buffer containing raw bitmap data. If the data is in planar configuration (*config* is YES), all the planes must follow each other in the same buffer. The **initDataPlanes:**... method can be used instead of this one if there are separate buffers for each plane.

If *data* is NULL, this method allocates a data buffer large enough to hold the image described by the other arguments. You can then obtain a pointer to this buffer (with the **data** or **getDataPlanes:** method) and fill in the image data. In this case the buffer will belong to the object and will be freed when it's freed.

If *data* is not NULL, the object will only reference the image data; it won't copy it. The buffer won't be freed when the object is freed.

All the other arguments to this method are the same as those to **initDataPlanes:**... See that method for descriptions.

See also: – **initDataPlanes:pixelsWide:pixelsHigh:bitsPerSample: samplesPerPixel:hasAlpha:isPlanar:colorSpace:bytesPerRow:bitsPerPixel:**

initDataPlanes:pixelsWide:pixelsHigh:bitsPerSample:samplesPerPixel: hasAlpha:isPlanar:colorSpace:bytesPerRow:bitsPerPixel:

– **initDataPlanes:**(unsigned char **)*planes*
  pixelsWide:(int)*width*
  pixelsHigh:(int)*height*
  bitsPerSample:(int)*bps*
  samplesPerPixel:(int)*spp*
  hasAlpha:(BOOL)*alpha*
  isPlanar:(BOOL)*config*
  colorSpace:(NXColorSpace)*space*
  bytesPerRow:(int)*rowBytes*
  bitsPerPixel:(int)*pixelBits*

Initializes the receiver, a newly allocated NXBitmapImageRep object, so that it can render the image specified in *planes* and described by the other arguments. If the object can't be initialized, this method frees it and returns **nil**. Otherwise, it returns the object (**self**).

planes is an array of character pointers, each of which points to a buffer containing raw image data. If the data is in planar configuration, each buffer holds one component—one plane—of the data. Color planes are arranged in the standard order—for example, red before green before blue for RGB color. All color planes precede the coverage plane.

If the data is in meshed configuration (*config* is NO), only the first buffer is read. The **initData:**... method can be used instead of this one for data in meshed configuration.

If *planes* is NULL or if it's an array of NULL pointers, this method allocates enough memory to hold the image described by the other arguments. You can then obtain pointers to this memory (with the **getDataPlanes:** or **data** method) and fill in the image data. In this case, the allocated memory will belong to the object and will be freed when it's freed.

If *planes* is not NULL and the array contains at least one data pointer, the object will only reference the image data; it won't copy it. The buffers won't be freed when the object is freed.

Each of the other arguments (besides *planes*) informs the NXBitmapImageRep object about the image. They're explained below:

- *width* and *height* specify the size of the image in pixels. The size in each direction must be greater than 0.
- *bps* (bits per sample) is the number of bits used to specify one pixel in a single component of the data. All components are assumed to have the same bits per sample.

- *spp* (samples per pixel) is the number of data components. It includes both color components and the coverage component (alpha), if present. Meaningful values range from 1 through 5. An image with cyan, magenta, yellow, and black (CMYK) color components plus a coverage component would have an *spp* of 5; a gray-scale image that lacks a coverage component would have an *spp* of 1.

- *alpha* should be YES if one of the components counted in the number of samples per pixel (*spp*) is a coverage component, and NO if there is no coverage component.

- *config* should be YES if the data components are laid out in a series of separate "planes" or channels ("planar configuration"), and NO if component values are interwoven in a single channel ("meshed configuration").

  For example, in meshed configuration, the red, green, blue, and coverage values for the first pixel of an image would precede the red, green, blue, and coverage values for the second pixel, and so on. In planar configuration, red values for all the pixels in the image would precede all green values, which would precede all blue values, which would precede all coverage values.

- *space* indicates how data values are to be interpreted. It should be one of the following enumerated values (defined in the header file **appkit/graphics.h**):

NX_OneIsBlack	A gray scale between 1 (black) and 0 (white)
NX_OneIsWhite	A gray scale between 0 (black) and 1 (white)
NX_RGBColorSpace	Red, green, and blue color values
NX_CMYKColorSpace	Cyan, magenta, yellow, and black color values

- *rowBytes* is the number of bytes that are allocated for each scan line in each plane of data. A scan line is a single row of pixels spanning the width of the image.

  Normally, *rowBytes* can be figured from the *width* of the image, the number of bits per pixel in each sample (*bps*), and, if the data is in a meshed configuration, the number of samples per pixel (*spp*). However, if the data for each row is aligned on word or other boundaries, it may have been necessary to allocate more memory for each row than there is data to fill it. *rowBytes* lets the object know whether that's the case. In the current release, an NXBitmapImageRep cannot render an image with empty space at the end of a scan line.

  If *rowBytes* is 0, the NXBitmapImageRep assumes that there's no empty space at the end of a row.

- *pixelBits* informs the NXBitmapImageRep how many bits are actually allocated per pixel in each plane of data. If the data is in planar configuration, this normally equals *bps* (bits per sample). If the data is in meshed configuration, it normally equals *bps* times *spp* (samples per pixel). However, it's possible for a pixel specification to be followed by some meaningless bits (empty space), as may happen, for example, if pixel data is aligned on byte boundaries. In the current release, an NXBitmapImageRep cannot render an image if this is the case.

  If *pixelBits* is 0, the object will interpret the number of bits per pixel to be the expected value, without any meaningless bits.

This method is the designated initializer for NXBitmapImageReps that handle raw image data.

See also: – **initData:pixelsWide:pixelsHigh:bitsPerSample:samplesPerPixel: hasAlpha:isPlanar:colorSpace:bytesPerRow:bitsPerPixel:**

initFromFile:

– **initFromFile:**(const char *)*filename*

Initializes the receiver, a newly allocated NXBitmapImageRep object, with the TIFF image found in the *filename* file. This method reads some information about the image from *filename*, but not the image itself. Image data will be read when it's needed to render the image.

If the new object can't be initialized for any reason (for example, *filename* doesn't exist or doesn't contain TIFF data), this method frees it and returns **nil**. Otherwise, it returns **self**.

This method is the designated initializer for NXBitmapImageReps that read image data from a file.

See also: + **newListFromFile:**, – **initFromSection:**

initFromSection:

– **initFromSection:**(const char *)*name*

Initializes the receiver, a newly allocated NXBitmapImageRep object, with the TIFF image found in the *name* section in the __TIFF segment of the application executable. This method reads some information about the image from the section, but not the image itself. Image data is read only when it's needed to render the image.

If the new object can't be initialized for any reason (for example, the *name* section doesn't exist or doesn't contain TIFF data), this method frees it and returns **nil.** Otherwise, it returns **self**.

This method is the designated initializer for NXBitmapImageReps that read image data from a section of the __TIFF segment.

See also: + **newListFromSection:**, – **initFromFile:**

initFromStream:

– **initFromStream:**(NXStream *)*stream*

Initializes the receiver, a newly allocated NXBitmapImageRep object, with the TIFF image read from *stream.* If the new object can't be initialized for any reason (for example, *stream* doesn't contain TIFF data), this method frees it and returns **nil.** Otherwise, it returns **self**.

This method is the designated initializer for NXBitmapImageReps that read image data from a stream.

See also: + **newListFromStream:**

isPlanar

– (BOOL)**isPlanar**

Returns YES if image data is segregated into a separate plane for each color and coverage component (planar configuration), and NO if the data is integrated into a single plane (meshed configuration).

See also: – **samplesPerPixel**

numPlanes

– (int)**numPlanes**

Returns the number of separate planes that image data is organized into. This will be the number of samples per pixel if the data has a separate plane for each component (**isPlanar** returns YES) and 1 if the data is meshed (**isPlanar** returns NO).

See also: – **isPlanar**, – **samplesPerPixel**, – **hasAlpha**, – **numColors** (NXImageRep)

read:

– **read:**(NXTypedStream *)*stream*

Reads the NXBitmapImageRep from the typed stream *stream.*

See also: – **write:**

samplesPerPixel

– (int)**samplesPerPixel**

Returns the number of components in the data. It includes both color components and the coverage component, if present.

See also: – **hasAlpha**, – **numColors** (NXImageRep)

write:

– **write:**(NXTypedStream *)*stream*

Writes the NXBitmapImageRep to the typed stream *stream.*

See also: – **read:**

writeTIFF:

– **writeTIFF:**(NXStream *)*stream*

Writes a TIFF representation of the image to *stream.* This method is equivalent to **writeTIFF:usingCompression:andFactor:** when NX_TIFF_COMPRESSION_NONE is passed as the second argument. The TIFF data is not compressed.

See also: – **writeTIFF:usingCompression:andFactor:**

writeTIFF:usingCompression:

– **writeTIFF:**(NXStream *)*stream* **usingCompression:**(int)*compression*

Writes a TIFF representation of the image to *stream*, compressing the data according to the *compression* scheme. This method is equivalent to **writeTIFF:usingCompression:andFactor:** when 0.0 is passed as the third argument. If *compression* is NX_TIFF_COMPRESSION_JPEG, the default compression factor will be used. This and the other *compression* constants are listed under the next method.

See also: – **writeTIFF:usingCompression:andFactor:**

writeTIFF:usingCompression:andFactor:

– **writeTIFF:**(NXStream *)*stream*
usingCompression:(int)*compression*
andFactor:(float)*factor*

Writes a TIFF representation of the image to *stream.* If the stream isn't currently positioned at location 0, this method assumes that it contains another TIFF image. It will try to append the TIFF representation it writes to that image. To do this, it must read the header of the image already in the stream. Therefore, the stream must be opened with NX_READWRITE permission.

The second argument, *compression*, indicates whether or not the data should be compressed and, if so, which compression scheme to use. It should be one of the following constants:

NX_TIFF_COMPRESSION_LZW	LZW compression
NX_TIFF_COMPRESSION_PACKBITS	PackBits compression
NX_TIFF_COMPRESSION_JPEG	JPEG compression
NX_TIFF_COMPRESSION_NONE	No compression

The third argument, *factor*, is used in the JPEG scheme to determine the degree of compression. If *factor* is 0.0, the default compression factor of 10.0 will be used. Otherwise, *factor* should fall within the range 1.0–255.0, with higher values yielding greater compression but also greater information loss.

The compression schemes are discussed briefly under "CLASS DESCRIPTION" above.

NXBrowser

INHERITS FROM	Control : View : Responder : Object
DECLARED IN	appkit/NXBrowser.h

CLASS DESCRIPTION

NXBrowser provides a user interface for displaying and selecting hierarchically organized data such as directory paths. The levels of the hierarchy are displayed in columns. Columns are numbered from left to right, beginning with 0. Each column consists of a ScrollView or ClipView containing a Matrix filled with NXBrowserCells. NXBrowser must have a delegate; the delegate's role is to provide the data that fills the columns as the user navigates through the hierarchy.

You can implement one of three delegate types—normal, lazy, or very-lazy—depending on your needs for performance and memory use. A normal delegate implements the **browser:fillMatrix:inColumn:** method; implemented alone, this method may improve performance if the data space is small, since it always creates and loads all the entries in a column. A lazy delegate implements the **browser:fillMatrix:inColumn:** and **browser:loadCell:atRow:inColumn:** methods; lazy delegates create all cells in a column, but they load only those that are displayed. A very-lazy delegate implements the **browser:loadCell:atRow:inColumn:** and **browser:getNumRowsInColumn:** methods. Very-lazy delegates make spare use of memory by not creating a cell for an entry until it's to be displayed; this is useful for large, potentially open-ended data spaces. A delegate must implement either the normal, lazy, or very-lazy methods; however, it shouldn't implement both the **browser:fillMatrix:inColumn:** and **browser:getNumRowsInColumn:** methods.

An entry in NXBrowser's columns can be either a branch node (such as a directory) or a leaf node (such as a file). As the delegate loads an entry in a Cell, it invokes NXBrowserCell's **setLeaf:** method to specify the type of entry. When the user selects a single branch node entry in a column, the NXBrowser sends itself the **addColumn** message, which messages the delegate to load the next column. NXBrowser can be set to allow selection of multiple entries in a column, or to limit selection to a single entry. When set for multiple selection, it can also be set to limit multiple selection to leaf nodes only, or to allow selection of both types of nodes together.

As a subclass of Control, NXBrowser has a target object and action message. Each time the user selects one or more entries in a column, the action message is sent to the target.

You can change the appearance and user interface features of NXBrowser in a number of ways. Columns in the NXBrowser may have up and down scroll buttons, scroll bars, both, or neither. The NXBrowser itself may or may not have left and right scroll buttons. You generally won't create NXBrowser without scrollers; if you do, you must make sure the bounds rectangle of the NXBrowser is large enough that all its rows and columns can be displayed. The NXBrowser's columns may be bordered and titled, bordered and untitled, or unbordered and untitled. A column's title may be taken from the selected entry in the column to its left, or may be provided explicitly by NXBrowser or its delegate.

You can drag NXBrowser into an application from the Interface Builder Palettes panel. Interface Builder provides easier ways to set many of the user interface features described previously.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from View	NXRect	frame;
	NXRect	bounds;
	id	superview;
	id	subviews;
	id	window;
	struct __vFlags	vFlags;
Inherited from Control	int	tag;
	id	cell;
	struct _conFlags	conFlags;
Defined in NXBrowser	id	target;
	id	delegate;
	SEL	action;
	SEL	doubleAction;
	id	matrixClass;
	id	cellPrototype;
	unsigned short	pathSeparator;

target	The object notified by NXBrowser when one or more items are selected in a column.
delegate	The object providing the data which is browsed by the NXBrowser.
action	The message sent to the target when one or more entries are selected in a column.
doubleAction	The message sent to the target when an entry in the NXBrowser is double-clicked.
matrixClass	The class used to instantiate the matrices in the columns of NXBrowser; Matrix by default.
pathSeparator	The character which separates the substrings of a path (see **getPath:ToColumn:**, **setPath:**).

METHOD TYPES

Initializing and freeing	– initFrame:
	– free
Setting the delegate	– delegate
	– setDelegate:
Setting target and action	– action
	– setAction:
	– target
	– setTarget:
	– doubleAction
	– setDoubleAction:
Setting the Matrix class	– setMatrixClass:
Setting the Cell class	– setCellClass:
	– cellPrototype
	– setCellPrototype:
Setting NXBrowser behavior	– allowMultiSel:
	– allowBranchSel:
	– reuseColumns:
	– acceptArrowKeys:
	– acceptsFirstResponder
	– setEnabled:
	– hideLeftAndRightScrollButtons:
	– useScrollButtons:
	– useScrollBars:
Setting NXBrowser appearance	– setMinColumnWidth:
	– minColumnWidth
	– setMaxVisibleColumns:
	– maxVisibleColumns
	– numVisibleColumns
	– firstVisibleColumn
	– lastVisibleColumn
	– lastColumn
	– separateColumns:
	– columnsAreSeparated

Manipulating columns	– loadColumnZero
	– isLoaded
	– addColumn
	– reloadColumn:
	– displayColumn:
	– displayAllColumns
	– setLastColumn:
	– selectAll:
	– selectedColumn
	– columnOf:
	– validateVisibleColumns
Manipulating column titles	– getTitleFromPreviousColumn:
	– isTitled
	– setTitled:
	– getTitleFrame:ofColumn:
	– setTitle:ofColumn:
	– drawTitle:inRect:ofColumn:
	– clearTitleInRect:ofColumn:
	– titleHeight
	– titleOfColumn:
Scrolling the NXBrowser	– scrollColumnsRightBy:
	– scrollColumnsLeftBy:
	– scrollColumnToVisible:
	– scrollUpOrDown:
	– reflectScroll:
Event handling	– mouseDown:
	– keyDown:
	– doClick:
	– doDoubleClick:
Getting column Matrices and Cells	– getLoadedCellAtRow:inColumn:
	– matrixInColumn:
Getting column frames	– getFrame:ofColumn:
	– getFrame:ofInsideOfColumn:
Paths	– setPathSeparator:
	– setPath:
	– getPath:toColumn:
Drawing	– drawSelf::
Resizing the NXBrowser	– sizeTo::
	– sizeToFit
Arranging NXBrowser components	
	– tile

acceptArrowKeys:

– **acceptArrowKeys:**(BOOL)*flag*

Sets NXBrowser handling of arrow key input. If *flag* is YES, then the keyboard arrow keys move the selection whenever the NXBrowser or one of its subviews is the first responder; if *flag* is NO, arrow key input has no effect. Returns **self**.

acceptsFirstResponder

– (BOOL)**acceptsFirstResponder**

Returns YES if the NXBrowser accepts arrow key input; NO otherwise. The default setting is NO.

See also: – **acceptArrowKeys:**

action

– (SEL)**action**

Returns the action sent to the target by the NXBrowser when the user makes a selection in one of its columns.

See also: – **doubleAction**, – **setAction:**, – **setDoubleAction:**

addColumn

– **addColumn**

Adds a column to the right of the last column in the NXBrowser and, if necessary, scrolls the NXBrowser so that the new column is visible. You never invoke this method; it's invoked by **doClick:** and **keyDown:** when the user selects a single branch node entry in the NXBrowser, and by s**etPath:** when it matches a path substring with a branch node entry. Returns **self**.

See also: – **loadColumnZero**, – **reloadColumn:**, – **setPath:**

allowBranchSel:

– **allowBranchSel:**(BOOL)*flag*

Sets whether the user can select multiple branch and leaf node entries. If *flag* is YES and multiple selection is enabled (by **allowMultiSel:**), then multiple branch and leaf node entries can be selected. By default, a user can choose only multiple leaf node entries when multiple entry selection is enabled. Returns **self**.

See also: – **allowMultiSel:**

allowMultiSel:

– **allowMultiSel:**(BOOL)*flag*

Sets whether the user can select multiple entries in a column. If *flag* is YES, the user can choose any number of leaf entries in a column (or leaf and branch entries if enabled by **allowBranchSel:**). By default, the user can choose just one entry in a column at a time. Returns **self.**

See also: – **allowBranchSel:**

cellPrototype

– **cellPrototype**

Returns the NXBrowser's prototype cell. This cell is copied to create new cells in the columns of the NXBrowser.

See also: – **setCellPrototype:**

clearTitleInRect:ofColumn:

– **clearTitleInRect:**(const NXRect *)*aRect* **ofColumn:**(int)*column*

Clears the title displayed in *aRect* above *column.* You don't invoke this method directly; it's called whenever a title of a column needs to be cleared. You can override this method if you draw your own column titles. *aRect* is in the NXBrowser's coordinate system. Returns **self.**

columnOf:

– (int)**columnOf:***matrix*

Returns the index of the column containing *matrix*; the leftmost (root) column is 0. Returns –1 if no column contains *matrix.*

See also: – **matrixInColumn:**

columnsAreSeparated

– (BOOL)**columnsAreSeparated**

Returns YES if columns are separated by a bezeled bar; NO otherwise. If the NXBrowser is set to display column titles, its columns are automatically separated by bezels; however, the value returned by this method is not changed by the **setTitled:** method.

See also: – **separateColumns:**, – **setTitled:**

delegate

– **delegate**

Returns the delegate of the NXBrowser, the object that provides the data to be browsed.

See also: – **setDelegate:**, "METHODS IMPLEMENTED BY THE DELEGATE"

displayAllColumns

– **displayAllColumns**

Causes columns currently visible in the NXBrowser to be redisplayed. You can call this to update the NXBrowser after manipulating it with display disabled in the window. Returns **self**.

displayColumn:

– **displayColumn:**(int)*column*

Validates and displays column number *column*. You can call this method to update the NXBrowser after manipulating it with display disabled in *column*. Returns **self**.

See also: – **displayAllColumns**

doClick:

– **doClick:***sender*

You never invoke this method. This is the action message sent to the NXBrowser by a column's Matrix when a mouse-down event occurs in a column. It sets the **lastColumn** to that of the Matrix where the click occurred, and removes any columns to the right that were previously loaded in the NXBrowser. If a single branch node entry is selected by the event, this method sends **addColumn** to **self** to display the corresponding data in the column to the right. It sends the NXBrowser's action message to its target and returns **self**.

See also: – **action**, – **setAction**, – **setTarget**, – **target**

doDoubleClick:

– **doDoubleClick:***sender*

You never invoke this method. This is the action message sent to the NXBrowser by a column's Matrix when a double-click occurs in a column. This method simply sends the doubleAction message to the target; if no doubleAction message is set, it sends the action. Override this method to add specific behavior for double-click events. Returns **self**.

See also: – **doubleAction**, – **setDoubleAction**, – **setTarget**, – **target**

doubleAction

– (SEL)**doubleAction**

Returns the action sent by the NXBrowser to its target when the user double-clicks on an entry. If no doubleAction message is specified, this method returns the action.

See also: – **setDoubleAction:**

drawSelf::

– **drawSelf:**(const NXRect *)*rects* **:**(int)*rectCount*

Draws the NXBrowser; loads column 0 if it has not been loaded. Override this method if you change the way NXBrowser draws itself. You never invoke this method; it's invoked by the **display** method. Returns **self**.

drawTitle:inRect:ofColumn:

– **drawTitle:**(const char *)*title*
  inRect:(const NXRect *)*aRect*
  ofColumn:(int)*column*

You never invoke this method. It's invoked whenever the NXBrowser needs to draw a column title. You may override it if you draw your own column titles. Returns **self**.

firstVisibleColumn

– (int)**firstVisibleColumn**

Returns the index of the leftmost visible column.

See also: – **lastVisibleColumn**

free

– **free**

Frees the NXBrowser and all the objects it manages: scrollviews, matrices, cells, scroll buttons, prototypes, and so on. Returns **nil**.

getFrame:ofColumn:

– (NXRect *)**getFrame:**(NXRect *)*theRect* **ofInsideOfColumn:**(int)*column*

Returns a pointer to the rectangle (in NXBrowser coordinates) containing *column*; the pointer is returned both explicitly by the method and implicitly in *theRect*. The returned rectangle includes the bezel area surrounding the column. If *column* isn't currently loaded or displayed, this method returns NULL explicitly, without changing the coordinates of the rectangle represented in *theRect*. It also returns NULL if *theRect* is NULL.

getFrame:ofInsideOfColumn:

– (NXRect *)**getFrame:**(NXRect *)*theRect* **ofInsideOfColumn:**(int)*column*

Returns a pointer to the rectangle (in NXBrowser coordinates) containing the "insides" of *column*; the pointer is returned both explicitly by the method and implicitly in *theRect*. The "insides" are defined as the area in the column that contains the cells and only that area (i.e., no bezels). If *column* isn't currently loaded or displayed, this method returns NULL explicitly, without changing the coordinates of the rectangle represented in *theRect*. It also returns NULL if *theRect* is NULL.

getLoadedCellAtRow:inColumn:

– **getLoadedCellAtRow:**(int)*row* **inColumn:**(int)*column*

Returns the cell at *row* in *column*, if that column is currently in the NXBrowser. This method creates and loads the cell if necessary. It's the safest way to get a particular cell in a column, since lazy delegates don't load every cell in a matrix and very-lazy delegates don't even create all cells until they're displayed. This method is preferred to the Matrix method **cellAt::**. If the specified *column* isn't in the NXBrowser, or if *row* doesn't exist in *column*, returns **nil**.

getPath:toColumn:

– (char *)**getPath:**(char *)*thePath* **toColumn:**(int)*column*

Returns a pointer to the string representing the path to *column*, both explicitly and in *thePath*. Before invoking this method, you must allocate sufficient memory to accept the entire path string, and set *thePath* as a pointer to that memory. *column* must currently be loaded in the NXBrowser. If *column* isn't loaded or *thePath* is a null pointer, this method returns NULL.

The path is constructed by concatenating the string values in the selected cells in each column, preceding each with the pathSeparator. For example, consider a pathSeparator "@" and an NXBrowser with two columns. If the selected cell in the left column has the string value "foo" and the selected cell in the right column has the string value "bar," the resulting path is "@foo@bar." The default pathSeparator is the slash character ("/").

See also: – **pathSeparator**, – **setPath:**, – **setPathSeparator:**

getTitleFrame:ofColumn:

– (NXRect *)**getTitleFrame:**(NXRect *)*theRect* **ofColumn:**(int)*column*

Returns *theRect*, a pointer to the rectangle (in NXBrowser coordinates) enclosing the title of column number *column*. If the NXBrowser isn't displaying titles or the specified column isn't loaded, returns NULL.

getTitleFromPreviousColumn:

– **getTitleFromPreviousColumn:**(BOOL)*flag*

If *flag* is YES, sets the NXBrowser so that each column takes its title from the string value in the selected cell in the column to its left, leaving column 0 untitled; use **setTitle:ofColumn:** to give column 0 a title. This method affects the receiver only when it is titled (**isTitled** returns YES).

By default, the NXBrowser is set to get column titles from the previous column. Send this message with NO as the argument if your delegate implements the **browser:titleOfColumn:** method or if you use the **setTitle:ofColumn:** method to set all column titles. Returns **self**.

See also: – **isTitled**, – **setTitle:ofColumn:**, – **setTitled:**, – **browser:titleOfColumn:** in "METHODS IMPLEMENTED BY THE DELEGATE"

hideLeftAndRightScrollButtons:

– **hideLeftAndRightScrollButtons:**(BOOL)*flag*

If *flag* is YES, sets the NXBrowser to hide left and right scroll buttons. Generally, you shouldn't hide left and right scroll buttons unless your data is nonhierarchical, thus limited to a single column, or restricted so that the NXBrowser will always display enough columns for all data. Returns **self**.

initFrame

– **initFrame:**(const NXRect *)*frameRect*

Initializes a new instance of NXBrowser with a bounds of *frameRect*. The initialized NXBrowser is set to have column titles, to get titles from previous columns, and to use scrollbars. The minimum column width is set to 100 and the path separator is set to the slash ("/") character. The NXBrowser is set not to clip. This method invokes the tile method to arrange the components of the NXBrowser (titles, scroll bars, matrices, and so on).

isLoaded

– (BOOL)**isLoaded**

Returns YES if any of the NXBrowser's columns are loaded.

See also: **loadColumnZero**

isTitled

– (BOOL)**isTitled**

Returns YES if the NXBrowser's columns are displayed with titles above them; NO otherwise.

See also: – **getTitleFromPreviousColumn:**, – **setTitled:**

keyDown

– **keyDown:**(NXEvent *)*theEvent*

Handles arrow key events. This method is invoked when the NXBrowser or one of its subviews is the first responder. If the NXBrowser has been set to accept arrow keys, and the key represented in *theEvent* is an arrow key, this method scrolls through the NXBrowser in the direction indicated.

See also: – **acceptArrowKeys:**, – **acceptsFirstResponder**

lastVisibleColumn

– (int)**lastVisibleColumn**

Returns the index of the rightmost visible column. This may be less than the value returned by **lastColumn** if the NXBrowser has been scrolled left.

See also: – **firstVisibleColumn**, – **lastColumn**

lastColumn

– (int)**lastColumn**

Returns the index of the last column in the NXBrowser.

loadColumnZero

– **loadColumnZero**

Loads and displays data in column 0 of the NXBrowser, unloading any columns to the right that were previously loaded. Invoke this method to force the NXBrowser to be loaded. You may want to override this method if you subclass NXBrowser.

See also: – **addColumn**, – **reloadColumn:**

matrixInColumn:

– **matrixInColumn:**(int)*column*

Returns the matrix found in column number *column*. Returns **nil** if column number *column* isn't loaded in the NXBrowser.

maxVisibleColumns

– (int)**maxVisibleColumns**

Returns the maximum number of visible columns allowed. No matter how many loaded columns the NXBrowser contains, or how large the NXBrowser is made (for example, by resizing its window), it will never display more than this number of columns. If the number of loaded columns can exceed the value returned by this method, the NXBrowser must display left and right scroll buttons.

See also: **– hideLeftAndRightScrollButtons, – setMaxVisibleColumns**

minColumnWidth

– (int)**minColumnWidth**

Returns the minimum width of a column in PostScript points (rounded to the nearest integer). No column will be smaller than the returned value unless the NXBrowser itself is smaller than that. The default setting is 100 points.

See also: – **setMinColumnWidth**

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

Handles a mouse down in the NXBrowser's left or right scroll buttons. Returns **self**.

numVisibleColumns

– (int)**numVisibleColumns**

Returns the number of columns which can be visible at the same time in the NXBrowser (that is, the current width, in columns, of the NXBrowser). This may be less than the value returned by **maxVisibleColumns** if the window containing the NXBrowser has been resized.

See also: – **setMaxVisibleColumns**

reflectScroll:

– **reflectScroll:***clipView*

This method updates scroll bars in the column containing *clipView*. Scroll bars are enabled if a column contains more data than can be displayed at once and disabled if the column can display all data. Returns **self**.

See also: – **useScrollBars**

reloadColumn:

– **reloadColumn:**(int)*column*

Reloads column number *column* by sending a message to the delegate to update the Cells in its Matrix, then reselecting the previously selected Cell if it's still in the Matrix. Redraws the column and returns **self**.

reuseColumns:

– **reuseColumns:**(BOOL)*flag*

Sets whether the NXBrowser saves a column's Matrix and ClipView or ScrollView when the column is removed, and whether it then reuses these subviews when the column is reloaded. If *flag* is YES, the NXBrowser reuses columns for somewhat faster display of columns as they are reloaded. If *flag* is NO, the NXBrowser frees columns as they're unloaded, reducing average memory use. Returns **self**.

scrollColumnsLeftBy:

– **scrollColumnsLeftBy:**(int)*shiftAmount*

Scrolls the NXBrowser left (toward the first column) by *shiftAmount* columns. If *shiftAmount* exceeds the number of columns to the left of the first visible column, then the NXBrowser scrolls left until the column 0 is visible. Redraws and returns **self**.

See also: – **scrollColumnsRightBy:**

scrollColumnsRightBy:

– **scrollColumnsRightBy:**(int)*shiftAmount*

Scrolls the NXBrowser right (toward the last column) by *shiftAmount* columns. If *shiftAmount* exceeds the number of loaded columns to the right of the first visible column, then the NXBrowser scrolls right until the last loaded column is visible. Redraws and returns **self**.

See also: – **scrollColumnsLeftBy:**

scrollColumnToVisible:

– **scrollColumnToVisible:**(int)*column*

Scrolls the NXBrowser to make column number *column* visible. If there's no *column* in the NXBrowser, this method scrolls to the right as far as possible. Redraws and returns **self**.

scrollUpOrDown:

– **scrollUpOrDown:***sender*

Scrolls a column up or down. You don't send this message; NXBrowser receives it from a column's scroll buttons. Returns **self**.

selectedColumn

– (int)**selectedColumn**

Returns the column number of the rightmost column containing a selected cell. Returns –1 if no column in the NXBrowser contains a selected cell.

selectAll

– **selectAll:***sender*

Selects all entries in the last column loaded in the NXBrowser if multiple selection is allowed. Returns **self**.

See also: – **allowMultiSel:**

separateColumns:

– **separateColumns:**(BOOL)*flag*

If *flag* is YES, sets NXBrowser so that columns have bezeled borders separating them; if NO, the borders are removed. When titles are set to display (by **setTitled:**), columns are automatically separated; however, the flag set by this method is unchanged. Redraws the NXBrowser and returns **self**.

See also: – **setTitled:**

setAction:

– **setAction:**(SEL)*aSelector*

Sets the action of the NXBrowser. *aSelector* is the selector for the message sent to the NXBrowser's target when a mouse-down event occurs in a column of the NXBrowser. Returns **self**.

See also: – **action**, – **doubleAction**, – **doClick**, – **doDoubleClick**, – **setTarget**, – **target**

setCellClass:

– **setCellClass:***classId*

Sets the class of Cell used when adding Cells to a Matrix in a column of the NXBrowser. *classId* must be the value returned when sending the **class** message to NXBrowserCell or a subclass of NXBrowserCell. Returns **self**.

See also: – **cellClass**, – **setCellPrototype**

setCellPrototype:

– **setCellPrototype:***aCell*

Sets *aCell* as the Cell prototype copied when adding Cells to the Matrices in the columns of NXBrowser. *aCell* must be an instance of NXBrowserCell or a subclass of NXBrowserCell. Returns **self**.

See also: – **cellPrototype**

setDelegate:

– **setDelegate:***anObject*

Sets the delegate of the NXBrowser to *anObject* and returns **self**. If *anObject* is of a class that implements the **browser:fillMatrix:inColumn:** method (normal or lazy delegates) or the **browser:loadCell:atRow:inColumn** and **browser:getNumRowsInColumn:** methods (very lazy delegate), it's set as the NXBrowser's delegate; otherwise, the delegate is set to **nil**. Returns **self**.

See also: – **delegate**, "METHODS IMPLEMENTED BY THE DELEGATE"

setDoubleAction:

– **setDoubleAction:**(SEL)*aSelector*

Sets the double action of the NXBrowser. *aSelector* is the selector for the action message sent to the target when a double-click occurs in one of the columns of the NXBrowser. Returns **self**.

setEnabled:

– **setEnabled:**(BOOL)*flag*

Enables the NXBrowser when *flag* is YES; disables it when *flag* is NO. Returns **self**.

setLastColumn:

– **setLastColumn:**(int)*column*

Sets the last column loaded in and displayed by the NXBrowser. Removes any columns to the right of *column* from the NXBrowser. Scrolls columns in the NXBrowser to make the new last column visible if it wasn't previously. If *column* is to the right of the last column in the NXBrowser, this method does nothing. Returns **self**.

setMatrixClass:

– **setMatrixClass:***classId*

Sets the matrixClass instance variable, representing the class used when adding new columns to the NXBrowser. *classId* must be the value returned by sending the **class** message to Matrix or a subclass of Matrix; otherwise this method retains the previous setting for matrixClass. Returns **self**.

setMaxVisibleColumns:

– **setMaxVisibleColumns:**(int)*columnCount*

Sets the maximum number of columns that may be displayed by the NXBrowser. Returns **self**.

To set the number of columns displayed in a new NXBrowser, first send it a **setMinColumnWidth:** message with a small argument (1 for example) to ensure that the desired number of columns will fit in the NXBrowser's frame. Then invoke this method to set the number of columns you want your NXBrowser to display.

See also: **– maxVisibleColumns**, **– setMinColumnWidth:**

setMinColumnWidth:

– **setMinColumnWidth:**(int)*columnWidth*

Sets the minimum width for each column to *columnWidth* and redisplays the NXBrowser with columns set to the new width. *columnWidth* is measured in PostScript points rounded to the nearest integer. The default setting is 100. Returns **self**.

See also: **– minColumnWidth**

setPath:

– **setPath:**(const char *)*path*

Parses *aPath*—a string consisting of one or more substrings separated by the path separator—and selects column entries in the NXBrowser that match the substrings. If the first character in *aPath* is the path separator, this method begins searching for matches in column 0; otherwise, it begins searching in the last column loaded. If no column is loaded, this method loads column 0 and begins the search there. While parsing the current substring, it tries to locate a matching entry in the search column. If it finds an exact match, this method selects that entry and moves to the next column (loading the column if necessary) to search for the next substring.

If this method finds a valid path (one in which each substring is matched by an entry in the corresponding column), it returns **self**. If it doesn't find an exact match on a substring, it stops parsing *aPath* and returns **nil**; however, column entries that it has already selected remain selected.

See also: – **getPath:toColumn**, – **pathSeparator**, – **setPathSeparator**

setPathSeparator:

– **setPathSeparator:**(unsigned short)*charCode*

Sets the character used as the path separator; the default is the slash character ("/"). Returns **self**.

See also: – **getPath:toColumn**, – **pathSeparator**, – **setPath:**

setTarget:

– **setTarget:***anObject*

Sets the target of the NXBrowser. Returns **self**.

setTitle:ofColumn:

– **setTitle:**(const char *)*aString* **ofColumn:**(int)*column*

Sets the title of column number *column* in the NXBrowser to *aString*. Returns **self**.

See also: – **browser:TitleOfColumn:** in "METHODS IMPLEMENTED BY THE DELEGATE," – **getTitleFromPreviousColumn:**, and – **setTitled:**

setTitled:

– **setTitled:**(BOOL)*flag*

If *flag* is YES, columns display titles and are separated by bezeled borders. Returns **self**.

See also: – **browser:TitleOfColumn:** in "METHODS IMPLEMENTED BY THE DELEGATE," – **getTitleFromPreviousColumn:**, and – **setTitle:ofColumn:**

sizeTo::

– **sizeTo:**(NXCoord)*width* **:**(NXCoord)*height*

Resizes the NXBrowser to the new *width* and *height*. Usually sent by the window. Returns **self**.

sizeToFit

– **sizeToFit**

Resizes the NXBrowser to contain all the columns and controls displayed in it. Returns **self**.

target

– **target**

Returns the target for the NXBrowser's action message.

See also: – **action**, – **doClick**, – **doDoubleClick**, – **doubleAction**, – **setAction**, – **setDoubleAction**, – **setTarget:**

tile

– **tile**

Arranges the various subviews of NXBrowser—scrollers, columns, titles, and so on—without redrawing. You shouldn't send this message. Rather, it's invoked any time the appearance of the NXBrowser changes; for example, when scroll buttons or scroll bars are set, a column is added, and so on. Override this method if you change the appearance of the NXBrowser, for example, if you draw your own titles above columns. Returns **self**.

titleHeight

– (NXCoord)**titleHeight**

Returns the height of titles drawn above the columns of the NXBrowser. Override this method if you display your own titles above the NXBrowser's columns.

titleOfColumn:

– (const char *)**titleOfColumn:**(int)*column*

Returns a pointer to the title string displayed above column number *column*. If no such column is loaded in the NXBrowser, returns NULL.

useScrollBars:

– **useScrollBars:**(BOOL)*flag*

If *flag* is YES, sets NXBrowser to use scroll bars for its columns. By default, NXBrowser does use scroll bars. Redraws and returns **self**.

See also: – **useScrollButtons**

useScrollButtons:

– **useScrollButtons:**(BOOL)*flag*

If *flag* is YES, sets the NXBrowser to use scroll buttons for its columns. When the NXBrowser is also set to use scroll bars, this method causes scroll buttons to display at the base of the scroll bars. Redraws and returns **self**.

See also: – **useScrollBars**

validateVisibleColumns

– **validateVisibleColumns**

Validates the columns visible in the NXBrowser by invoking the delegate method **browser:columnIsValid:** for all visible columns. Use this method to confirm that the entries displayed in each visible column are valid before redrawing.

See also: **browser:columnIsValid** in "METHODS IMPLEMENTED BY THE DELEGATE"

METHODS IMPLEMENTED BY THE DELEGATE

browser:columnIsValid:

– (BOOL)**browser:***sender* **columnIsValid:**(int)*column*

This method is invoked by NXBrowser's **validateVisibleColumns** method to determine whether the contents currently loaded in column number *column* need to be updated. Returns YES if the contents are valid; NO otherwise.

browserDidScroll:

– **browserDidScroll:***sender*

Notifies the delegate when the browser has finished scrolling. Returns **self**.

browser:fillMatrix:inColumn:

– (int)**browser:***sender*
  fillMatrix:*matrix*
  inColumn:(int)*column*

Invoked by the NXBrowser to query a normal or lazy browser for the contents of *column*. This method should create NXBrowserCells by sending **addRow** or **insertRowAt:** messages to *matrix*. A normal delegate should then load each new NXBrowserCell and send them the messages **setLoaded:** and **setLeaf:**. A lazy delegate loads Cells only when they are about to be displayed. This method returns the number of entries in *column*.

If you implement this method, don't implement the delegate method **browser:getNumRowsInColumn:**.

browser:getNumRowsInColumn:

– (int)**browser:**sender **getNumRowsInColumn:**(int)*column*

Implemented by very-lazy delegates, this method is invoked by the NXBrowser to ask the delegate for the number of rows in column number *column*. This method allows the NXBrowser to resize its scroll bar for a column, without loading all the cells in that column. Returns the number of rows in *column*.

If you implement this method, don't implement the delegate method **browser:fillMatrix:inColumn:**.

browser:loadCell:atRow:inColumn:

– **browser:***sender*
  loadCell:*cell*
  atRow:(int)*row*
  inColumn:(int)*column*

Implemented by lazy and very-lazy delegates. This method loads the entry in *cell* in the specified *row* and *column* in the NXBrowser. This method should send **setLoaded:** and **setLeaf:** messages to *cell*. Returns **self** (the **id** of the delegate).

browser:selectCell:inColumn:

– (BOOL)**browser:***sender*
  selectCell:(const char *)*entry*
  inColumn:(int)*column*

Asks NXBrowser's delegate to validate and select an entry in column number *column*. This method should load *entry* if necessary and send it **setLoaded:** and **setLeaf:** messages to indicate its state. Returns YES if the method successfully selects *entry* in *column*; NO otherwise.

browser:titleOfColumn:

– (const char *)**browser:***sender* **titleOfColumn:**(int)*column*

Invoked by NXBrowser to get the title for *column* from the delegate. This method is invoked only when the NXBrowser is titled and has received a **getTitleFromPreviousColumn:** message with NO as the argument. By default, the NXBrowser makes each column title the string value of the selected cell in the previous column. Returns the string representing the title belonging above *column*.

See also: – **getTitledFromPreviousColumn:**, – **setTitle:ofColumn:**, – **setTitled:**

browserWillScroll:

– **browserWillScroll:***sender*

This method notifies the delegate when the browser is about to scroll. Returns **self**.

NXBrowserCell

INHERITS FROM	Cell : Object
DECLARED IN	appkit/NXBrowserCell.h

CLASS DESCRIPTION

NXBrowserCell is the subclass of Cell used to display data in the column Matrices of NXBrowser. Many of NXBrowserCell's methods are designed to interact with NXBrowser and NXBrowser's delegate. The delegate implements methods for loading the Cells in NXBrowser by setting their values and status. If you need access to a specific NXBrowserCell, you can use the NXBrowser method **getLoadedCellAtRow:inColumn:**.

You may find it useful to subclass NXBrowserCell to alter its behavior and to enable it to work with and display the type of data you wish to represent. Use NXBrowser's **setCellClass:** or **setCellPrototype:** methods to use your subclass.

See also: NXBrowser

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Cell	char	*contents;
	id	support;
	struct _cFlags1	cFlags1;
	struct _cFlags2	cFlags2;

METHOD TYPES

Creating an NXBrowserCell	– init
	– initTextCell:
Determining icons	+ branchIcon
	+ branchIconH
Determining component sizes	– calcCellSize:inRect:
Displaying	– drawInside:inView:
	– drawSelf:inView:
Highlighting behavior	– highlight:inView:lit:

Placing in browser hierarchy	– isLeaf – setLeaf:
Determining loaded status	– isLoaded – setLoaded:
Determining reset status	– reset
Modifying graphic attributes	– isOpaque

CLASS METHODS

branchIcon

+ **branchIcon**

Returns the **id** of the NXImage object "NXMenuArrow." This is the icon displayed to indicate a branch node in an NXBrowserCell. Override this method if you want to display a different branch icon.

See also: – **isBranch**, – **setBranch**

branchIconH

+ **branchIconH**

Returns the **id** of the NXImage object "NXMenuArrowH." This is the highlighted icon displayed to indicate a branch node in an NXBrowserCell. Override this method if you want to display a different branch icon.

See also: – **isBranch**, – **setBranch**

INSTANCE METHODS

calcCellSize:inRect:

– **calcCellSize:**(NXSize *)*theSize* **inRect:**(const NXRect *)*aRect*

Calculates the minimum width and height required for displaying the NXBrowserCell in a given rectangle. Makes sure *theSize* remains large enough to accommodate the branch arrow icon. If it isn't possible for the NXBrowserCell to fit in *aRect*, the width and/or height returned in *theSize* could be bigger than those of the rectangle. The computation is done by trying to size the NXBrowserCell so that it fits in the rectangle argument (by wrapping the text, for instance). If a choice must be made between extending the width or height of *aRect* to fit the text, the height will be extended. Returns **self** and, by reference, the minimum size for the NXBrowserCell.

drawInside:inView:

– **drawInside:**(const NXRect *)*cellFrame* **inView:***controlView*

Draws the inside of the NXBrowserCell (that is, it's the same as **drawSelf:inView:** except that it doesn't draw the bezel or border if there is one). Returns **self**.

drawSelf:inView:

– **drawSelf:**(const NXRect *)*cellFrame* **inView:***controlView*

Draws the NXBrowserCell, including the bezel or border. Returns **self**.

See also: – **drawInside:inView:**

highlight:inView:lit:

– **highlight:**(const NXRect *)*cellFrame* **inView:***controlView* **lit:**(BOOL)lit

Sets the highlighted state to *lit* and redraws the NXBrowserCell. Returns **self**.

See also: – **reset**

init

– **init**

Initializes and returns the receiver, a new NXBrowserCell instance, by invoking the **initTextCell:** method. Sets the NXBrowserCell's string value to "BrowserItem" and returns **self**.

initTextCell:

– **initTextCell:**(const char *)*aString*

Initializes the receiver, a new NXBrowserCell instance, by sending the message [**super initTextCell:***aString*]. Sets the NXBrowserCell so it doesn't wrap text. Returns **self**. This method is the designated initializer for the NXBrowserCell class. Override this method if you create a subclass of NXBrowserCell that performs its own initialization.

isLeaf

– (BOOL)**isLeaf**

Determines whether the entry in the receiver represents a leaf node (such as a file) or branch node (such as a directory). This method is invoked by NXBrowser to check whether to display the branch icon in the Cell and, when an NXBrowserCell is selected, whether to load a column to the right of the column containing the receiving Cell. Returns YES if the cell represents a leaf, NO if the cell represents a branch.

See also: – **setLeaf:**

isLoaded

– (BOOL)**isLoaded**

Returns YES if the NXBrowserCell is loaded, NO if it isn't. Used by NXBrowser to determine if a particular Cell is loaded in a column. When an NXBrowserCell is created, this value is YES. NXBrowser and its delegate change the value returned by this method using the **setLoaded:** method to reflect the current status of the cell.

See also: – **setLoaded:**

isOpaque

– (BOOL)**isOpaque**

Returns YES if the NXBrowserCell is opaque (that is, it touches every pixel in its bounds).

reset

– **reset**

Sets the NXBrowserCell's state to 0, sets the highlighted flag to NO, and returns **self**.

See also: – **highlight:inView:lit**

setLeaf:

– **setLeaf:**(BOOL)*flag*

Invoked by NXBrowser's delegate when it loads an NXBrowserCell. When *flag* is YES, the NXBrowserCell represents a leaf node; it will display without the branch icon. When *flag* is NO, the NXBrowserCell represents a branch node; it will display with the branch icon.

See also: – **branchIcon**, – **branchIconH**, – **isLeaf**

setLoaded:

– **setLoaded:**(BOOL)*flag*

Sets the loaded status of the NXBrowser cell to *flag*. This method is invoked by NXBrowser or its delegate to set the status of the NXBrowserCell. The delegate should send the **setLoaded:** message with YES as the argument when it loads the cell.

See also: – **isLoaded**, "METHODS IMPLEMENTED BY THE DELEGATE" (NXBrowser)

NXCachedImageRep

INHERITS FROM	NXImageRep : Object
DECLARED IN	appkit/NXCachedImageRep.h

CLASS DESCRIPTION

An NXCachedImageRep is a rendered image in a window, typically a window that stays off-screen. The only data that's available for reproducing the image is the image itself. Thus an NXCachedImageRep differs from the other kinds of NXImageReps defined in the Application Kit, all of which can reproduce an image from the information originally used to draw it.

Instances of this class are generally used indirectly, through an NXImage object. An NXCachedImageRep must be able to provide the NXImage with some information about the image—so that the NXImage can match it to a display device, for example, or know whether to scale it. Therefore, it's a good idea to use these inherited methods to inform the NXCachedImageRep object about the image in the cache:

setNumColors:
setAlpha:
setPixelsHigh:
setPixelsWide:
setBitsPerSample:

These methods are all defined in the NXImageRep class.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from NXImageRep	NXSize	size;
Declared in NXCachedImageRep	(none)	

METHOD TYPES

Initializing a new NXCachedImageRep	– initFromWindow:rect:
Freeing an NXCachedImageRep	– free
Getting the representation	– getWindow:andRect:
Drawing the image	– draw
Archiving	– read: – write:

INSTANCE METHODS

draw

– (BOOL)**draw**

Reads image data from the cache and reproduces the image from that data. The reproduction is rendered in the current window at location (0.0, 0.0) in the current coordinate system.

It's much more efficient to reproduce an image by compositing it, which can be done through the NXImage class. An NXBitmapImageRep can also be used to reproduce an existing image.

This method returns YES if successful in reproducing the image, and NO if not.

See also: – **drawIn:** (NXImageRep), – **drawAt:** (NXImageRep), – **initData:fromRect:** (NXBitmapImageRep)

free

– **free**

Deallocates the NXCachedImageRep.

getWindow:andRect:

– **getWindow:**(Window **)*theWindow* **andRect:**(NXRect *)*theRect*

Copies the **id** of the Window object where the image is located into the variable referred to by *theWindow*, and copies the rectangle that bounds the image into the structure referred to by *theRect*. If *theRect* is NULL, only the Window **id** is provided. Returns **self**.

init

Generates an error message. This method cannot be used to initialize an NXCachedImageRep. Use the **initFromWindow:rect:** method instead.

See also: – **initFromWindow:rect:**

initFromWindow:rect:

– **initFromWindow:**(Window *)*aWindow* **rect:**(const NXRect *)*aRect*

Initializes the receiver, a new NXCachedImageRep instance, for an image that will be rendered within the *aRect* rectangle in *aWindow*, and returns the initialized object. The rectangle is specified in *aWindow*'s base coordinate system. The size of the image is set from the size of the rectangle.

You must draw the image in the rectangle yourself; there are no NXCachedImageRep methods for this purpose.

read:

– **read:**(NXTypedStream *)*stream*

Reads the NXCachedImageRep from the typed stream *stream*.

write:

– **write:**(NXTypedStream *)*stream*

Writes the NXCachedImageRep to the typed stream *stream*.

NXColorPanel

INHERITS FROM	Panel : Window : Responder : Object
DECLARED IN	appkit/NXColorPanel.h

CLASS DESCRIPTION

NXColorPanel provides a standard user interface for selecting color in an application. It provides seven color selection modes, including four that correspond to industry-standard color models. It allows the user to set swatches containing frequently used colors. Once set, these swatches are displayed by NXColorPanel in any application where it is used, giving the user color consistency between applications. The NXColorPanel also enables the user to capture a color anywhere on the screen for use in the active application, and to drag colors between views in an application.

The color mask determines which of the color modes are enabled for NXColorPanel. This mask is set by using color mask constants when you initialize a new instance of NXColorPanel. When an instance of NXColorPanel is masked for more than one color mode, your program can set its mode by invoking the **setMode:** method with a color mode constant as its argument; the user can set the mode by clicking buttons on the panel. Here are the color modes with corresponding mask and mode constants:

Mode	**Color Mask/Color Mode Constants**
Grayscale-Alpha	NX_GRAYMODEMASK NX_GRAYMODE
Red-Green-Blue	NX_RGBMODEMASK NX_RGBMODE
Cyan-Yellow-Magenta-Black	NX_CMYKMODEMASK NX_CMYKMODE
Hue-Saturation-Brightness	NX_HSBMODEMASK NX_HSBMODE
TIFF image	NX_CUSTOMPALETTEMODEMASK NX_CUSTOMPALETTEMODE
Custom color lists	NX_CUSTOMCOLORMODEMASK NX_CUSTOMCOLORMODE
Color wheel	NX_BEGINMODEMASK NX_BEGINMODE
All of the above	NX_ALLMODESMASK none

NX_ALLMODESMASK represents the logical OR of the other color mask constants. When NXColorPanel is initialized using NX_ALLMODESMASK, it can be set to any of the modes. When initializing a new instance of NXColorPanel, you can logically OR any combination of color mask constants to restrict the available color modes.

In grayscale-alpha, red-green-blue, cyan-magenta-yellow-black, and hue-saturation-brightness modes, the user adjusts colors by manipulating sliders. In the custom palette mode, the user can load a TIFF file into the NXColorPanel, then select colors from the TIFF image. In custom color list mode, the user can create and load lists of named colors. The two custom modes provide PopUpLists for loading and saving files. Finally, color wheel mode provides a simplified control for selecting colors; by default, it's the initial mode when the NX_ALLMODESMASK constant is used to initialize the NXColorPanel.

NXColorPanel's action message is sent to the target object when the user changes the current color.

An application has only one instance of NXColorPanel, the shared instance. Once the shared instance has been created, invoking any of the **new** methods returns the shared instance rather than a new NXColorPanel.

One of NXColorPanel's methods, **dragColor:withEvent:fromView:**, allows colors to be moved between Views in an application. For example, NXColorWell invokes this method from its **mouseDown:** method to allow you to move colors from a well to other views. Any View can implement the **acceptColor:atPoint:** method to accept a color dragged from an NXColorWell or NXColorPanel.

You can put NXColorPanel in any application created with Interface Builder by adding the "Colors..." item from the Menu palette to the application's menu.

See also: NXColorWell

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;

Inherited from Window	NXRect	frame;
	id	contentView;
	id	delegate;
	id	firstResponder;
	id	lastLeftHit;
	id	lastRightHit;
	id	counterpart;
	id	fieldEditor;
	int	winEventMask;
	int	windowNum;
	float	backgroundGray;
	struct _wFlags	wFlags;
	struct _wFlags2	wFlags2;
Inherited from Panel	(none)	
Declared in NXColorPanel	(none)	

METHOD TYPES

Creating a New NXColorPanel	+ newContent:style:backing:buttonMask:defer:
	+ newContent:style:backing:buttonMask:defer: colorMask:
	+ newMask:
	+ sharedInstance:
Setting NXColorPanel Behavior	– colorMask
	– setColorMask:
	– setContinuous:
	– setMode:
	– setAccessoryView:
	– setShowAlpha:
Setting Color	+ dragColor:withEvent:fromView:
	– color
	– setColor:
	– updateCustomColorList
Target and Action	– setAction:
	– setTarget:

CLASS METHODS

dragColor:withEvent:fromView:

+ (BOOL)**dragColor:**(NXColor)*color*
  withEvent:(NXEvent*)*theEvent*
  fromView:*controlView*

Allows colors to be dragged between views in an application. This method is usually invoked by the **mouseDown:** method of *controlView*; the **mouseDown:** method sets up a modal loop until the subsequent NX_MOUSEUP event occurs, then sends this message to the NXColorPanel class object. *theEvent* is always the NX_MOUSEUP event; this method uses the cursor coordinates from *theEvent* to determine the receiving View.

To accept the dragged color, the receiving view must implement the method **acceptColor:**(NXColor)*color* **atPoint:**(NXPoint)*mouseUpPoint*. The only View subclass in the application kit that implements this method is NXColorWell. Implementing **acceptColor:atPoint:** in a View subclass is described in "METHODS IMPLEMENTED BY A VIEW SUBCLASS" at the end of this section.

Because it is a class method, **dragColor:withEvent:fromView:** can be used whether or not an instance of NXColorPanel exists. Returns YES.

See also: **– mouseDown:** (NXColorWell), – **acceptColor:atPoint:** in "METHODS IMPLEMENTED BY A VIEW SUBCLASS" below and in NXColorWell

newContent:style:backing:buttonMask:defer:

+ **newContent:**(const NXRect *)*contentRect*
  style:(int)*aStyle*
  backing:(int)*bufferingType*
  buttonMask:(int)*mask*
  defer:(BOOL)*flag*

Invokes the **newContent:style:backing:buttonMask:defer:colorMask:** method with NX_ALLMODESMASK as the argument. This method is implemented to override the method inherited from the Panel class.

See also: + **newContent:style:backing:buttonMask:defer:colorMask:**

newContent:style:backing:buttonMask:defer:colorMask:

+ **newContent:**(const NXRect *)*contentRect*
 style:(int)*aStyle*
 backing:(int)*bufferingType*
 buttonMask:(int)*mask*
 defer:(BOOL)*flag*
 colorMask:(int)*colormask*

Creates, if necessary, and returns the shared instance of NXColorPanel. Only one instance of NXColorPanel can be created in an application. This method allocates a new instance of NXColorPanel from its own zone, then initializes it by invoking the **initContent:style:backing:buttonMask:defer:colorMask:** method. The **newColorMask:** method below lists the constants to use for *colormask*.

See also: + **newColorMask:**

newColorMask:

+ **newMask:**(int)*colormask*

Creates, if necessary, and returns the shared instance of the NXColorPanel. Only one instance of NXColorPanel can be created in an application. This method allocates a new instance of NXColorPanel from its own zone, then initializes it by invoking the **initColorMask:** method.

To set the color selection modes available in a new instance of NXColorPanel, use one of the following constants for *colormask*:

NX_GRAYMODEMASK
NX_RGBMODEMASK
NX_CMYKMODEMASK
NX_HSBMODEMASK
NX_CUSTOMPALETTEMODEMASK
NX_CUSTOMCOLORMODEMASK
NX_BEGINMODEMASK
NX_ALLMODESMASK

To enable multiple selection modes for the new NXColorPanel, use a *colormask* expression containing the logical OR of two or more color mask constants. NX_ALLMODESMASK represents the logical OR of all the other masks.

To change the color selection modes available in an existing instance of NXColorPanel, use the **setColorMask** method.

See also: **– colorMask**, – **setColorMask**, – **setMode**

sharedInstance:

+ **sharedInstance:**(BOOL)*create*

Tests for the shared instance of NXColorPanel. If *create* is NO and the shared instance exists, this method returns its id; if no instance of NXColorPanel exists, returns **nil**. If *create* is YES, this method creates, if necessary, and returns the **id** of the shared NXColorPanel.

INSTANCE METHODS

alloc

Generates an error message. This method cannot be used to create NXColorPanel instances. Use the **newFrame:** method instead.

See also: + **newFrame:**

allocFromZone

Generates an error message. This method cannot be used to create NXColorPanel instances. Use the **newFrame:** method instead.

See also: + **newFrame:**

color

– (NXColor)**color**

Returns the current color selection of the NXColorPanel.

See also: – s**etColor**

colorMask

– (int)**colorMask**

Returns the color mask. The return value will be one of the color mask constants described in the **newMask:** method or a logical OR of two or more of the constants.

See also: + **newMask:**

setAccessoryView:

– **setAccessoryView:***aView*

Sets the accessory view displayed in the NXColorPanel to *aView*. The accessory View can be any custom View that you want to display with NXColorPanel, for example, a View offering patterns or brush shapes in a drawing program. The accessory View is displayed below the regular controls in the NXColorPanel. The NXColorPanel automatically resizes to accommodate the accessory View. Returns **self**.

setAction:

– **setAction:**(SEL)*aSelector*

Sets the action of the NXColorPanel to *aSelector*. Returns **self**.

setColor:

– **setColor:**(NXColor)*color*

Sets the color setting of the NXColorPanel to *color* and redraws the panel. Returns **self**.

setColorMask:

– **setColorMask:**(int)*colormask*

Sets the color mode mask of the NXColorPanel. Returns **self**.

setContinuous:

– **setContinuous:**(BOOL)*flag*

Sets the NXColorPanel to send the action message to its target continuously as the color of the NXColorPanel is set by the user. Send this message with *flag* YES if, for example, you want tot continuously update the color of the target. Returns **self**.

setMode:

– **setMode:**(int)*mode*

Sets the mode of the NXColorPanel if *mode* is one of the modes allowed by the color mask. The color mask is set when you first create the shared instance of NXColorPanel for an application. *mode* can be one of seven constants:

NX_GRAYMODE
NX_RGBMODE
NX_CMYKMODE
NX_HSBMODE
NX_CUSTOMPALETTEMODE
NX_CUSTOMCOLORMODE
NX_BEGINMODE

Color modes and masks are described in more detail in "CLASS DESCRIPTION" at the beginning of this discussion.

Returns **self**.

See also: "CLASS DESCRIPTION"

setShowAlpha:

– **setShowAlpha:**(BOOL)*flag*

If *flag* is YES, sets the NXColorPanel to show alpha. Returns **self**.

setTarget:

– **setTarget:***anObject*

Sets the target of the NXColorPanel. The NXColorPanel's target is the object to which the action message is sent when the user selects a color. Returns **self**.

See also: **– setAction**, **– setContinuous**

updateCustomColorList

– **updateCustomColorList**

Updates the custom color list to reflect the current entries. This method is invoked by Controls on the NXColorPanel in NX_CUSTOMCOLORMODE.

METHODS IMPLEMENTED BY A VIEW SUBCLASS

acceptColor:atPoint:

– **acceptColor:**(NXColor)*color* **atPoint:**(NXPoint *)*aPoint*

Allows the View to accept *color* when *aPoint* is a point within its bounds. Implement this method if you want to be able to drag a color from an NXColorPanel or NXColorWell into your subclass of View. This method is invoked by NXColorPanel's class method **dragColor:withEvent:fromView:**. Returns **self**.

See also: **– acceptColor:atPoint:** in NXColorWell, **+ dragColor:withEvent:fromView:**

CONSTANTS

```
#define NX_GRAYMODE              0
#define NX_RGBMODE               1
#define NX_CMYKMODE              2
#define NX_HSBMODE               3
#define NX_CUSTOMPALETTEMODE     4
#define NX_CUSTOMCOLORMODE       5
#define NX_BEGINMODE             6

#define NX_GRAYMODEMASK          1
#define NX_RGBMODEMASK           2
#define NX_CMYKMODEMASK          4
#define NX_HSBMODEMASK           8
#define NX_CUSTOMPALETTEMODEMASK 16
#define NX_CUSTOMCOLORMODEMASK   32
#define NX_BEGINMODEMASK         64

#define NX_ALLMODESMASK \
    (NX_GRAYMODEMASK| \
    NX_RGBMODEMASK| \
    NX_CMYKMODEMASK| \
    NX_HSBMODEMASK| \
    NX_CUSTOMPALETTEMODEMASK| \
    NX_CUSTOMCOLORMODEMASK| \
    NX_BEGINMODEMASK)
```

NXColorWell

INHERITS FROM	Control: View: Responder: Object
DECLARED IN	appkit/NXColorWell.h

CLASS DESCRIPTION

NXColorWell is a Control for selecting and displaying a single color value. An example of NXColorWell is found in NXColorPanel, which uses a well to display the current color selection. NXColorWell is available from the Palettes panel of Interface Builder.

An application can have one or more active NXColorWells. You can activate multiple NXColorWells by invoking the **activate:** method with NO as its argument. You can set the same color for all active color wells by invoking the class method **activeWellsTakeColorFrom:**. You can deactivate multiple wells using the class method **deactivateAllWells**. When a mouse-down event occurs in an NXColorWell, it becomes the only active well.

The **mouseDown:** method enables an instance of NXColorWell to send its color to another NXColorWell or any other subclass of View that implements the **acceptColor:atPoint:** method. NXColorWell's **mouseDown:** method invokes NXColorPanel's **dragColor:withEvent:fromView:** class method, which sends an **acceptColor:atPoint:** message to the receiving View.

See also: NXColorPanel

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from View	NXRect	frame;
	NXRect	bounds;
	id	superview;
	id	subviews;
	id	window;
	struct __vFlags	vFlags;
Inherited from Control	int	tag;
	id	cell;
	struct _conFlags	conFlags;
Defined in NXColorWell	NXColor	color;

color	The current color value of the NXColorWell

METHOD TYPES

Initializing an NXColorWell	– initFrame:
Multiple NXColorWells	+ activeWellsTakeColorFrom:
	+ activeWellsTakeColorFrom:continuous:
	+ deactivateAllWells
Drawing	– drawSelf::
	– drawWellInside:
Handling events	– acceptsFirstMouse
	– mouseDown:
	– isContinuous
	– setContinuous:
Activating and enabling	– activate:
	– deactivate
	– isActive
	– setEnabled:
Setting color	– acceptColor:atPoint:
	– setColor:
	– color
	– takeColorFrom:
Target and action	– target
	– action
	– setTarget:
	– setAction:

CLASS METHODS

activeWellsTakeColorFrom:

+ **activeWellsTakeColorFrom:***sender*

This method changes the color of all active NXColorWells by invoking their **takeColorFrom:** method with *sender* as the argument. Returns the NXColorWell class object.

See also: – **activate:**, + **activeWellsTakeColorFrom:continuous:**, – **deactivate**, + **deactivateAllWells**, – **takeColorFrom:**

activeWellsTakeColorFrom:continuous

+ activeWellsTakeColorFrom:*sender* **continuous:**(BOOL)*flag*

If *flag* is YES, this method continuously changes the color of all active NXColorWells that are continuous; If NO, all active NXColorWells, continuous or not, change their color just once. NXColorWells are updated by invoking their **takeColorFrom:** method with *sender* as the argument. Use this method with YES as the *flag* in a modal event loop if you want active NXColorWells to continuously update to reflect the current color of *sender.* Returns the NXColorWell class object.

See also: – **activate:**, – **deactivate**, + **deactivateAllWells,** – **isContinuous**, – **setContinuous:**, – **takeColorFrom:**

deactivateAllWells

+ deactivateAllWells

Deactivates all currently active NXColorWells. Returns the NXColorWell class object.

See also: – **activate:**, – **deactivate**

INSTANCE METHODS

acceptColor:atPoint:

– **acceptColor:**(NXColor)*color* **atPoint:**(NXPoint *)*aPoint*

Changes the color value of the NXColorWell to *color* when *aPoint* is a point within the bounds of the NXColorWell. This method is invoked by the NXColorPanel method **dragColor:withEvent:fromView:** to move color into an NXColorWell. Returns **self**.

Note that any subclass of View can accept colors from an NXColorWell by implementing a version of this method.

See also: – **dragColor:withEvent:fromView:** (NXColorPanel)

acceptsFirstMouse

– **acceptFirstMouse**

Returns YES.

action

– **action**

Returns the action sent by the NXColorWell to the target.

activate:

– (int)**activate:**(int)*exclusive*

If *exclusive* is YES, this method activates the receiving NXColorWell and deactivates any other active NXColorWells. If NO, this method activates the receiving NXColorWell and keeps previously active NXColorWells active. Redraws the receiver. An active NXColorWell will have its color updated as the NXColorPanel's current color changes.

This method returns the number of active NXColorWells.

See also: **+ activeWellsTakeColorFrom:**, **– deactivate**

color

– (NXColor)**color**

Returns the color of the NXColorWell.

See also: **– acceptColor:atPoint**, **– setColor:**, **– takeColorFrom:**

deactivate

– **deactivate**

Sets the NXColorWell to inactive and redraws it. Returns **self**.

drawSelf::

– **drawSelf:**(const NXRect *)*rects*:(int)*rectCount*

Draws the entire NXColorWell, including its border. Returns **self**.

drawWellInside:

– **drawWellInside:**(const NXRect *)*insideRect*

Draws the inside of the NXColorWell only, the area where the color is displayed. Returns **self**.

initFrame:

– **initFrame:**(NXRect const *)*theFrame*

Initializes and returns the receiver, a new instance of NXColorPanel within *theFrame*. By default, the color is NX_COLORWHITE and the NXColorWell is bordered and inactive. Returns **self**.

isActive

– (BOOL)**isActive**

Returns YES if the receiving NXColorWell is active, NO if not active.

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

Makes the receiver the only active NXColorWell. If theEvent occurs within the colored area of the NXColorWell (not on its border), then this method invokes NXColorPanel's **dragColor:withEvent:fromView:** method. The user can then drag the color from the NXColorWell to another View that has an **acceptColor:atPoint:** method, such as another NXColorWell. Returns **self**.

You never invoke this method. It's sent when an NX_MOUSEDOWN event occurs within the bounds of the NXColorWell.

See also: – **acceptColor:atPoint:**, – **activate**, – **deactivate**, + **dragColor:withEvent:fromView:** (NXColorPanel), – **isActive**

setAction:

– **setAction:**(SEL) *aSelector*

Sets the default action method of the NXColorWell. The action message is sent to the target by NXColorWell's **acceptColor:atPoint:** and **takeColorFrom:** methods. Returns **self**.

setColor:

– **setColor:**(NXColor)*color*

Sets the color of the NXColorWell to *color* and redraws it. Returns **self**.

setContinuous:

– **setContinuous:**(BOOL)*flag*

If *flag* is YES, the NXColorWell continuously updates its color and sends its action message to its target in response to an **activeWellsTakeColorFrom:continuous:**. If NO, the NXColorWell doesn't respond to this message. Returns **self**.

setEnabled:

– **setEnabled:**(BOOL)*flag*

If *flag* is YES, the receiving NXColorWell is enabled. If NO, the receiver is disabled. An NXColorWell cannot be both disabled and active; enabling an NXColorWell doesn't activate. Returns **self**.

See also: **– activate**, **– deactivate**, – **isActive**

setTarget:

– **setTarget:***anObject*

Sets the target of the NXColorWell. The action message is sent to the target by NXColorWell's **acceptColor:atPoint:** and **takeColorFrom:** methods. Returns **self**.

takeColorFrom

– **takeColorFrom:***sender*

Causes the receiving NXColorWell to set its color by sending a **color** message to *sender*. Sends the NXColorWell's action message to its target and returns **self**.

See also: – **color**

target

– **target**

Returns the target of the NXColorWell. The action message is sent to the target by NXColorWell's **acceptColor:atPoint:** and **takeColorFrom:** methods. Returns **self**.

See also: – **setTarget:**

NXCursor

INHERITS FROM	Object
DECLARED IN	appkit/NXCursor

CLASS DESCRIPTION

An NXCursor object holds an image that can become the image the Window Server displays for the cursor. A **set** message makes the receiver the current cursor:

```
[myNXCursor set];
```

For automatic cursor management, an NXCursor can be assigned to a cursor rectangle within a window. When the window is the key window and the user moves the cursor into the rectangle, the NXCursor is set to be the current cursor. It ceases to be the current cursor when the cursor leaves the rectangle. The assignment is made using View's **addCursorRect:cursor:** method, usually inside a **resetCursorRects** method:

```
- resetCursorRects
{
    [self addCursorRect:&someRect cursor:theNXImageObject];
    return self;
}
```

This is the recommended way of associating a cursor with a particular region inside a window. However, the NXCursor class provides two other ways of setting the cursor:

- The class maintains its own stack of cursors. Pushing an NXCursor instance on the stack sets it to be the current cursor. Popping an NXCursor from the stack sets the next NXCursor in line, the one that's then at the top of the stack, to be the current cursor.

- An NXCursor can be made the owner of a tracking rectangle and told to set itself when it receives a mouse-entered or mouse-exited event.

The Application Kit provides two ready-made NXCursor instances and assigns them to global variables:

NXArrow	The standard arrow cursor
NXIBeam	The cursor that's displayed over editable or selectable text

There's no NXCursor instance for the wait cursor. The wait cursor is displayed automatically by the system, without any required program intervention.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in NXCursor	NXPoint	hotSpot;
	struct _csrFlags {	
	unsigned int	onMouseExited:1;
	unsigned int	onMouseEntered:1;
	}	cFlags;
	id	image;

hotSpot	The point in the cursor image whose location on the screen is reported as the cursor's location.
cFlags.onMouseExited	A flag indicating whether to set the cursor when the Cursor object receives a mouse-exited event.
cFlags.onMouseEntered	A flag indicating whether to set the cursor when the Cursor object receives a mouse-entered event.
image	The cursor image, an NXImage object.

METHOD TYPES

Initializing a new NXCursor object	– init – initFromImage:
Defining the cursor	– setImage: – image – setHotSpot:
Setting the cursor	– push – pop + pop – set – setOnMouseEntered: – setOnMouseExited: – mouseEntered: – mouseExited: + currentCursor
Archiving	– read: – write:

CLASS METHODS

currentCursor

+ currentCursor

Returns the last NXCursor to have been set.

See also: – **set**, – **push**, + **pop**, – **mouseEntered:**, – **mouseExited:**,

pop

+ pop

Removes the NXCursor currently at the top of the cursor stack, and sets the NXCursor that was beneath it (but is now at the top of the stack) to be the current cursor. Returns **self** (the class object).

This method can be used in conjunction with the **push** method to manage a group of cursors within a local context. Every **push** should be balanced by a subsequent **pop**. When the last remaining cursor is popped from the stack, the Application Kit restores a cursor appropriate for the larger context.

The **pop** instance method provides the same functionality as this class method.

See also: – **push**

INSTANCE METHODS

image

– image

Returns the NXImage object that supplies the cursor image for the receiving NXCursor.

See also: – **initFromImage:**, – **setImage:**

init

– init

Initializes the receiver, a newly allocated NXCursor instance, by sending it an **initFromImage:** message with **nil** as the argument. This doesn't assign an image to the object. An image must then be set (with the **setImage:** method) before the cursor can be used. Returns **self**.

See also: – **setImage:**, – **initFromImage:**

initFromImage:

– **initFromImage:***image*

Initializes the receiver, a newly allocated NXCursor instance, by setting the image it will use to *image*, an NXImage object. For a standard cursor, the image should be 16 pixels wide by 16 pixels high. The default hot spot is at the upper left corner of the image.

This method is the designated initializer for the class. Returns **self**.

See also: – **setHotSpot:**

mouseEntered:

– **mouseEntered:**(NXEvent *)*theEvent*

Responds to a mouse-entered event by setting the NXCursor to be the current cursor, but only if enabled to do so by a previous **setOnMouseEntered:** message. This method does not push the receiver on the cursor stack. Returns **self**.

See also: – **setOnMouseEntered:**

mouseExited:

– **mouseExited:**(NXEvent *)*theEvent*

Responds to a mouse-exited event by setting the NXCursor to be the current cursor, but only if enabled to do so by a previous **setOnMouseExited:** message. This method does not push the receiver on the cursor stack. Returns **self**.

See also: – **setOnMouseExited:**

pop

– **pop**

Removes the topmost NXCursor object, not necessarily the receiver, from the cursor stack, and makes the next NXCursor down the current cursor. Returns **self**.

This method is an interface to the class method with the same name.

See also: + **pop**, – **push**

push

– **push**

Puts the receiving NXCursor on the cursor stack and sets it to be the Window Server's current cursor. Returns **self**.

This method can be used in conjunction with the **pop** method to manage a group of cursors within a local context. Every **push** should be matched by a subsequent **pop**.

See also: + **pop**

read:

– **read:**(NXTypedStream *)*stream*

Writes the NXCursor, including the image, to *stream*.

See also: – **write:**

set

– **set**

Makes the NXCursor the current cursor displayed by the Window Server, and returns **self**. This method doesn't push the receiver on the cursor stack.

setHotSpot:

– **setHotSpot:**(const NXPoint *)*aPoint*

Sets the point on the cursor that will be used to report its location. The point is specified relative to a flipped coordinate system with an origin at the upper left corner of the cursor image and coordinate units equal to those of the base coordinate system. The point should not have any fractional coordinates, meaning that it should lie at the corner of four pixels. The point selects the pixel below it and to its right. This pixel is the one part of the cursor image that's guaranteed never to be off-screen.

When the pixel selected by the hot spot lies inside a rectangle (say a button), the cursor is said to be over the rectangle. When the pixel is outside the rectangle, the cursor is taken to be outside the rectangle, even if other parts of the image are inside.

The default hot spot is at the upper left corner of the image. Returns **self**.

setImage:

– **setImage:***image*

Sets a new *image* for the NXCursor, and returns the old image. The new *image* should be an NXImage object. If the old image is of no further use, it should be freed. Resetting the image while the cursor is displayed may have unpredictable results.

See also: – **image**, – **initFromImage:**

setOnMouseEntered:

– **setOnMouseEntered:**(BOOL)*flag*

Determines whether the NXCursor should set itself to be the current cursor when it receives a **mouseEntered:** event message. To be able to receive the event message, an NXCursor must first be made the owner of a tracking rectangle by Window's **setTrackingRect:inside:owner:tag:left:right:** method.

Cursor rectangles are a more convenient way of associating cursors with particular areas within a window.

Returns **self**.

See also: – **mouseEntered:**, – **setTrackingRect:inside:owner:tag:left:right:** (Window)

setOnMouseExited:

– **setOnMouseExited:**(BOOL)*flag*

Determines whether the NXCursor should set itself to be the current cursor when it receives a **mouseExited:** event message. To be able to receive the event message, an NXCursor must first be made the owner of a tracking rectangle by Window's **setTrackingRect:inside:owner:tag:left:right:** method.

Cursor rectangles are a more convenient way of associating cursors with particular areas within windows.

Returns **self**.

See also: – **mouseExited:**, – **setTrackingRect:inside:owner:tag:left:right:** (Window)

write:

– **write:**(NXTypedStream *)*stream*

Writes the NXCursor and its image to *stream*.

See also: – **read:**

NXCustomImageRep

INHERITS FROM	NXImageRep : Object
DECLARED IN	appkit/NXCustomImageRep.h

CLASS DESCRIPTION

An NXCustomImageRep is an object that uses a delegated method to render an image. When called upon to produce the image, it sends a message to have the method performed.

Like most other kinds of NXImageReps, an NXCustomImageRep is generally used indirectly, through an NXImage object. To be useful to the NXImage, it must be able to provide some information about the image. The following methods, inherited from the NXImageRep class, inform the NXCustomImageRep about the size of the image, whether it can be drawn in color, and so on. Use them to complete the initialization of the object.

setSize:
setNumColors:
setAlpha:
setPixelsHigh
setPixelsWide
setBitsPerSample:

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from NXImageRep	NXSize	size;
Declared in NXCustomImageRep	SEL	drawMethod;
	id	drawObject;

drawMethod	The method that draws the image.
drawObject	The object that receives messages to perform the **drawMethod**.

METHOD TYPES

Initializing a new NXCustomImageRep	– initDrawMethod:inObject:
Drawing the image	– draw
Archiving	– read: – write:

INSTANCE METHODS

draw

– (BOOL)**draw**

Sends a message to have the image drawn. Returns YES if the message is successfully sent, and NO if not. The message will not be sent if the intended receiver is **nil** or it can't respond to the message.

See also: – **drawAt:** (NXImageRep), – **drawIn:** (NXImageRep)

init

Generates an error message. This method cannot be used to initialize an NXCustomImageRep. Use **initDrawMethod:inObject:** instead.

See also: – **initDrawMethod:inObject:**

initDrawMethod:inObject:

– **initDrawMethod:**(SEL)*aSelector* **inObject:***anObject*

Initializes the receiver, a newly allocated NXCustomImageRep instance, so that it delegates responsibility for rendering the image to *anObject*. When the NXCustomImageRep receives a **draw** message, it will in turn send a message to *anObject* to perform the *aSelector* method. The *aSelector* method should take only one argument, the **id** of the NXCustomImageRep. It should draw the image at location (0.0, 0.0) in the current coordinate system.

Returns **self**.

read:

– **read:**(NXTypedStream *)*stream*

Reads the NXCustomImageRep from the typed stream *stream*.

See also: – **write:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the NXCustomImageRep to the typed stream *stream*. The object that's delegated to draw the image is not explicitly written.

See also: – **read:**

NXEPSImageRep

INHERITS FROM	NXImageRep : Object
DECLARED IN	appkit/NXEPSImageRep

CLASS DESCRIPTION

An NXEPSImageRep is an object that can render an image from encapsulated PostScript code (EPS). The size of the object is set from the bounding box specified in the EPS header comments. Other information about the image should be supplied using inherited NXImageRep methods.

Like most other kinds of NXImageReps, an NXEPSImageRep is generally used indirectly, through an NXImage object.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from NXImageRep	NXSize	size;
Declared in NXEPSImageRep	(none)	

METHOD TYPES

Initializing a new NXEPSImageRep instance
- – initFromSection:
- – initFromFile:
- – initFromStream:

Creating a List of NXEPSImageReps
- + newListFromSection:
- + newListFromSection:zone:
- + newListFromFile:
- + newListFromFile:zone:
- + newListFromStream:
- + newListFromStream:zone:

Copying and freeing an NXEPSImageRep
- – copy
- – free

Getting the rectangle that bounds the image
- – getBoundingBox:

Getting image data	– getEPS:length:
Drawing the image	– prepareGState – drawIn: – draw
Archiving	– read: – write:

newListFromFile:

+ (List *)**newListFromFile:**(const char *)*filename*

Creates one new NXEPSImageRep instance for each EPS image specified in the *filename* file, and returns a List object containing all the objects created. If no NXEPSImageReps can be created (for example, if *filename* doesn't exist or doesn't contain EPS code), **nil** is returned. The List should be freed when it's no longer needed.

Each new NXEPSImageRep is initialized by the **initFromFile:** method, which reads a minimal amount of information about the image from the header comments in the file. The PostScript code will be read when it's needed to render the image.

See also: + **newListFromFile:zone:**, – **initFromFile:**

newListFromFile:zone:

+ (List *)**newListFromFile:**(const char *)*filename* **zone:**(NXZone *)*aZone*

Returns a List of new NXEPSImageRep instances, just as **newListFromFile:** does, except that the NXEPSImageReps and the List object are allocated from memory located in *aZone*.

See also: + **newListFromFile:**, – **initFromFile:**

newListFromSection:

+ (List *)**newListFromSection:**(const char *)*name*

Creates one new NXEPSImageRep instance for each image specified in the *name* section of the __EPS segment in the executable file, and returns a List object containing all the objects created. If not even one NXEPSImageRep can be created (for example, if the *name* section doesn't exist or doesn't contain EPS code), **nil** is returned. The List should be freed when it's no longer needed.

Each new NXEPSImageRep is initialized by the **initFromSection:** method, which reads reads a minimal amount of information about the image from the EPS header comments. The PostScript code will be read only when it's needed to render the image.

See also: + **newListFromSection:zone:**, – **initFromSection:**

newListFromSection:zone:

+ (List *)**newListFromSection:**(const char *)*name* **zone:**(NXZone *)*aZone*

Returns a List of new NXEPSImageRep instances, just as **newListFromSection:** does, except that the List object and the NXEPSImageReps are allocated from memory located in *aZone*.

See also: + **newListFromSection:**, – **initFromSection:**

newListFromStream:

+ (List *)**newListFromStream:**(NXStream *)*stream*

Creates one new NXEPSImageRep instance for each EPS image that can be read from *stream*, and returns a List object containing all the objects created. If not even one NXEPSImageRep can be created (for example, if the *stream* doesn't contain EPS code), **nil** is returned. The List should be freed when it's no longer needed.

The data is read and each new object initialized by the **initFromStream:** method.

See also: + **newListFromStream:zone:**, – **initFromStream:**

newListFromStream:zone:

+ (List *)**newListFromStream:**(NXStream *)*stream* **zone:**(NXZone *)*aZone*

Returns a List of new NXEPSImageRep instances, just as **newListFromStream:** does, except that the List object and the NXEPSImageReps are allocated from memory located in *aZone*.

See also: + **newListFromStream:**, – **initFromStream:**

INSTANCE METHODS

copy

– **copy**

Returns a new NXEPSImageRep instance that's an exact copy of the receiver. The new object will have its own copy of the image data. It doesn't need to be initialized.

draw

– (BOOL)**draw**

Draws the image at (0.0, 0.0) in the current coordinate system on the current device. This method returns YES if successful in rendering the image, and NO if not.

An NXEPSImageRep draws in a separate PostScript context and graphics state. Before the EPS code is interpreted, all graphics state parameters—with the exception of the CTM and device—are set to the Window Server's defaults and the defaults required by EPS conventions. If you want to change any of these defaults, you can do so by implementing a **prepareGState** method in an NXEPSImageRep subclass. The **draw** method invokes **prepareGState** just before sending the EPS code to the Window Server. For example, if you need to set a transfer function or halftone screen that's specific to the image, **prepareGState** is the place to do it.

See also: – **drawAt:** (NXImageRep), – **drawIn:**, – **prepareGState**

drawIn:

– (BOOL)**drawIn:**(const NXRect *)*rect*

Draws the image so that it fits inside the rectangle referred to by *rect*. The current coordinate system is translated and scaled so the image will appear at the right location and fit within the rectangle. The **draw** method is then invoked to produce the image. This method passes through the return value of the **draw** method, which indicates whether the image was successfully drawn.

The coordinate system is not restored after it has been altered.

See also: – **draw**, – **drawAt:** (NXImageRep)

free

– **free**

Deallocates the NXEPSImageRep.

getBoundingBox:

– **getBoundingBox:**(NXRect *)*theRect*

Provides the rectangle that bounds the image. The rectangle is copied from the "%%BoundingBox:" comment in the EPS header to structure referred to by *theRect*. Returns **self**.

getEPS:length:

– **getEPS:**(char **)*theEPS* **length:**(int *)*numBytes*

Sets the pointer referred to by *theEPS* so that it points to the EPS code. The length of the code in bytes is provided in the integer referred to by *numBytes*. Returns **self**.

init

Generates an error message. This method can't be used to initialize an NXEPSImageRep. Use one of the other **init**... methods instead.

See also: – **initFromSection:**, – **initFromFile:**, – **initFromStream:**

initFromFile:

– **initFromFile:**(const char *)*filename*

Initializes the receiver, a newly allocated NXEPSImageRep object, with the EPS image found in the *filename* file. Some information about the image is read from the EPS header comments, but the PostScript code won't be read until it's needed to render the image.

If the new object can't be initialized for any reason (for example, *filename* doesn't exist or doesn't contain EPS code), this method frees it and returns **nil**. Otherwise, it returns **self**.

This method is the designated initializer for NXEPSImageReps that read EPS code from a file.

See also: + **newListFromFile:**, – **initFromSection:**

initFromSection:

– **initFromSection:**(const char *)*name*

Initializes the receiver, a newly allocated NXEPSImageRep object, with the image found in the *name* section in the __EPS segment of the application executable. Some information about the image is read from the EPS header comments, but the PostScript code won't be read until it's needed to render the image.

If the new object can't be initialized for any reason (for example, the *name* section doesn't exist or doesn't contain EPS code), this method frees it and returns **nil**. Otherwise, it returns **self**.

This method is the designated initializer for NXEPSImageReps that read image data from the __EPS segment..

See also: + **newListFromSection:**, – **initFromFile:**

initFromStream:

– **initFromStream:**(NXStream *)*stream*

Initializes the receiver, a newly allocated NXEPSImageRep object, with the EPS image read from *stream*. If the new object can't be initialized for any reason (for example, *stream* doesn't contain EPS code), this method frees it and returns **nil**. Otherwise, it returns **self**.

This method is the designated initializer for NXEPSImageReps that read image data from a stream.

See also: + **newListFromStream:**

prepareGState

– **prepareGState**

Implemented by subclasses to initialize the graphics state before the image is drawn. The **draw** method sends a **prepareGState** message just before rendering the EPS code. This default implementation of the method does no initialization; it simply returns **self**.

See also: – **draw**

read:

– **read:**(NXTypedStream *)*stream*

Reads the NXEPSImageRep from the typed stream *stream*.

See also: – **write:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the NXEPSImageRep to the typed stream *stream*.

See also: – **read:**

NXImage

INHERITS FROM	Object
DECLARED IN	appkit/NXImage.h

CLASS DESCRIPTION

An NXImage object contains an image that can be composited anywhere without first being drawn in any particular View. It manages the image by:

- Reading image data from the __ICON, __TIFF, or __EPS segments of the application executable, from a separate file, or from an NXStream.
- Keeping multiple representations of the same image.
- Choosing the representation that's appropriate for any given display device.
- Caching the representations it uses by rendering them in off-screen windows.
- Optionally retaining the data used to draw the representations, so that they can be reproduced when needed.
- Compositing the image from the off-screen cache to where it's needed on-screen.
- Reproducing the image for the printer so that it matches what's displayed on-screen, yet is the best representation possible for the printed page.

Defining an Image

An image can be created from various types of data:

- Encapsulated PostScript code (EPS)
- Bitmap data in Tag Image File Format (TIFF)
- Untagged (raw) bitmap data

If TIFF or EPS image data is placed in a section of the application executable or in a separate file, the NXImage object can access the data whenever it's needed to create the image. If TIFF or EPS data is read from a stream, the NXImage object may need to retain the data itself.

Images can also be defined by the program, in two ways:

- By drawing the image in an off-screen window maintained by the NXImage object. In this case, the NXImage maintains only the cached image.
- By defining a method that can be used to draw the image when needed. This allows the NXImage to delegate responsibility for producing the image to some other object.

Image Representations

An NXImage object can keep more than one representation of an image. Multiple representations permit the image to be customized for the display device. For example, different hand-tuned TIFF images can be provided for monochrome and color screens, and an EPS representation or a custom method might be used for printing. All representations are versions of the same image.

An NXImage returns a List of its representations in response to a **representationList** message. Each representation is a kind of NXImageRep object:

NXEPSImageRep	An image that can be recreated from EPS data that's either retained by the object or at a known location in the file system.
NXBitmapImageRep	An image that can be recreated from bitmap or TIFF data.
NXCustomImageRep	An image that can be redrawn by a method defined in the application.
NXCachedImageRep	An image that has been rendered in an off-screen cache from data or instructions that are no longer available. The image in the cache provides the only data from which the image can be reproduced.

You can also define other NXImageRep subclasses for objects that render images from other kinds of source information.

Choosing and Caching Representations

The NXImage object will choose the representation that best matches the rendering device. By default, the choice is made according to the following set of ordered rules. Each rule is applied in turn until the choice of representation is narrowed to one:

6. Choose a color representation for a color device, and a gray-scale representation for a monochrome device.

7. Choose a representation with a resolution that matches the resolution of the device, or if no representation matches, choose the one with the highest resolution.

   By default, any image representation with a resolution that's an integer multiple of the device resolution is considered to match. If more than one representation matches, the NXImage will choose the one that's closest to the device resolution. However, you can force resolution matches to be exact by passing NO to the **setMatchedOnMultipleResolution:** method.

   Rule 2 prefers TIFF and bitmap representations, which have a defined resolution, over EPS representations, which don't. However, you can use the **setEPSPreferredOnResolutionMismatch:** method to have the NXImage choose an EPS representation in case a resolution match isn't possible.

8. If all else fails, choose the representation with a specified bits per sample that matches the depth of the device. If no representation matches, choose the one with the highest bits per sample.

By passing NO to the **setColorMatchPreferred:** method, you can have the NXImage try for a resolution match before a color match. This essentially inverts the first and second rules above.

When first asked to composite the image, the NXImage object chooses the representation that's best for the destination display device. It renders the representation in an off-screen window on the same device, then composites it from this cache to the desired location. Subsequent requests to composite the image use the same cache. Representations aren't cached until they're needed for compositing.

When printing, the NXImage tries not to use the cached image. Instead, it attempts to render on the printer—using the appropriate EPS or TIFF data, or a delegated method—the best version of the image that it can. Only as a last resort will it image the cached bitmap.

Image Size

Before an NXImage can be used, the size of the image must be set, in units of the base coordinate system. If a representation is smaller or larger than the specified size, it can be scaled to fit.

If the size of the image hasn't already been set when the NXImage is provided with an EPS or TIFF representation, the size will be set from the EPS or TIFF data. The EPS bounding box and TIFF "ImageLength" and "ImageWidth" fields specify an image size.

Coordinate Systems

Images have the horizontal and vertical orientation of the base coordinate system; they can't be rotated or flipped. When composited, an image maintains this orientation, no matter what coordinate system it's composited to. (The destination coordinate system is used only to determine the location of a composited image, not its size or orientation.)

It's possible to refer to portions of an image when compositing (or when defining subimages), by specifying a rectangle in the image's coordinate system, which is identical to the base coordinate system, except that the origin is at the lower left corner of the image.

Named Images

An NXImage object can be identified either by its **id** or by a name. Assigning an NXImage a name adds it to a database kept by the class object; each name in the database identifies one and only one instance of the class. When you ask for an NXImage object by name (with the **findImageNamed:** method), the class object returns the one from its database, which also includes all the system bitmaps provided by the Application Kit. If there's no object in the database for the specified name, the class object tries to create one by looking in the __ICON, __EPS, and __TIFF segments of the application's executable file, and then in the directory of the executable file (the file package).

If a section or file matches the name, an NXImage is created from the data stored there. You can therefore create NXImage objects simply by including EPS or TIFF data for them within the executable file, or in files inside the application's file package.

The job of displaying an image within a View can be entrusted to a Cell object. A Cell identifies the image it's to display by the name of the NXImage object. The following code sets **myCell** to display one of the system bitmaps:

```
id myCell = [[Cell alloc] initIconCell:"NXswitch"];
```

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in NXImage	char	*name;
name	The name assigned to the image.	

METHOD TYPES

Initializing a new NXImage instance	
	– init
	– initSize:
	– initFromSection:
	– initFromFile:
	– initFromStream:
	– initFromImage:rect:
Freeing an NXImage object	– free
Setting the size of the image	– setSize:
	– getSize:
Referring to images by name	– setName:
	– name
	+ findImageNamed:
Specifying the image	– useDrawMethod:inObject:
	– useFromSection:
	– useFromFile:
	– useRepresentation:
	– useCacheWithDepth:
	– loadFromStream:
	– lockFocus
	– lockFocusOn:
	– unlockFocus
Using the image	– composite:toPoint:
	– composite:fromRect:toPoint:
	– dissolve:toPoint:
	– dissolve:fromRect:toPoint:
Choosing which image representation to use	
	– setColorMatchPreferred:
	– isColorMatchPreferred
	– setEPSUsedOnResolutionMismatch:
	– isEPSUsedOnResolutionMismatch
	– setMatchedOnMultipleResolution:
	– isMatchedOnMultipleResolution
Getting the representations	– lastRepresentation
	– bestRepresentation
	– representationList
	– removeRepresentation:

Determining how the image is stored	
	– setUnique:
	– isUnique
	– setDataRetained:
	– isDataRetained
	– setCacheDepthBounded:
	– isCacheDepthBounded
	– getImage:rect:
Determining how the image is drawn	
	– setFlipped:
	– isFlipped
	– setScalable:
	– isScalable
	– setBackgroundColor:
	– backgroundColor
	– drawRepresentation:inRect:
	– recache
Assigning a delegate	– setDelegate:
	– delegate
Producing TIFF data for the image	– writeTIFF:
	– writeTIFF:allRepresentations:
Archiving	– read:
	– write:
	– finishUnarchiving

CLASS METHODS

findImageNamed:

+ **findImageNamed:**(const char *)*name*

Returns the NXImage instance associated with *name*. The returned object can be:

- One that's been assigned a name with the **setName:** method,
- One of the named system bitmaps provided by the Application Kit, or
- One that's been created and named by this method.

If there's no known NXImage with *name*, this method tries to create one by searching for image data in the __ICON, __EPS, and __TIFF segments of the application executable and in the directory (file package) where the executable resides:

1. It looks first in the __ICON segment for a *name* section containing either Encapsulated PostScript code (EPS) or Tag Image File Format (TIFF) data.

2. Failing to find image data there, it looks next for a section with TIFF data in the __TIFF segment if *name* includes a ".tiff" extension, or for a section containing EPS data in the __EPS segment if *name* includes a ".eps" extension. If *name* has neither extension, both segments are searched, first after adding the appropriate extension to *name*, then for *name* alone, without an extension. If it finds sections in both segments, it creates both EPS and TIFF representations of the image.

3. If this method can't find a EPS or TIFF representation in any segment, it searches for *name*.eps and *name*.tiff files in the directory containing the application executable (the file package). This allows you to keep image data in separate files during the development phase (so that you won't have to relink every time the image changes), then later insert the data in a segment of the finished executable.

If a section or file contains data for more than one image, a separate representation is created for each one. If an image representation can't be found for *name*, no object is created and **nil** is returned.

The preferred way to name an EPS or TIFF image is to ask for a *name* without the ".eps" or ".tiff" extension, but to include the extension on the section name or file name.

This method treats all images found in the __ICON segment as application or document icons, since the point of putting an image in that segment rather than in __TIFF or __EPS is to advertise it to the Workspace Manager. The Workspace Manager requires icons to be no more than 48 pixels wide by 48 pixels high. Therefore, an NXImage created from an __ICON section has its size set to 48.0 by 48.0 and is made scalable.

See also: – **setName:**, – **name**

INSTANCE METHODS

backgroundColor

– (NXColor)**backgroundColor**

Returns the background color of the rectangle where the image is cached. If no background color has been specified, NX_COLORCLEAR is returned, indicating a totally transparent background.

The background color will be visible when the image is composited only if the image doesn't completely cover all the pixels within the area specified for its size.

See also: – **setBackgroundColor:**

bestRepresentation

– (NXImageRep *)**bestRepresentation**

Returns the image representation that best matches the display device with the deepest frame buffer currently available to the Window Server.

See also: – **representationList**

composite:fromRect:toPoint:

– **composite:**(int)*op*
  fromRect:(const NXRect *)*aRect*
  toPoint:(const NXPoint *)*aPoint*

Composites the area enclosed by the *aRect* rectangle to the location specified by *aPoint* in the current coordinate system. The *op* and *aPoint* arguments are the same as for **composite:toPoint:**. The source rectangle is specified relative to a coordinate system that has its origin at the lower left corner of the image, but is otherwise the same as the base coordinate system.

This method doesn't check to be sure that the rectangle encloses only portions of the image. Therefore, it can conceivably composite areas that don't properly belong to the image, if the *aRect* rectangle happens to include them. If this turns out to be a problem, you can prevent it from happening by having the NXImage cache its representations in their own individual windows (with the **setUnique:** method). The window's clipping path will prevent anything but the image from being composited.

Compositing part of an image is as efficient as compositing the whole image, but printing just part of an image is not. When printing, it's necessary to draw the whole image and rely on a clipping path to be sure that only the desired portion appears.

If successful in compositing (or printing) the image, this method returns **self**. If not, it returns **nil**.

See also: – **composite:toPoint:**, – **setUnique:**

composite:toPoint:

– **composite:**(int)*op* **toPoint:**(const NXPoint *)*aPoint*

Composites the image to the location specified by *aPoint*. The first argument, *op*, names the type of compositing operation requested. It should be one of the following constants:

NX_CLEAR	NX_SOVER	NX_DOVER	NX_XOR
NX_COPY	NX_SIN	NX_DIN	
NX_PLUSD	NX_SOUT	NX_DOUT	
NX_PLUSL	NX_SATOP	NX_DATOP	

aPoint is specified in the current coordinate system—the coordinate system of the currently focused View—and designates where the lower left corner of the image will appear. The image will have the orientation of the base coordinate system, regardless of the destination coordinates.

The image is composited from its off-screen window cache. Since the cache isn't created until the image representation is first used, this method may need to render the image before compositing.

When printing, the compositing methods do not composite, but attempt to render the same image on the page that compositing would render on the screen, choosing the best available representation for the printer. The *op* argument is ignored.

If successful in compositing (or printing) the image, this method returns **self**. If not, it returns **nil**.

See also: – **composite:fromRect:toPoint:**, – **dissolve:toPoint:**

delegate

– **delegate**

Returns the delegate of the NXImage object, or **nil** if no delegate has been set.

See also: – **setDelegate:**

dissolve:fromRect:toPoint:

– **dissolve:**(float)*delta*
  fromRect:(const NXRect *)*aRect*
  toPoint:(const NXPoint *)*aPoint*

Composites the *aRect* portion of the image to the location specified by *aPoint*, just as **composite:fromRect:toPoint:** does, but uses the **dissolve** operator rather than **composite**. *delta* is a fraction between 0.0 and 1.0 that specifies how much of the resulting composite will come from the NXImage.

When printing, this method is identical to **composite:fromRect:toPoint:**. The *delta* argument is ignored.

If successful in compositing (or printing) the image, this method returns **self**. If not, it returns **nil**.

See also: – **dissolve:toPoint:**, – **composite:fromRect:toPoint:**

dissolve:toPoint:

– **dissolve:**(float)*delta* **toPoint:**(const NXPoint *)*aPoint*

Composites the image to the location specified by *aPoint*, just as **composite:toPoint:** does, but uses the **dissolve** operator rather than **composite**. *delta* is a fraction between 0.0 and 1.0 that specifies how much of the resulting composite will come from the NXImage.

To slowly dissolve one image into another, this method (or **dissolve:fromRect:toPoint:**) needs to be invoked repeatedly with an ever-increasing *delta*. Since *delta* refers to the fraction of the source image that's combined with the original destination (not the destination image after some of the source has been dissolved into it), the destination image should be replaced with the original destination before each invocation. This is best done in a buffered window before the results of the composite are flushed to the screen.

When printing, this method is identical to **composite:toPoint:**. The *delta* argument is ignored.

If successful in compositing (or printing) the image, this method returns **self**. If not, it returns **nil**.

See also: – **dissolve:fromRect:toPoint:**, – **composite:toPoint:**

drawRepresentation:inRect:

– (BOOL)**drawRepresentation:**(NXImageRep *)*imageRep*
  inRect:(const NXRect *)*rect*

Fills the specified rectangle with the background color, then sends the *imageRep* a **drawIn:** message to draw itself inside the rectangle (if the NXImage is scalable), or a **drawAt**: message to draw itself at the location of the rectangle (if the NXImage is not scalable). The rectangle is located in the current window and is specified in the current coordinate system.

This method is not ordinarily used in program code; the NXImage uses it to cache its representations and to print them. By overriding it in a subclass, you can change how representations appear in the cache, and thus how they'll appear when composited. For example, you could scale or rotate the coordinate system, then send a message to **super** to perform this version of the method.

This method passes through the return of the **drawIn:** or **drawAt:** method, which indicates whether or not the representation was successfully drawn. When NO is returned, the NXImage will ask another representation, if there is one, to draw the image.

If the background color is fully transparent and the image is not being cached by the NXImage, the rectangle won't be filled before the representation draws.

See also: – **drawIn** (NXImageRep), – **drawAt:** (NXImageRep)

finishUnarchiving

– **finishUnarchiving**

Registers the name of the newly unarchived receiver, if it has a name, and returns **nil**. It also returns **nil** if the receiving NXImage doesn't have a name. However, if the receiver has a name that can't be registered because it's already in use, this method frees the receiver and returns the existing NXImage with that name, thus replacing the unarchived object with one that's already in use.

finishUnarchiving messages are generated automatically (by **NXReadObject()**) after the object has be unarchived (by **read:**) and initialized (by **awake**).

free

– **free**

Deallocates the NXImage and all its representations. If the object had been assigned a name, the name is removed from the class database.

getImage:rect:

– **getImage:**(NXImage **)*theImage* **rect:**(NXRect *)*theRect*

Provides information about the receiving NXImage object, if it's a subimage of another NXImage. The parent NXImage is assigned to the variable referred to by *theImage*, and the rectangle where the receiver is located in that NXImage is copied into the structure referred to by *theRect*.

If the receiver is not a subimage of another NXImage object (if it wasn't initialized by **initFromImage:rect:**), the variable referred to by *theImage* is set to **nil** and the rectangle is not modified.

Returns **self**.

See also: – **initFromImage:rect:**

getSize:

– **getSize:**(NXSize *)*theSize*

Copies the size of the image into the structure specified by *theSize*. If no size has been set, all values in the structure will be set to 0.0. Returns **self**.

See also: – **setSize:**

init

– **init**

Initializes the receiver, a newly allocated NXImage instance, but does not set the size of the image. The size must be set, and at least one image representation provided, before the NXImage object can be used. The size can be set either through a **setSize:** message or by providing an image from data (EPS or TIFF) that specifies a size.

See also: – **initSize:**, – **setSize:**

initFromFile:

– **initFromFile:**(const char *)*filename*

Initializes the receiver, a newly allocated NXImage instance, with the image specified in *filename*, which can be a full or relative pathname. The file should contain EPS or TIFF data for one or more versions of the image. An image representation will be created and added to the NXImage for each image specified. The size of the NXImage is set from information found in the TIFF fields or the EPS bounding box comment.

After finishing the initialization, this method returns **self**. However, if the new instance can't be initialized, it's freed and **nil** is returned.

This method invokes the **useFromFile:** method to find *filename* and create representations for the NXImage. It's equivalent to a combination of **init** and **useFromFile:**.

See also: – **useFromFile:**, – **initSize:**

initFromImage:rect:

– **initFromImage:**(NXImage *)*image* **rect:**(const NXRect *)*rect*

Initializes the receiver, a newly allocated NXImage instance, so that it's a subimage for the *rect* portion of another NXImage object, *image*. The size of the new object is set from the size of the *rect* rectangle. Returns **self**.

Once initialized, the new instance can't be altered and will remain dependent on the original image. Changes made to the original will also change the subimage.

Subimages should be used only as a way of avoiding **composite:fromRect:toPoint:** and **dissolve:fromRect:toPoint:** messages. They permit you to divide a large image into sections and assign each section a name. The name can then be passed to those Button and Cell methods that identify images by name rather than **id**.

See also: – **getImage:rect:**, – **initSize:**

initFromSection:

– **initFromSection:**(const char *)*name*

Initializes the receiver, a newly allocated NXImage instance, with the image specified in the *name* section of the __EPS or __TIFF segment of the application executable. If the section contains EPS or TIFF data for more than one version of the image, a representation will be created and added to the NXImage for each image specified. The size of the NXImage is set from information taken from the TIFF fields or the EPS bounding box comment.

After finishing the initialization, this method returns **self**. However, if the new instance can't be initialized, it's freed and **nil** is returned.

This method uses the **useFromSection:** method to find the *name* section and create representations for the NXImage. It's equivalent to a combination of **init** and **useFromFile:**.

See also: – **useFromSection:**, – **initSize:**

initFromStream:

– **initFromStream:**(NXStream *)*stream*

Initializes the receiver, a newly allocated NXImage instance, with the image or images specified in the data read from *stream*, and returns **self**. If the receiver can't be initialized for any reason, it's freed and **nil** is returned.

Since this method must store the data read from the stream or render the specified image immediately, it's less preferred than **initFromSection:** or **initFromFile:**, which can wait until the image is needed.

The stream should contain recognizable image data, either EPS or TIFF. It's read using the **loadFromStream:** method, which will set the size of the NXImage from information found in the TIFF fields or the EPS bounding box comment. This method is equivalent to a combination of **init** and **loadFromStream:**.

See also: – **loadFromStream:**, – **initSize:**

initSize:

– **initSize:**(const NXSize *)*aSize*

Initializes the receiver, a newly allocated NXImage instance, to the size specified and returns **self**. The size should be specified in units of the base coordinate system. It must be set before the NXImage can be used.

This method is the designated initializer for the class (the method that incorporates the initialization of classes higher in the hierarchy through a message to **super**). All other **init**... methods defined in this class work through this method.

See also: – **setSize:**

isCacheDepthBounded

– (BOOL)**isCacheDepthBounded**

Returns YES if the depth of the off-screen windows where the NXImage's representations are cached are bounded by the application's default depth limit, and NO if the depth of the caches can exceed that limit. The default is YES.

See also: – **setCacheDepthBounded:**, + **defaultDepthLimit** (Window)

isColorMatchPreferred

Returns YES if, when selecting the representation it will use, the NXImage first looks for one that matches the color capability of the rendering device (choosing a gray-scale representation for a monochrome device and a color representation for a color device), then if necessary narrows the selection by looking for one that matches the resolution of the device. If the return is NO, the NXImage first looks for a representation that matches the resolution of the device, then tries to match the representation to the color capability of the device. The default is YES.

See also: – **setColorMatchPreferred:**

isDataRetained

– (BOOL)**isDataRetained**

Returns YES if the NXImage retains the data needed to render the image, and NO if it doesn't. The default is NO. If the data is available in a section of the application executable or in a file that won't be moved or deleted, or if responsibility for drawing the image is delegated to another object with a custom method, there's no reason for the NXImage to retain the data. However, if the NXImage reads image data from a stream, you may want to have it keep the data itself.

See also: – **setDataRetained:**, – **loadFromStream:**

isEPSUsedOnResolutionMismatch

– (BOOL)**isEPSUsedOnResolutionMismatch**

Returns YES if an EPS representation of the image should be used whenever it's impossible to match the resolution of the device to the resolution of another representation of the image (a TIFF representation, for example). By default, this method returns NO to indicate that EPS representations are not necessarily preferred.

See also: – **setEPSUsedOnResolutionMismatch:**

isFlipped

– (BOOL)**isFlipped**

Returns YES if a flipped coordinate system is used when drawing the image, and NO if it isn't. The default is NO.

See also: – **setFlipped:**

isMatchedOnMultipleResolution

– (BOOL)**isMatchedOnMultipleResolution**

Returns YES if the resolution of the device and the resolution specified for the image are considered to match if one is a multiple of the other, and NO if device and image resolutions are considered to match only if they are exactly the same. The default is YES.

See also: – **setMatchedOnMultipleResolution:**

isScalable

– (BOOL)**isScalable**

Returns YES if image representations are scaled to fit the size specified for the NXImage. If representations are not scalable, this method returns NO. The default is NO.

Representations created from data that specifies a size (for example, the "ImageLength" and "ImageWidth" fields of a TIFF representation or the bounding box of an EPS representation) will have the size the data specifies, which may differ from the size of the NXImage.

See also: – **setScalable:**

isUnique

– (BOOL)**isUnique**

Returns YES if each representation of the image is cached alone in an off-screen window of its own, and NO if they can be cached in off-screen windows together with other images. A return of NO doesn't mean that the windows are, in fact, shared, just that they can be. The default is NO.

See also: – **setUnique:**

lastRepresentation

– (NXImageRep *)**lastRepresentation**

Returns the last representation that was specified for the image (the last one added with methods like **useCacheWithDepth:**, **loadFromStream:**, and **initFromStream:**). If the NXImage has no representations, this method returns **nil**.

See also: – **representationList**, – **bestRepresentation**

loadFromStream:

– (BOOL)**loadFromStream:**(NXStream *)*stream*

Creates an image representation from the data read from *stream* and adds it to the receiving NXImage's list of representations. The data must be of a recognizable type, either TIFF or EPS. If the size of the NXImage hasn't yet been set, it will be set from information found in the TIFF fields or from the EPS bounding box comment. If the stream contains data specifying more than one image, a separate representation is created for each one.

If the NXImage object doesn't retain image data (**isDataRetained** returns NO), the image will be rendered in an off-screen window and the representations will be of type NXCachedImageRep. If the data is retained, the representations will be of type NXBitmapImageRep or NXEPSImageRep, depending on the data.

If successful in creating at least one representation, this method returns YES. If not, it returns NO.

See also: – **initFromStream:**

lockFocus

– (BOOL)**lockFocus**

Focuses on the best representation for the NXImage, by making the off-screen window where the representation will be cached the current window and a coordinate system specific to the area where the image will be drawn the current coordinate system. The best representation is the one that best matches the deepest available frame buffer; it's the same object returned by the **bestRepresentation** method.

If the NXImage has no representations, **lockFocus** creates one with the **useCacheWithDepth:** method, specifying the best depth for the deepest frame buffer currently in use. To add additional representations, **useCacheWithDepth:** messages must be sent explicitly.

This method returns YES if it's successful in focusing on the representation, and NO if not. A successful **lockFocus** message must be balanced by a subsequent **unlockFocus** message to the same NXImage. These messages bracket the code that draws the image.

If **lockFocus** returns NO, it will not have altered the current graphics state and should not be balanced by an **unlockFocus** message.

See also: – **lockFocusOn:**, – **lockFocus** (View), – **unlockFocus**, – **useCacheWithDepth:**, – **bestRepresentation**

lockFocusOn:

– (BOOL)**lockFocusOn:**(NXImageRep *)*imageRep*

Focuses on the *imageRep* representation, by making the off-screen window where it will be cached the current window and a coordinate system specific to the area where the image will be drawn the current coordinate system.

This method returns YES if it's successful in focusing on the representation, and NO if it's not. A successful **lockFocusOn:** message must be balanced by a subsequent **unlockFocus** message to the same receiver. These messages bracket the code that draws the image. The **useCacheWithDepth:** method will add a representation specifically for this purpose. For example:

```
[myNXImage useCacheWithDepth:NX_TwoBitGrayDepth];
if ( [myNXImage lockFocusOn:[myImage lastRepresentation]] ) {
    /* drawing code goes here */
    [myNXImage unlockFocus];
}
```

If **lockFocusOn:** returns NO, it will not have altered the current graphics state and should not be balanced by an **unlockFocus** message.

See also: – **lockFocus**, – **lockFocus** (View), – **unlockFocus**, – **lastRepresentation**

name

– (const char *)**name**

Returns the name assigned to the NXImage, or NULL if no name has been assigned.

See also: – **setName:**, + **findImageNamed:**

read:

– **read:**(NXTypedStream *)*stream*

Reads the NXImage and all its representations from the typed stream *stream*.

See also: – **write:**

recache

– **recache**

Invalidates the off-screen caches of all representations and frees them. The next time any representation is composited, it will first be asked to redraw itself in the cache. NXCachedImageReps are not destroyed by this method.

If an image is likely not to be used again, it's a good idea to free its caches, since that will reduce that amount of memory consumed by your program and therefore improve performance.

Returns **self**.

removeRepresentation:

– **removeRepresentation:**(NXImageRep *)*imageRep*

Frees the *imageRep* representation after removing it from the NXImage's list of representations. Returns **self**.

See also: – **representationList**

representationList

– (List *)**representationList**

Returns the List object containing all the representations of the image. The List belongs to the NXImage object, and there's no guarantee that the same List object will be returned each time. Therefore, rather than saving the object that's returned, you should ask for it each time you need it.

See also: – **bestRepresentation**, – **lastRepresentation**

setBackgroundColor:

– **setBackgroundColor:**(NXColor)*aColor*

Sets the background color of the image. The default is NX_COLORCLEAR, indicating a totally transparent background. The background color will be visible only for representations that don't touch all the pixels within the image when drawing. Returns **self**.

See also: – **backgroundColor**

setCacheDepthBounded:

– **setCacheDepthBounded:**(BOOL)*flag*

Determines whether the depth of the off-screen windows where the NXImage's representations are cached should be limited by the application's default depth limit. If *flag* is NO, window depths will be determined by the specifications of the representations, rather than by the current display devices. The default is YES. Returns **self**.

See also: – **isCacheDepthBounded**, + **defaultDepthLimit** (Window)

setColorMatchPreferred:

– **setColorMatchPreferred:**(BOOL)*flag*

Determines how the NXImage will select which representation to use. If *flag* is YES, it first tries to match the representation to the color capability of the rendering device (choosing a color representation for a color device and a gray-scale representation for a monochrome device), and then if necessary narrows the selection by trying to match the resolution of the representation to the resolution of the device. If *flag* is NO, the NXImage first tries to match the representation to the resolution of the device, and then tries to match it to the color capability of the device. The default is YES. Returns **self**.

See also: – **isColorMatchPreferred**

setDataRetained:

– **setDataRetained:**(BOOL)*flag*

Determines whether the NXImage retains the data needed to render the image. The default is NO. If the data is available in a section of the application executable or in a file that won't be moved or deleted, or if responsibility for drawing the image is delegated to another object with a custom method, there's no reason for the NXImage to retain the data. However, if the NXImage reads image data from a stream, you may want to have it keep the data itself.

See also: – **isDataRetained**

setDelegate:

– **setDelegate:***anObject*

Makes *anObject* the delegate of the NXImage. Returns **self**.

See also: – **delegate**

setEPSUsedOnResolutionMismatch:

– **setEPSUsedOnResolutionMismatch:**(BOOL)*flag*

Determines whether EPS representations will be preferred when there are no representations that match the resolution of the device. The default is NO. Returns **self**.

See also: – **isEPSUsedOnResolutionMismatch**

setFlipped:

– **setFlipped:**(BOOL)*flag*

Determines whether the polarity of the y-axis is inverted when drawing an image. If flag is YES, the image will have its coordinate origin in the upper left corner and the positive y-axis will extend downward. This method affects only the coordinate system used to draw the image, whether through a method assigned with the **useDrawMethod:object:** method or directly by focusing on a representation. It doesn't affect the coordinate system for specifying portions of the image for methods like **composite:fromRect:toPoint:** or **initFromImage:rect:**.

See also: – **isFlipped**

setMatchedOnMultipleResolution:

– **setMatchedOnMultipleResolution:**(BOOL)*flag*

Determines whether image representations with resolutions that are exact multiples of the resolution of the device are considered to match the device. The default is NO. Returns **self**.

See also: – **isMatchedOnMultipleResolution**

setName:

– (BOOL)**setName:**(const char *)*string*

Sets *string* to be the name of the NXImage object and registers it under that name. If the object already has a name, that name is discarded. If *string* is already the name of another object or if the receiving NXImage is one of the system bitmaps provided by the Application Kit, the assignment fails.

If successful in naming or renaming the receiver, this method returns YES. Otherwise it returns NO.

See also: + **findImageNamed:**, – **name**

setScalable:

– **setScalable:**(BOOL)*flag*

Determines whether representations with sizes that differ from the size of the NXImage will be scaled to fit. The default is NO.

Generally, representations that are created through NXImage methods (such as **useCacheWithDepth:** or **initFromSection:**) have the same size as the NXImage. However, a representation that's added with the **useRepresentation:** method may have a different size, and representations created from data that specifies a size (for example, the "ImageLength" and "ImageWidth" fields of a TIFF representation or the bounding box of an EPS representation) will have the size specified.

Returns **self**.

See also: – **isScalable**

setSize:

– **setSize:**(const NXSize *)*aSize*

Sets the width and height of the image. The size referred to by *aSize* should be in units of the base coordinate system. The size of an NXImage must be set before it can be used. Returns **self**.

The size of an NXImage can be changed after it has been used, but changing it invalidates all its caches and frees them. When the image is next composited, the selected representation must draw itself in an off-screen window to recreate the cache.

See also: – **getSize:**, – **initSize:**

setUnique:

– **setUnique:**(BOOL)*flag*

Determines whether each image representation will be cached in its own off-screen window or in a window shared with other images. If *flag* is YES, each representation is guaranteed to be in a separate window. If *flag* is NO, a representation can be cached together with other images, though in practice it might not be. The default is NO.

If an NXImage is to be resized frequently, it's more efficient to cache its representations in unique windows.

This method does not invalidate any existing caches. Returns **self**.

See also: – **isUnique**

unlockFocus

– **unlockFocus**

Balances a previous **lockFocus** or **lockFocusOn:** message. All successful **lockFocus** and **lockFocusOn:** messages (those that return YES) must be followed by a subsequent **unlockFocus** message. Those that return NO should never be followed by **unlockFocus**.

Returns **self**.

See also: – **lockFocus**, – **lockFocusOn:**

useCacheWithDepth:

– (BOOL)**useCacheWithDepth:**(NXWindowDepth)*depth*

Creates a representation of type NXCachedImageRep and adds it to the NXImage's list of representations. Initially, the representation is nothing more than an empty area equal to the size of the image in an off-screen window with the specified *depth*. You must focus on the representation and draw the image. The following code shows how an NXImage might be created with the same appearance as a View.

```
id myImage;
NXRect theRect;

[myView getFrame:&theRect];
myImage = [[NXImage alloc] initSize:&theRect.size];
[myImage useCacheWithDepth:NX_DefaultDepth]
if ( [myImage lockFocus] ) {
    [myView drawSelf:(NXRect *)0 :0];
    [myImage unlockFocus];
}
```

depth should be one of the following enumerated values, defined in the header file **appkit/graphics.h**:

NX_DefaultDepth
NX_TwoBitGrayDepth
NX_EightBitGrayDepth
NX_TwelveBitRGBDepth
NX_TwentyFourBitRGBDepth

If successful in adding the representation, this method returns YES. If the size of the image has not been set or the cache can't be created for any other reason, it returns NO.

useDrawMethod:inObject:

– (BOOL)**useDrawMethod:**(SEL)*aSelector* **inObject:***anObject*

Creates a representation of type NXCustomImageRep and adds it to the NXImage object's list of representations. *aSelector* should name a method that can draw the image in the NXImage object's coordinate system, and that takes a single argument, the **id** of the NXCustomImageRep. *anObject* should be the **id** of an object that can perform the method.

This type of representation allows you to delegate responsibility for creating an image to another object within the program.

This method returns YES if it's successful in creating the representation, and NO if it's not.

useFromFile:

– (BOOL)**useFromFile:**(const char *)*filename*

Creates an image representation from the data found in *filename*, which can be a full or relative path, and adds the representation to the receiving NXImage. The data must be of a recognizable type, either EPS or TIFF. If the size of the NXImage has not yet been set, it will be set from information found in the TIFF fields or from the EPS bounding box comment.

If a representation can be added to the NXImage, this method returns YES. If not, it returns NO. In the current implementation, it may return YES even if the *filename* file doesn't exist or it contains bad data.

If *filename* contains data specifying more than one image, a separate representation is added for each one.

See also: – **initFromFile:**

useFromSection:

– (BOOL)**useFromSection:**(const char *)*name*

Creates an image representation from the data found in the *name* section of the __EPS or __TIFF segment of the application executable, and adds the representation to the NXImage. The data must be of a recognizable type, either EPS or TIFF. If the size of the NXImage has not yet been set, it will be set from information found in the TIFF fields or from the EPS bounding box comment.

If *name* includes a ".tiff" extension, this method looks in the __TIFF segment for a TIFF representation of the image; if *name* includes a ".eps" extension, it looks in the __EPS segment for an EPS representation. If *name* has neither extension, both segments are searched after adding the appropriate extension. Failing to find a section that matches the extended name, both segments are searched again for a section that matches *name* alone, without the extensions.

If no section is found that matches *name*, with or without the extension, this method searches for *name*.tiff and *name*.eps files in the directory where the application executable resides.

If sections that match the name are found in both the __EPS and __TIFF segments (or both ".eps" and ".tiff" files are found), this method creates both EPS and TIFF representations for the image. If the data in a section or file specifies more than one image, a separate representation is created for each one.

This method returns YES if a representation can be added to the NXImage, and NO if not. In the current implementation, it may return YES even if the section matching *name* contains bad data or no such section can be found.

See also: – **initFromSection:**

useRepresentation:

– (BOOL)**useRepresentation:**(NXImageRep *)*imageRep*

Adds *imageRep* to the receiving NXImage object's list of representations. If successful in adding the representation, this method returns YES. If not, it returns NO.

Any representation that's added by this method will belong to the NXImage and will be freed when the NXImage is freed. Representations can't be shared among NXImages.

See also: – **representationList**

write:

– **write:**(NXTypedStream *)*stream*

Writes the NXImage and all its representations to the typed stream *stream*.

See also: – **read:**

writeTIFF:

– **writeTIFF:**(NXStream *)*stream*

Writes TIFF data for the representation that best matches the display device with the deepest frame buffer to *stream*. This method is a shorthand for **writeTIFF:allRepresentations:** with a *flag* of NO. Returns **self**.

writeTIFF:allRepresentations:

– **writeTIFF:**(NXStream *)*stream*
  allRepresentations:(BOOL)*flag*

Writes TIFF data for the representations to *stream*. If *flag* is YES, data will be written for each of the representations. If *flag* is NO, data will be written only for the representation that best matches the display device with the deepest frame buffer. Returns **self**.

If *stream* is positioned anywhere but at the beginning of the stream, this method will append the representation(s) it writes to the TIFF data it assumes is already in the stream. To do this, it must be able to read the TIFF header from the stream. Therefore, the stream must be opened for NX_READWRITE permission.

METHOD IMPLEMENTED BY THE DELEGATE

imageDidNotDraw:inRect:

– (NXImage *)**imageDidNotDraw:***sender* **inRect:**(NXRect *)*aRect*

Implemented by the delegate to respond to a message sent by the *sender* NXImage when the *sender* was unable, for whatever reason, to composite its image. The delegate can return another NXImage to draw in the *sender*'s place. If not, it should return **nil** to indicate that *sender* should give up the attempt at drawing the image.

NXImageRep

INHERITS FROM	Object
DECLARED IN	appkit/NXImageRep.h

CLASS DESCRIPTION

NXImageRep is an abstract superclass for objects that know how to render an image. Each of its subclasses defines an object that can draw an image from a particular kind of source data. There are four subclasses defined in the Application Kit:

Subclass	**Source Data**
NXBitmapImageRep	Tag Image File Format (TIFF) and other bitmap data
NXEPSImageRep	Encapsulated PostScript code (EPS)
NXCustomImageRep	A delegated method that can draw the image
NXCachedImageRep	A rendered image, usually in an off-screen window

An NXImageRep can be used simply to render an image, but is more typically used indirectly, through an NXImage object. An NXImage manages a group of representations, choosing the best one for the current output device.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in NXImageRep	NXSize	size;

size	The size of the image in screen pixels.

METHOD TYPES

Setting the size of the image	– setSize:
	– getSize:

Specifying information about the representation

– setNumColors:
– numColors
– setAlpha:
– hasAlpha
– setBitsPerSample:
– bitsPerSample
– setPixelsHigh:
– pixelsHigh
– setPixelsWide:
– pixelsWide

Drawing the image	– draw – drawAt: – drawIn:
Archiving	– read: – write:

INSTANCE METHODS

bitsPerSample

– (int)**bitsPerSample**

Returns the number of bits used to specify a single pixel in each component of the data. If the image isn't specified by pixel values, but is device-independent, the return value will be NX_MATCHESDEVICE.

See also: – **setBitsPerSample:**

draw

– (BOOL)**draw**

Implemented by subclasses to draw the image at location (0.0, 0.0) in the current coordinate system. Subclass methods return YES if the image is successfully drawn, and NO if it isn't. This version of the method simply returns YES.

See also: – **drawAt:**, – **drawIn:**

drawAt:

– (BOOL)**drawAt:**(const NXPoint *)*point*

Translates the current coordinate system to the location specified by *point* and has the receiver's **draw** method draw the image at that point.

This method returns NO without translating or drawing if the size of the image has not been set. Otherwise, it passes through the return of the **draw** method, which indicates whether the image is successfully drawn.

The coordinate system is not restored after it has been translated.

See also: – **draw**, – **drawIn:**

drawIn:

– (BOOL)**drawIn:**(const NXRect *)*rect*

Draws the image so that it fits inside the rectangle referred to by *rect*. The current coordinate system is first translated to the point specified in the rectangle and is then scaled so the image will fit within the rectangle. The receiver's **draw** method is then invoked to draw the image.

This method returns NO without translating, scaling, or drawing if the size of the image has not been set. Otherwise it passes through the return of the **draw** method, which indicates whether the image is successfully drawn.

The previous coordinate system is not restored after it has been altered.

See also: – **draw**, – **drawAt:**

getSize:

– **getSize:**(NXSize *)*theSize*

Copies the size of the image to the structure referred to by *theSize*, and returns **self**. The size is provided in units of the base coordinate system.

See also: – **setSize:**

hasAlpha

– (BOOL)**hasAlpha**

Returns YES if the receiver has been informed that the image has a coverage component (alpha), and NO if not.

See also: – **setAlpha:**

numColors

– (int)**numColors**

Returns the number of color components in the image. For example, the return value will be 4 for images specified by cyan, magenta, yellow, and black (CMYK) or any other four components. It will be 3 for images specified by red, green, and blue (RGB), hue, saturation, and brightness (HSB), or any other three components. And it will be 1 for images that use only a gray scale. NX_MATCHESDEVICE is a meaningful return value for representations that vary their drawing depending on the output device.

See also: – **setNumColors:**

pixelsHigh

– (int)**pixelsHigh**

Returns the height of the image in pixels, as specified in the image data. If the image isn't specified by pixel values, but is device-independent, the return value will be NX_MATCHESDEVICE.

See also: **– setPixelsHigh:**

pixelsWide

– (int)**pixelsWide**

Returns the width of the image in pixels, as specified in the image data. If the image isn't specified by pixel values, but is device-independent, the return value will be NX_MATCHESDEVICE.

See also: **– setPixelsWide:**

read:

– **read:**(NXTypedStream *)*stream*

Reads the NXImageRep from the typed stream *stream*.

See also: **– write:**

setAlpha:

– **setAlpha:**(BOOL)*flag*

Informs the NXImageRep whether the image has an alpha component. *flag* should be YES if it does, and NO if it doesn't. Returns **self**.

See also: **– hasAlpha**

setBitsPerSample:

– **setBitsPerSample:**(int)*anInt*

Informs the NXImageRep that the image has *anInt* bits of data for each pixel in each component. If the image isn't specified by pixel values, but is device-independent, *anInt* should be NX_MATCHESDEVICE. Returns **self**.

See also: **– bitsPerSample**

setNumColors:

– **setNumColors:**(int)*anInt*

Informs the NXImageRep that the image has *anInt* number of color components. For color images with cyan, magenta, yellow, and black (CMYK) components, *anInt* should be 4, for color images with red, green, and blue (RGB) components, it should be 3, and for images that use only a gray scale, it should be 1. The alpha component is not included. NX_MATCHESDEVICE could be a meaningful value, if the representation varies its drawing depending on the output device. Returns **self**.

See also: – **numColors**

setPixelsHigh:

– **setPixelsHigh:**(int)*anInt*

Informs the NXImageRep that the data specifies an image *anInt* pixels high. If the image isn't specified by pixel values, but is device-independent, *anInt* should be NX_MATCHESDEVICE. Returns **self**.

See also: – **pixelsHigh**

setPixelsWide:

– **setPixelsWide:**(int)*anInt*

Informs the NXImageRep that the data specifies an image *anInt* pixels wide. If the image isn't specified by pixel values, but is device-independent, *anInt* should be NX_MATCHESDEVICE. Returns **self**.

See also: – **pixelsWide**

setSize:

– **setSize:**(const NXSize *)*aSize*

Sets the size of the image in units of the base coordinate system, and returns **self**. This determines the size of the image when it's rendered; it's not necessarily the same as the width and height of the image in pixels as specified in the image data.

See also: – **getSize:**

write:

– **write:**(NXTypedStream *)*stream*

Writes the NXImageRep to the typed stream *stream*.

See also: – **read:**

CONSTANTS AND DEFINED TYPES

```
/*
 * NX_MATCHESDEVICE indicates a value that's variable, depending
 * on the output device. It can be passed to the setNumColors:,
 * setBitsPerSample:, setPixelsWide:, and setPixelsHigh: methods,
 * and is returned by their counterparts.
 */
#define NX_MATCHESDEVICE    (0)


/*
 * Names of segments
 */
#define NX_EPSSEGMENT  "__EPS"
#define NX_TIFFSEGMENT "__TIFF"
#define NX_ICONSEGMENT "__ICON"
```

NXJournaler

INHERITS FROM Object

DECLARED IN appkit/NXJournaler.h

CLASS DESCRIPTION

The NXJournaler class defines an object that lets an application record and play back events and sounds, a process called *journaling*. By using an NXJournaler object, an application can journal events flowing to one or more applications—including itself. Optionally, sound can be recorded synchronously with the events. Later, the recorded events and sound can be played back, reenacting the activities as they occurred during the recording. With journaling, you can implement event-based macros or complete self-running demonstrations for your application. See the **ShowAndTell** application in **/NextDeveloper/Demos** for an example of journaling.

Journaling is initiated by creating a new NXJournaler object and sending it a **setEventStatus:soundStatus:eventStream:soundfile:** message. The status arguments may have the values NX_STOPPED, NX_PLAYING, and NX_RECORDING. The event stream argument is a stream to record to or play back from. If you're recording, any data in the stream will be overwritten. It's not currently possible to add to the end of an existing event stream. The sound file argument is the name of a sound file to record to or play back from.

When recording, by default all events going to any application are captured. Sometimes, you may not want certain applications to be recorded. For example, you might want to prevent the application that's recording the journal from being recorded. There are two ways to control this: with the defaults system and by sending a **setJournalable:** message to the Application object. Of the two, the defaults system is the more general.

To use the defaults system to control journaling, add this code to the **initialize** method of the object that will be controlling the journaling:

```
+ initialize
{
    static NXDefaultsVector myDefaults = {
        {"NXAllowJournaling", "NO"},
        {NULL}};
    NXRegisterDefaults([NXApp appName], myDefaults);
    return self;
}
```

This will prevent the application that contains the object from being journaled unless a user overrides the default for that application in the user's default database.

Users can also disallow journaling of any given application by adding an entry to their defaults database for that application. This would be done by entering the following command line in a Terminal window:

```
dwrite applicationName NXAllowJournaling NO
```

A less common way of allowing or disallowing journaling in an application is to send a **setJournalable:** message to the Application object. This allows more precise runtime control over journaling in that application.

Event recording may be aborted by clicking the right mouse button while holding down the Alternate key. (Note: For this to work, you must have the right mouse button enabled in the Preferences application.) Event playback can be aborted by typing a character with any key on the keyboard.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in NXJournaler	(none)	

METHOD TYPES

Initializing and freeing an NXJournaler	– init – free
Controlling journaling	– setEventStatus: soundStatus: eventStream: soundfile: – getEventStatus: soundStatus: eventStream: soundfile: – setRecordDevice: – recordDevice
Identifying associated objects	– speaker – listener – setDelegate: – delegate

delegate

– **delegate**

Returns the NXJournaler's delegate.

See also: – **setDelegate:**

free

– **free**

Frees the NXJournaler. Send this message to an NXJournaler after you're completely done with it.

getEventStatus:soundStatus:eventStream:soundfile:

– **getEventStatus:**(int *)*eventStatusPtr*
  soundStatus:(int *)*soundStatusPtr*
  eventStream:(NXStream **) *streamPtr*
  soundfile:(char **)*soundfilePtr*

Provides status information about the NXJournaler. Values returned at *eventStatusPtr* and *soundStatusPtr* can be NX_PLAYING, NX_RECORDING, or NX_STOPPED. *streamPtr* is the address of a pointer to the event stream. *soundfilePtr* is the address of a pointer to the name of the sound file. Any of the arguments may be NULL if you don't want that piece of information. Returns **self**.

See also: – **setEventStatus:soundStatus:eventStream:soundfile:**

init

– **init**

Initializes a newly allocated NXJournaler object. The delegate of the new object is **nil**. This is the designated initializer for an NXJournaler object. Returns **self**.

listener

– **listener**

Returns the listener used by the NXJournaler to communicate with other applications.

See also: – **speaker**

recordDevice

– (int)**recordDevice**

Returns whether sound is recorded from the CODEC microphone or from the DSP. The return value is either NX_CODEC or NX_DSP.

See also: – **setRecordDevice:**

setDelegate:

– **setDelegate:***anObject*

Sets the delegate used by the NXJournaler. The delegate is sent the method **journalerDidEnd:** when either playing or recording the journal finishes. If the journal was aborted, the delegate will first receive the message **journalerDidUserAbort:**. Returns **self**.

See also: – **delegate**

setEventStatus:soundStatus:eventStream:soundfile:

– **setEventStatus:**(int)*eventStatus*
    soundStatus:(int)*soundStatus*
    eventStream:(NXStream *)*stream*
    soundfile:(const char *)*soundfile*

Controls the recording and playback of events and sounds. This is the main control point of the NXJournaler. The arguments *eventStatus* and *soundStatus* may be independently set to NX_STOPPED, NX_PLAYING, NX_RECORDING. By setting *eventStatus* to NX_RECORDING and *soundStatus* to NX_STOPPED, it's possible to record events without the sound. By setting *eventStatus* to NX_PLAYING and *soundStatus* to NX_RECORDING, it's possible to dub new sound over an existing event track.

The *stream* argument is the stream to record events to or playback events from. When recording, any preexisting data in the stream will be overwritten. It's not currently possible to record onto the end of an existing event stream.

The *soundfile* argument is the name of the file to record sound to or playback sound from.

See also: – **getEventStatus:soundStatus:eventStream:soundfile:**

setRecordDevice:

– **setRecordDevice:**(int)*device*

Sets whether sound is recorded from the CODEC microphone (the default device) or from the DSP. The constants NX_CODEC and NX_DSP can be used to specify the device.

See also: – **recordDevice**

speaker

– **speaker**

Returns the speaker used by the NXJournaler to communicate with the other applications.

See also: – **listener**

METHODS IMPLEMENTED BY THE DELEGATE

journalerDidEnd:

– **journalerDidEnd:***journaler*

Responds to a message informing the delegate that recording or playback of the journal is finished or has been aborted.

See also: – **journalerDidUserAbort:**

journalerDidUserAbort:

– **journalerDidUserAbort:***journaler*

Responds to a message informing the delegate that the user has aborted the recording or playback session. A **journalerDidUserAbort:** message is sent when the NXJournaler in the controlling application receives notice from one of the controlled applications that the user has generated an abort event during recording or playback. The delegate receives this message just before the NXJournaler stops the recording or playback.

See also: – **journalerDidEnd:**

CONSTANTS AND DEFINED TYPES

```
/* NX_JOURNALEVENT subtypes */
#define NX_WINDRAGGED     0
#define NX_MOUSELOCATION  1
#define NX_LASTJRNEVENT   2

/* Window encodings in .evt file */
#define NX_KEYWINDOW        (-1)
#define NX_MAINWINDOW       (-2)
#define NX_MAINMENU         (-3)
#define NX_MOUSEDOWNWINDOW  (-4)
#define NX_APPICONWINDOW    (-5)
#define NX_UNKNOWNWINDOW    (-6)

/* Values for eventStatus and soundStatus */
#define NX_STOPPED      (0)
#define NX_PLAYING      (1)
#define NX_RECORDING    (2)

/* Values for recordDevice */
#define NX_CODEC    0
#define NX_DSP      1

#define NX_JOURNALREQUEST    "NXJournalerRequest"

typedef struct {
    int                 version;
    unsigned int        offsetToAppNames;
    unsigned int        lastEventTime;
    unsigned int        reserved1;
    unsigned int        reserved2;
}NXJournalHeader;
```

NXSplitView

INHERITS FROM	View : Responder : Object
DECLARED IN	appkit/NXSplitView.h

CLASS DESCRIPTION

The NXSplitView class defines an object that lets several Views share a region within a window. The NXSplitView resizes its subviews so that each subview is the same width as the NXSplitView, and the total of the subviews' heights is equal to the height of the NXSplitView. The NXSplitView positions its subviews so that the first subview is at the top of the NXSplitView, and each successive subview is positioned below. The user can set the height of two subviews by moving a horizontal bar called the *divider*, which makes one subview smaller and the other larger.

To add a View to an NXSplitView, you use the **addSubview:** View method. When the NXSplitView is displayed (as a result of a sending a display message, or because it was resized), it checks to see if its subviews are properly tiled. If not, it attempts to invoke the **splitView:resizeSubviews:** delegate method. If the delegate doesn't respond to this message, the **adjustSubviews** method is invoked to yield the default tiling behavior. If you want to set the height of a single subview to a specific value, you can simply set the height of its frame rectangle to that value. Remember, however, that the sum of the heights of the subviews plus the sum of the heights of the dividers must equal the frame height of the NXSplitView; otherwise, the NXSplitView will retile (and possibly resize) all its subviews. You can get the height of a divider with the **dividerHeight** method.

When a mouse event occurs in an NXSplitView, the NXSplitView determines if the event occurred in one of the dividers. If so, the NXSplitView determines the limits for the divider's travel, allows the delegate to limit the travel, and tracks the mouse to allow the user to drag the divider within the previously set limits. If the divider gets repositioned, the NXSplitView resizes the two affected subviews, informs the delegate that subviews were resized, and displays the affected Views and divider.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from View	NXRect	frame;
	NXRect	bounds;
	id	superview;
	id	subviews;
	id	window;
	struct __vFlags	vFlags;

Declared in NXSplitView	id	delegate;

delegate	The object that receives notification messages from the NXSplitView.

METHOD TYPES

Initializing a new NXSplitView	– initFrame:
Handling Events	– mouseDown: – acceptsFirstMouse
Managing component Views	– adjustSubviews – resizeSubviews: – dividerHeight – drawSelf:: – drawDivider: – setAutoresizeSubviews:
Assigning a delegate	– delegate – setDelegate:

INSTANCE METHODS

acceptsFirstMouse

– (BOOL) **acceptsFirstMouse**

Overrides the View method to allow the NXSplitView to respond to the mouse event that made its window the key window. Returns YES.

See also: – **acceptsFirstMouse** (View)

adjustSubviews

– **adjustSubviews**

Adjusts the heights of the NXSplitView's subviews so that the total of the subviews' heights fill the NXSplitView. The subviews are resized proportionally; the size of a subview relative to the other subviews doesn't change. This method is invoked if the NXSplitView's delegate doesn't respond to a **splitView:resizeSubviews:** message. Returns **self**.

See also: – **setDelegate:**, – **splitView:resizeSubviews:** (delegate), – **setFrame:** (View)

delegate

– **delegate**

Returns the NXSplitView's delegate.

See also: – **setDelegate:**

dividerHeight

– (NXCoord)**dividerHeight**

Returns the height of the divider. You can override this method to change the divider's height, if necessary; the value that this method returns is used as the divider's height.

See also: – **drawDivider:**

drawDivider:

– **drawDivider:**(const NXRect *)*aRect*

Draws a divider between two of the NXSplitView's subviews. *aRect* describes the entire divider rectangle in the NXSplitView's coordinates, which are flipped. The default implementation simply composites an image to the center of *aRect*; if you override this method and use a different icon to identify the divider, you may want to change the height of the divider. Returns **self**.

See also: – **dividerHeight**, – **drawSelf::**, + **findImageNamed:** (NXImage), – **composite:toPoint:** (NXImage)

drawSelf::

– **drawSelf:**(const NXRect *) *rects* **:**(int)*rectCount*

Draws the NXSplitView. This method first checks all the NXSplitView's subviews to ensure that they are positioned properly: Each subview should be the width of the NXSplitView and butted against the left edge of its frame rectangle. Each subview should also be butted against the divider for the previous subview. If the subviews aren't positioned properly, this method invokes **resizeSubviews:** to reposition and resize the subviews. This method then fills the NXSplitView's background area and invokes the **drawDivider:** method one or more times to draw all the required dividers. This method is invoked by the View methods for display; you shouldn't invoke this method directly. Returns **self**.

See also: – **drawDivider:**, – **resizeSubviews:**, – **display:** (View)

initFrame:

– **initFrame:**(const NXRect *)*frameRect*

Initializes the NXSplitView, which must be a newly allocated NXSplitView instance. The NXSplitView's frame rectangle is made equivalent to that pointed to by *frameRect*. If *frameRect* is NULL the default frame containing all zeros is unaltered. The NXSplitView's coordinate system is flipped so that its origin is at its upper left corner, and a flag is set so the NXSplitView automatically resizes its subviews when it's resized. This method is the designated initializer for the NXSplitView class. Returns **self**.

See also: – **setAutoresizeSubviews:** (View)

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

Overrides the Responder method so that the user can resize the NXSplitView's subviews. If the mouse-down event occurs in one of the NXSplitView's dividers, the NXSplitView determines the limits within which the divider can be dragged. It then gives the delegate the opportunity to modify the divider's minimum and maximum limits. This method then tracks the mouse to allow the user to resize the subviews within the previously set limits. It then resizes the appropriate subviews, informs the delegate that the subviews were resized, and displays the appropriate area of the NXSplitView and its subviews. Returns **self**.

See also: – **splitView:getMinY:maxY:ofSubviewAt:** (delegate), – **splitViewDidResizeSubviews:** (delegate), – **setFrame:** (View)

resizeSubviews:

– **resizeSubviews:**(const NXSize *)*oldSize*

Ensures that the NXSplitView's subviews are properly sized to fill the NXSplitView. If the delegate implements the **splitView:resizeSubviews:** method, that method is invoked to resize the subviews; otherwise, the **adjustSubviews** method is invoked to resize the subviews. In either case, this method then informs the delegate that the subviews were resized. *oldSize* is the previous bounds rectangle size. Returns **self**.

See also: – **splitView:resizeSubviews:** (delegate), – **adjustSubviews**, – **splitViewDidResizeSubviews:** (delegate), – **resizeSubviews:** (View)

setAutoresizeSubviews:

– **setAutoresizeSubviews:**(BOOL)*flag*

Overrides View's **setAutoresizeSubviews:** method to ensure that automatic resizing of subviews will not be disabled. You should never invoke this method. Returns **self**.

setDelegate:

– **setDelegate:***anObject*

Makes *anObject* the NXSplitView's delegate. The notification messages that the delegate can expect to receive are listed at the end of the NXSplitView class specifications. The delegate doesn't need to implement all the delegate methods. Returns **self**.

See also: – **delegate**

METHODS IMPLEMENTED BY THE DELEGATE

splitView:getMinY:maxY:ofSubviewAt:

– **splitView:***sender*
 getMinY:(NXCoord *)*minY*
 maxY:(NXCoord *)*maxY*
 ofSubviewAt:(int)*offset*

Allows the delegate to constrain the *y* coordinate limits of a divider when the user drags the mouse. This method is invoked before the NXSplitView begins tracking the mouse to position a divider. When this method is invoked, the limits have already been set and are stored in *minY* (the topmost limit) and *maxY* (the bottommost limit). You may further constrain the limits by setting the variables indicated by *minY* and *maxY*, but you cannot extend the divider limits. *minY* and *maxY* are specified in the NXSplitView's flipped coordinate system. The divider to be repositioned is indicated by *offset*; the divider between the first two subviews is indicated by an offset of zero.

See also: – **mouseDown:**

splitView:resizeSubviews:

– **splitView:***sender*
 resizeSubviews:(const NXSize *)*oldSize*

Allows the delegate to specify custom sizing behavior for the subviews of the NXSplitView. If the delegate implements this method, **splitView:resizeSubviews:** is invoked after the NXSplitView is resized; otherwise, **adjustSubviews** is invoked to retile the subviews. The old size of the NXSplitView is indicated by *oldSize*; the subviews should be resized to fill the NXSplitView's new frame rectangle size. You may find it convenient to use **NX_ADDRESS**() to get the address of the array of the **id**s of the subviews in order to step through the subview list.

See also: – **adjustSubviews**, – **dividerHeight**, – **setFrame:** (View), **NX_ADDRESS**()

splitViewDidResizeSubviews:

– splitViewDidResizeSubviews:*sender*

Informs the delegate that the sizes of some or all of the NXSplitView's subviews were changed. This method is invoked when the NXSplitView resizes all its subviews because its frame rectangle changed, and also after the NXSplitView resizes two subviews in response to the repositioning of a divider.

See also: **– resizeSubviews:**, **– mouseDown:**

Object Methods

INHERITS FROM	none (*Object is the root class.*)
DECLARED IN	appkit/Application.h

CLASS DESCRIPTION

The methods described here are declared in the Application Kit as additions to the Object class. However, the Object class itself is a "common class," not part of the Kit. For a description of the class and the other methods it defines, see "Object" in the "Common Classes" section above.

METHOD TYPES

Sending messages determined at run time	– perform:with:afterDelay:cancelPrevious:
Saying whether to run the Print panel	– shouldRunPrintPanel:
Services menu support	– readSelectionFromPasteboard: – writeSelectionToPasteboard:types:

INSTANCE METHODS

perform:with:afterDelay:cancelPrevious:

– **perform:**(SEL)*aSelector*
with:*anObject*
afterDelay:(int)*ms*
cancelPrevious:(BOOL)*flag*

Registers a timed entry to send an *aSelector* message to the receiver after a delay of at least *ms* milliseconds, and returns **self**. The *aSelector* method should not have a significant return value and should take a single argument of type **id**; *anObject* will be the argument passed in the message. Since timed entries are checked only when the application goes to get another event, program activity could delay the *aSelector* message well beyond *ms* milliseconds.

If *flag* is YES and another **perform:with:afterDelay:cancelPrevious:** message is sent to the same receiver to have it perform the same *aSelector* method, the first request to perform the *aSelector* method is canceled. Thus successive **perform:with:afterDelay:cancelPrevious:** messages can repeatedly postpone the *aSelector* message.

If *flag* is NO, each **perform:with:afterDelay:cancelPrevious:** message will cause another delayed *aSelector* message to be sent.

This method permits you to register an action in response to a user event (such as a click), but delay it in case subsequent events alter the environment in which the action would be performed (for example, if the click turns out to be double-click). It can also be used to delay a **free** message to an object until after the application has finished responding to the current event, or to postpone a message that updates a display until after a number of changes have accumulated.

See also: – **perform:with:** (Object)

readSelectionFromPasteboard:

– **readSelectionFromPasteboard:***pboard*

Implemented by subclasses to replace the current selection with data read from the Pasteboard object *pboard*. The data would have been placed in the pasteboard by another application in response to a remote message from the Services menu. A **readSelectionFromPasteboard:** message is sent to the same object that previously received a **writeSelectionToPasteboard:types:** message.

There's no default **readSelectionFromPasteboard:** method. The Application Kit declares a prototype for this method in the Object class, but doesn't implement it.

See also: – **writeSelectionToPasteboard:types:**

shouldRunPrintPanel:

#import <appkit/View.h>
– (BOOL)**shouldRunPrintPanel:***aView*

Implemented by subclasses to indicate whether the Print panel (or Fax panel) should be run before printing (or faxing) a View or a Window.

Printing requests are initiated by sending a View or Window a message to perform one of these two methods:

printPSCode: (View and Window)
smartPrintPSCode: (Window only)

Each method takes an **id** argument, which usually identifies the initiator of the print request (the object that sent the message). A **shouldRunPrintPanel:** message is sent back to that object, if the object can respond to the message. The *aView* argument identifies the View being printed.

If **shouldRunPrintPanel:** returns YES, the Print panel is run before printing begins. If it returns NO, the panel is not run, and the previous settings of the Print panel are used. The Print panel is also run if this method is not implemented.

Requests to fax a View or a Window can be initiated (by users) from within the Print panel. An application can also bypass the Print panel using either of the following two methods, which parallel the printing methods listed above:

faxPSCode: (View and Window)
smartFaxPSCode: (Window only)

Like the printing methods, these methods each take an **id** argument, and the argument is sent a **shouldRunPrintPanel:** message if it can respond. However, in this case, the value returned by **shouldRunPrintPanel:** indicates whether the Fax panel (not the Print panel) should be run.

There's no default implementation of the **shouldRunPrintPanel:** method. The Application Kit declares a prototype for this method in the Object class, but doesn't define it.

See also: – **printPSCode:** (View and Window), – **smartPrintPSCode:** (Window), – **faxPSCode:** (View and Window), – **smartFaxPSCode:** (Window)

writeSelectionToPasteboard:types:

– (BOOL)**writeSelectionToPasteboard:***pboard* **types:**(NXAtom *)*types*

Implemented by subclasses to write the current selection to the Pasteboard object *pboard*. The selection should be written as one or more of the data types listed in *types*. After writing the data, this method should return YES. If for any reason it can't write the data, it should return NO.

A **writeSelectionToPasteboard:types:** message is sent to the first responder when the user chooses a command from the Services menu, but only if the receiver didn't return **nil** to a previous **validRequestorForSendType:andReturnType:** message. After the data is written to the pasteboard, a remote message is sent to the application that provides the service the user requested. If the service provider supplies return data to replace the selection, the first responder will receive a subsequent **readSelectionFromPasteboard:** message.

There's no default **writeSelectionToPasteboard:types:** method. The Application Kit declares a prototype for this method in the Object class, but doesn't implement it.

See also: – **validRequestorForSendType:andReturnType:** (Responder), – **readSelectionFromPasteboard:**

OpenPanel

INHERITS FROM	SavePanel : Panel : Window : Responder : Object
DECLARED IN	appkit/OpenPanel.h

CLASS DESCRIPTION

The OpenPanel provides a convenient way for an application to query the user for the name of a file to open. It can only be run modally (the user should use the directory browser in the Workspace for non-modal opens). It allows the specification of certain types (i.e., file name extensions) of files to be opened.

Every application has one and only one OpenPanel, and the **new** method returns a pointer to it. Do not attempt to create a new OpenPanel using the methods **alloc** or **allocFromZone**; these methods are inherited from SavePanel, which overrides them to return errors if used.

See the class description for SavePanel for more information.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from Window	NXRect	frame;
	id	contentView;
	id	delegate;
	id	firstResponder;
	id	lastLeftHit;
	id	lastRightHit;
	id	counterpart;
	id	fieldEditor;
	int	winEventMask;
	int	windowNum;
	float	backgroundGray;
	struct _wFlags	wFlags;
	struct _wFlags2	wFlags2;
Inherited from Panel	(none)	

Inherited from SavePanel	id	form;
	id	browser;
	id	okButton;
	id	accessoryView;
	id	separator;
	char	*filename;
	char	*directory;
	const char	**filenames;
	char	*requiredType;
	struct _spFlags	spFlags;
	unsigned short	directorySize;
Declared in OpenPanel	char	**filterTypes;

filterTypes	File types allowed to open

METHOD TYPES

Creating and Freeing an OpenPanel	+ new + newContent:style:backing:buttonMask:defer: – free
Filtering files	– allowMultipleFiles:
Querying the chosen files	– filenames
Running the OpenPanel	– runModalForDirectory:file: – runModalForDirectory:file:types: – runModalForTypes:

CLASS METHODS

new

+ new

Creates, if necessary, and returns the shared instance of OpenPanel. Each application has just one instance of OpenPanel. This method is implemented to override the inherited **new** method to assure that only one instance of OpenPanel is created in an application.

newContent:style:backing:buttonMask:defer:

+ **newContent:**(const NXRect *)*contentRect*
  style:(int)*aStyle*
  backing:(int)*bufferingType*
  buttonMask:(int)*mask*
  defer:(BOOL)*flag*

Don't use this method, invoke **new** instead. This method is implemented to override the **newContent:style:backing:buttonMask:defer:** method inherited from SavePanel. Returns **self**.

See also: + **new**

INSTANCE METHODS

allowMultipleFiles:

– **allowMultipleFiles:**(BOOL)*flag*

If *flag* is YES, then the user can select more than one file in the browser. If multiple files are allowed, then the **filenames** method will be non-NULL only if one and only one file was selected. The **filenames** method will always return the selected files (even if only one file was selected). Note further that, though **filenames** always returns a fully-specified path, **filenames** never returns a fully-specified path (the files in the list are always relative to the path returned by **directory**). Returns **self**.

See also: **– filenames**

filenames

– (const char *const *)**filenames**

Returns a NULL terminated list of files (relative to the path returned by **directory**). This will be valid even if **allowMultipleFiles** is NO. This is the preferred way to get the name or names of any files that the user has chosen.

free

– **free**

Frees the storage used by the shared OpenPanel object and returns **nil**. The next time **new** is sent to the OpenPanel, it will be recreated. You probably never need to invoke this method since there is one shared instance of the OpenPanel.

See also: + **new**

runModalForDirectory:file:

– (int)**runModalForDirectory:**(const char *)*path* **file:**(const char *)*filename*

Initializes the panel to the file specified by *path* and *filename*, then displays it and begins its event loop. Returns **self**.

runModalForDirectory:file:types:

– (int)**runModalForDirectory:**(const char *)*path*
  file:(const char *)*filename*
  types:(const char *const *)*fileTypes*

Loads up the directory specified in *path* and optionally sets *filename* as the default file to open. *fileTypes* is a NULL-terminated list of suffixes (not including the "."'s) to be used to filter which files the user is given the opportunity to open. If the FIRST item in the list is a NULL, then all ASCII files will be included. Returns **self**.

runModalForTypes:

– (int)**runModalForTypes:**(const char *const *)*fileTypes*

Same as **runModalForDirectory:file:types:** except that the last directory from which a file was chosen is used. Returns **self**.

CONSTANTS AND DEFINED TYPES

```
/* Tags for the Views in a SavePanel */
#define NX_OPICONBUTTON     NX_SPICONBUTTON
#define NX_OPTITLEFIELD     NX_SPTITLEFIELD
#define NX_OPCANCELBUTTON   NX_SPCANCELBUTTON
#define NX_OPOKBUTTON       NX_SPOKBUTTON
#define NX_OPFORM           NX_SPFORM
```

NeXT Computer, Inc.
900 Chesapeake Drive
Redwood City, CA 94063

Printed in U.S.A.
2909.00
12/90