XEROX

Xerox GlobalView

Document Interfaces Toolkit User Guide

VP Series Applications

XEROX

Xerox GLOBALVIEW

Document Interfaces Toolkit User Guide

VP Series Applications
VP Series Reference Library, Version 3.0

Xerox Corporation
Product Education and Documentation (ESCN-215)
701 South Aviation Boulevard
El Segundo, California 90245

Publication number: 610E22850

Printed in the United States of America

Table of contents

List of tables

Introduction

This document is part of the Document Interfaces Toolkit. This toolkit provides the interface to the GlobalView environment and XNS services from user-generated C programs.

About this manual

The *Document Interfaces Toolkit User Guide* describes the procedure for interfacing to the GlobalView environment and to the XNS services by using the Document Interfaces Toolkit libraries. It presents general guidelines as well as examples to illustrate the process.

How chapters are organized

Chapter 1, "Overview", presents a general description of the Document Interfaces Toolkit.

Chapter 2, "Desktop functions", describes the desktop functions and shows how to manipulate desktop files and folders.

Chapter 3, "XString functions", describes the XString format and shows how to convert and compare XString characters and strings.

Chapter 4, "Document interfaces", presents a general workflow for the document-specific functions, showing procedures for creating and enumerating documents, graphics, and tables.

Chapter 5, "Document function examples", presents examples of using the document-specific functions.

Chapter 6, "XNS services", describes the XNS functions and gives examples of user-defined procedures that must be supplied to these functions.

Chapter 7, "XNS function examples", presents examples of using the XNS functions for the various XNS services.

Document conventions

The *Document Interfaces Toolkit User Guide* uses the following convention:

- Italics. VP application names, manual names, library names, and header file names appear in *italics*.
- Bold. Names of properties, options, C functions, notes, and warnings appear in **bold**.

Related documentation

The following materials are recommended reading:

- *Document Interfaces Toolkit Reference Manual.* This manual provides a complete list of the functions available within the Document Interfaces Toolkit, along with their syntax and argument definitions.
- *UNIX Interoperability User Guide.* This manual describes the UNIX environment and the GLOBALVIEW environment. It also describes how to work within each environment, switch between the two, and transfer information back and forth.
- *Character Code Standard.* This manual presents the standard used to assign multilingual textual information to a sequence of numerical codes.
- *Clearinghouse Protocol.* This manual presents a complete specification of the protocol between the Clearinghouse service and its clients.
- *Authentication Protocol.* This manual defines the complete specification of the protocol employed for interactions between clients and the Authentication Service.
- *Xerox Network Systems Architecture.* This manual describes the architecture of Xerox Network Systems, providing information on the standards and protocols that comprise the architecture.
- *Filing Protocol.* This manual describes the protocol for interaction between clients and file services.
- *Printing Protocol.* This manual describes the protocol used for communicating print requests from clients to print services.
- *Mailing Protocol.* This manual defines the set of protocols and related data formats for interaction between clients and the mail system.
- *XNS for UNIX System V.3 User Guide.* This manual provides information on the lower-level courier function calls.

1. Overview

This chapter describes general information on the Document Interfaces Toolkit. It is assumed that the user has some understanding of the C programming language, the GLOBALVIEW environment, the VP Document Editor, and the XNS protocol. If more information is needed in any of these areas, please refer to the appropriate Xerox documents listed in the Introduction of this manual, or contact Xerox for a list of available training courses.

What is the Document Interfaces Toolkit?

The Document Interfaces Toolkit allows programmers to interface to the GLOBALVIEW environment and Xerox Network Systems (XNS) from the UNIX environment. The Document Interfaces Toolkit consists of a set of C libraries which enable you to access ViewPak (VP) documents and some XNS services. With these libraries you may write C programs to read the contents of a VP document, create new documents, and append objects to the new document. Currently supported *objects* include text, tables, graphics, fields, and format characters. After you have created a new document, your C program can place the document onto the GLOBALVIEW hierarchical file structure, or use the XNS interfaces to file, mail, or print it to an XNS server.

The document-specific operations available through the Document Interfaces Toolkit are:

- Place a document on the GLOBALVIEW desktop
- Create a new document
- Create graphic frames, anchored Cusp button frames, and text frames
- Create new tables
- Set the properties of any objects added to a new document
- Append rows to existing tables
- Append frames, fields & text to documents
- Enumerate properties of existing document contents
- Enumerate the contents of a document
- Enumerate frames and their contents

The Document Interfaces Toolkit allows you to append information to the end of existing documents or append rows to existing tables. It does not allow you to insert or change objects anywhere else within an existing document. You may, however, read the contents of a document, extract the desired information, and then add this information into a new document. More details on this process are given in Chapter 4.

The XNS services available through the Document Interfaces Toolkit are:

- Authentication
- Clearinghouse
- Printing
- Filing
- Mailing
- Gateway

Authentication: The Authentication Service helps users and services determine each other's identity in order to establish a communication link. A *service* refers to a set of software resources that implements XNS protocols for the user.

Clearinghouse: The Clearinghouse is conceptually similar to a telephone directory. The user provides the Clearinghouse with object names and properties. The Clearinghouse returns the associated address for the names and properties. These addresses are used in remote procedure calls during the user's interaction with XNS.

Printing: The XNS printing application provides a set of procedures for transporting jobs and interpress files to the print service. Interpress is a Xerox standard for encoding the description of a document to be printed.

Filing: The Filing Service provides the user the ability to store, delete, and retrieve files on an XNS file service.

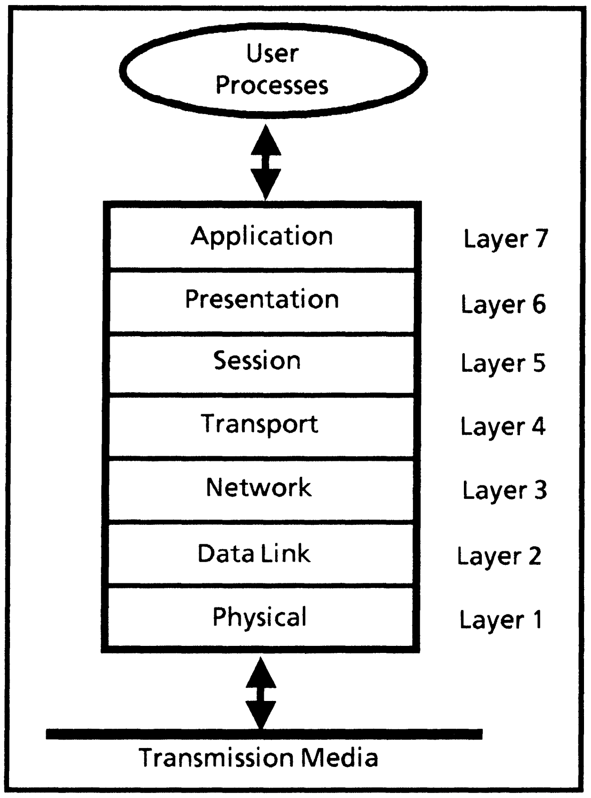
Mailing: The Mail Service provides posting and retrieving of electronic messages at Mail Services.

Gateway: The Gateway Service permits users to communicate with systems on different networks or with different communication architectures.

Note that these services allow users to access their corresponding servers. They do not provide the ability to implement your own Xerox compatible servers.

The XNS structure is layered, meaning that the various functions supported by XNS are divided into a series of levels. The more primitive tasks are located in the lower numbered layers while the higher layers are reserved for more sophisticated tasks. The services provided by the Document Interfaces Toolkit comprise the Application Support Environment within the application layer, which is called "layer 7". Figure 1-1 shows the relationship between the application layer and other layers within a layered network architecture. Note that the model shown in Figure 1-1 is an International Organization for Standardization (ISO) model. The XNS architecture groups some of the ISO model functions into fewer layers, but the general concept is the same. Please refer to the *Xerox Network Systems Architecture General Information Manual* for more details on the XNS structure.

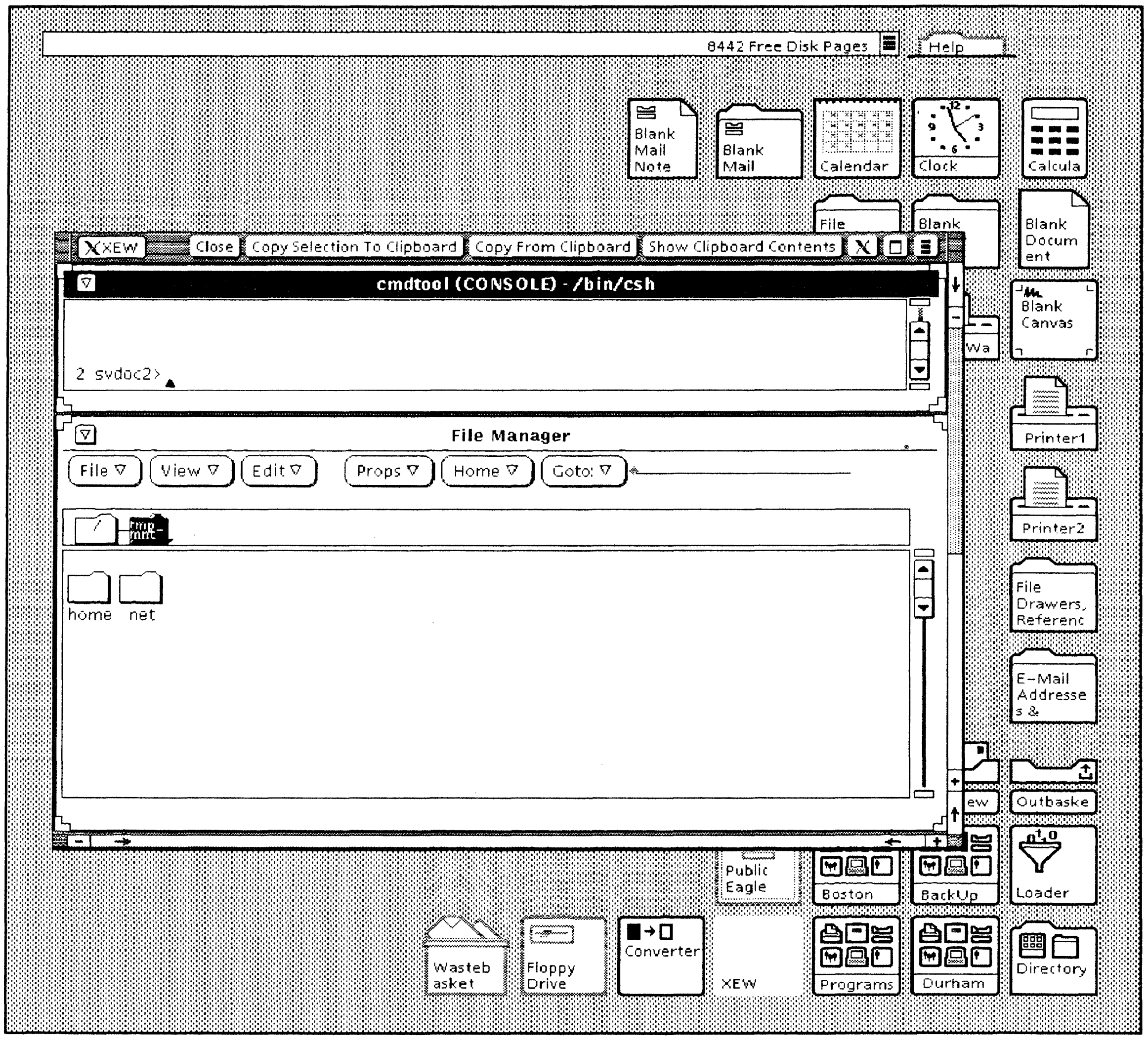
Figure 1-1. **Layers in a network architecture**



The desktop environment

C programs developed with the Document Interfaces Toolkit must be run within the GLOBALVIEW environment. The GLOBALVIEW desktop is the main work surface shown on your workstation screen. It displays icons that represent documents, file drawers, file folders, applications, etc. Double-clicking the mouse button on an icon will "open" that particular document or folder and reveal the document contents or a sublevel hierarchy of data, respectively. These icons depict the documents that the Document Interfaces Toolkit allow you to enumerate and to create. Figure 1-2 shows a typical GLOBALVIEW desktop containing icons and windows.

Figure 1-2. **GLOBALVIEW desktop**



The X Environment Window (XEW) may be described as a doorway from the GLOBALVIEW desktop to the UNIX environment. Figure 1-2 shows a snapshot of XEW in use. When opened, standard X Window programs compiled for a UNIX workstation may be executed from the desktop. In fact, it is just like having a UNIX X Window command tool shell loaded on your desktop. This window is one place where you will edit, compile, and invoke your Toolkit-specific programs.

Another place C programs may run is the *VP Window to UNIX Shell* application. When you work in the *VP Window to UNIX Shell* application, it is just like using the shell command interpreter in UNIX. The only difference is that *VP Window to UNIX Shell* provides only simple TTY access; no graphics or other advanced capabilities are available. This is also the main difference between *VP Window to UNIX Shell* and XEW. Please refer to the *UNIX Interoperability User Guide* for more information on GLOBALVIEW, XEW and *VP Window to UNIX Shell.*

System requirements

The following software applications must be present, enabled, and running on your workstation in order to use the Document Interfaces Toolkit:

- GLOBALVIEW
- VP Document Editor
- XEW or *VP Window to UNIX Shell*
- DocICToolkit

The Document Interfaces Toolkit adheres to the hardware and software system requirements set forth by the GLOBALVIEW package. Please refer to the GLOBALVIEW documents for more information on system requirements.

2. Desktop functions

The functions in the desktop portion of the Document Interfaces Toolkit allow for the copying, listing, and removal of files or folders on the GLOBALVIEW desktop. More importantly, desktop functions are the means by which applications written using Document Interfaces Toolkit C functions may interact with the files and folders on the desktop.

This chapter describes the manner in which files or folders may be placed on the desktop, removed from the desktop, or moved about the desktop.

Making and deleting folders

Making or deleting a folder is a very straightforward process. The functions **dsktp__makefolder()** and **dsktp__deletefolder()** are called to make and delete folders, respectively. They both require the same arguments:

- the name of the folder
- the full path name of the folder in which the folder is to be placed or removed
- the version number of the folder

To make a folder called "currentProjects" inside another folder called "bcDoc" the following code segment may be written:

```
main(argc, argv)
    char *argv[];
    ...
    {
       char *name;
       char *dstpath;
       XString fldrName, destPath;
       unsigned short vers;
       ...
       name = "currentProjects";
       dstpath = "bcDoc";
       fldrName = (XString) malloc((strlen( name) + 3 + 1)
       *2);
       destPath = (XString) malloc((strlen(dstpath) + 3 + 1)
       *2);
       XStrfromASC(fldrName, name);
       XStrfromASC(destPath, dstpath);
       dsktp__makefolder(fldrName, destPath, vers);
       ...
    }
```

Deleting a folder is very similar in format to making a folder, so the **dsktp__deletefolder()** function will not be described here.

Enumerating a folder

Listing, or enumerating, the contents of a folder is a two step process. The first step is to call **dsktp_enumerate()**. This function requires four arguments that specify:

- a criteria for determining which files and folders are to be returned and which are to be ignored
- a path at which to begin the search
- the levels of hierarchy to descend in the search for files
- a pointer to the buffer where the list is to be placed

The second step is to access the contents of the returned list.

For example, to obtain a list of all the contents of a desktop, you may use the following code:

```
main(argc, argv)
int argc;
char *argv[];
{
XString         pattern;
dsktp_reflist filelist;
char            ptrn[2];
...
ptrn[0] = '*';
ptrn[1] = '\0';

pattern = (XString)malloc(4);
XStrfromASC(pattern, ptrn);

if (dsktp_enumerate(pattern, NULL, 1, &filelist) )
    errorexit(fp);
...
```

Here the contents of the desktop are enumerated. The entire desktop is enumerated as indicated by the wildcard character "*" for the value of **ptrn[0]**. The second argument in the **dsktp_enumerate()** function is the path, and when set to NULL specifies the desktop.

If **dsktp_enumerate()** returns 0, the list should be present at the starting address location pointed to by the **&filelist** argument. To read the list, you may add the following lines of code:

```
...
for (i = 0; i < filelist.len; i + + ) {
    dsktp_getdocprops(filelist.refs[i], &props);
    asciiname = (char *)malloc(
      (sizeof(char)*XStrlen(props.name)) + 1);
    XStrtoASC(props.name, asciiname, subst);
    fprintf(fp, "Name is %s\n", asciiname);
}
...
```

Document reference handles

The preceding examples illustrate the usage of basic desktop functions. Another important aspect of desktop functions is file manipulation. This usually requires additional information — a document handle. A document reference handle is a data structure that points to the file, and may be obtained by a call to one of several functions, such as **dsktp__getdocref()** or **di__finish()**. Once a document reference handle is obtained, it may be passed as an argument to those Document Interfaces Toolkit functions that either initiate or terminate the document editing process.

The **dsktp__getdocref()** function is called first when manipulating a file that already exists on the desktop or in a folder on the desktop. The syntax for **dsktp__getdocref()** is:

```
dsktp__getdocref( name, vers, srcpath, ref)
```

where:

- **name** is the text string name of the document to be opened
- **vers** is an integer that specifies the version number of name
- **srcpath** is the text string location of name
- **ref** is the return value in which will be placed the document identifier

The following example illustrates a way in which a document reference handle may be obtained for a document. Once the document reference handle is retrieved, it can be passed to **di__start()**, **di__open()**, etc. to start or open the document.

```
main(argc, argv)
    char argv[];
    ....
    {
      ret__open ret;
      ret__ref ref;
      XString name[256];
      ...
      XStrfromASC(name, argv[1]);
      dsktp__getdocref(name, LASTVERS, NULL, &ref);
      di__open( ref, &ret);
      ...
    }
```

The arguments to the **dsktp__getdocref()** function in this example are:

- **name**, a variable for the document name, the value of which is specified upon invoking the application
- **LASTVERS**, a constant that specifies that the latest version of the document is to be accessed
- **NULL**, a constant that indicates the document is on the desktop, rather than nested within a folder
- **ref**, return value in which the document handle is placed

Once a document handle has been successfully obtained, the document may be manipulated in many ways. There are three

dsktp__*() functions available that manipulate documents. They are:

dsktp__copydoc()	copy an existing desktop file to a reserved buffer area
dsktp__deletedoc()	remove a document from off the desktop
dsktp__movedoc()	move document data from a buffer onto the desktop

Copying and moving documents

Copying a document is a two stage process. The first stage entails copying an existing document to a buffer. The second entails moving the buffer data onto the desktop. To copy a desktop document called "src__dat" to "bkup__dat", "src__dat" must first be copied to a buffer. To do this call **dsktp__copydoc()**:

```
main(argc, argv)
    char argv[];
{
    XString name[256];
    dsktp_docref ref;
    dsktop_docref new;
    ...
    dsktp_copydoc(ref, &new );
    ...
}
```

where:

- **ref** is the document handle returned by an earlier call to **dsktp__getdocref()**
- **&new** defines the structure into which the buffer document handle is to be placed.

At this point, the duplicate document only resides in the buffer. To create an icon for the duplicate document the buffer must be moved onto the desktop. The **dsktp__movedoc()** function is used to move the buffer. The syntax of this function is:

```
dsktp__movedoc( ref, dstpath, name, vers)
```

where:

- **ref** is the document handle that was returned by **dsktp__copydoc()**
- **dstpath** is the full path name of the folder in which the file is to go
- **name** is the text string the new document is to have as its name
- **vers** is the version number

Continuing with the previous example, the document in the buffer may be copied onto the desktop in the following manner:

```
main(argc, argv)
char *argv[];
...
{
    char *text;
    XString name;
    unsigned short vers;
    ret_sc ret;
    ret_fc ret2;
    int ok
    ...
    if (di_finish(ret.doc, NULL, NULL, &ret2)) {
      ...
      }
    text = "bcDoc.new"
    name = ( XString) malloc((strlen( text) + 1) *2);
    XStrfromASC( name, text);
    ok = dsktp_movedoc(&ret2.ref[0], NULL, name, &vers);
    if (ok) exit(1);
}
```

Please see the *Document Interfaces Toolkit Reference Manual* for more detailed information on the desktop functions.

3. XString Functions

The Document Interfaces Toolkit offers a set of string manipulation functions to perform tasks that are specific to the GLOBALVIEW environment. These functions are referred to collectively as XString functions. The string manipulation functions included in the Document Interfaces Toolkit are used primarily to convert character formats for transmission between UNIX (ASCII-based) and XString-based GLOBALVIEW interfaces and to allow C applications to control text processing from within the GLOBALVIEW environment.

This chapter describes how to use XString functions. Actual examples are included to further illustrate the application of these functions within a C program. Upon completing this chapter, you should be able to use XString functions to convert strings from Xerox Character Code formats to ASCII and back again, to copy strings, to compare strings, and more.

XString format

XStrings are in a special format that allow the support of a large number of languages. The format is described in the Xerox Character Code Standard. For most of the XString functions, there are C counterparts that provide similar functionality. The main difference is that XString functions operate on XStrings rather than simple C strings containing ASCII characters. There are functions which allow XStrings to be converted to ASCII strings and visa versa. The spelling of XString function names is often spelled identically to the conventional C counterparts, except that the XString function names are prefixed with an X. For example, **strcat()** and **XStrcat()**.

They are also very similar syntactically. That is, XString functions require the same arguments as their conventional C counterparts. The syntax of **strcat()** is:

```
strcat(s, ct)
```

The syntax of **XStrcat()** is:

```
XStrcat(xs1, xs2)
```

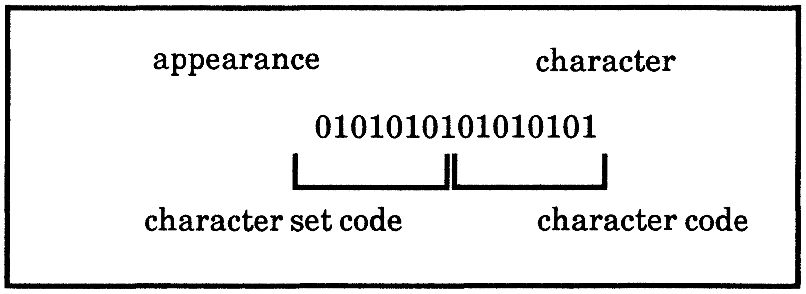
where, **ct** is the string to be concatenated to the end of **s** and **xs2** is the XString to be concatenated to the end of **xs1**.

Xerox Character Code Standard

An XString is a set of one or more characters that is structured in accordance to the *Xerox Character Code Standard.* All XStrings are represented as a sequence of numeric codes. These numeric codes may be used in such a way as to define characters, so that there is no ambiguity when communicating between various protocols or interchanging documents.

The *Xerox Character Code Standard* (XCCS) is used to define the meaning and the appearance of characters. Each character is expressed as a 16-bit entity having a numeric value between 0 and 65,535, inclusive. Each 16-bit character code can be viewed as consisting of two 8-bit bytes, where the high-order byte is the character set code and the low-order byte is the character's code within the character set.

Figure 3-1. **Character structure**



The range of numeric codes are described and defined in the *Character Code Standard* manual.

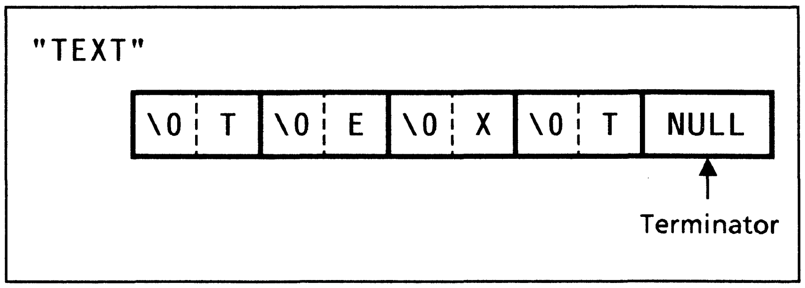
Some simple examples, using the 000_8 Character Set are used in this chapter.

XString functions are available that convert 8-bit encoded XStrings to 16-bit encoded XStrings, and visa versa, as described in the *Character Code Standard* manual. There are also functions that convert XStrings from ASCII to XString formats, and visa versa. Conversion to and from EUC (Extended Unix Code) or JIS (Japanese Industrial Standard) is not currently supported. Conversion to and from ISO 8859 is also not supported. ISO 8859 is a superset of ASCII that provides support for western European languages.

XString structure

An XChar is a character that adheres to the format specified in the XCCS. An XString is a simple array of XChar terminated by a 16-bit NULL code (0x0000).

Figure 3-2. **Structure of XString**



All XString creation and editing functions, except **XStrncpy()**, terminate the resulting XString with a NULL character. Furthermore, XCC8 (Xerox Character Code for 8-bit characters) is defined in the XCCS book as a data structure comprised of 8-bit encodings. XCC8 is analogous to *ByteSequence* in Mesa XString.

String Conversion

The C functions in the Document Interfaces Toolkit are used to interact with the GLOBALVIEW environment. Internally, the structure of text is different than that of regular ASCII text. When text is passed between ASCII-based and XString-based environments, it must be converted to the appropriate structure before the receiving environment is capable of processing it.

There are three text structures of interest: ASCII, XString, and XCC8. In ASCII, a character is described in one 8-bit byte. An XString character is described in two 8-bit bytes. An XCC8 character is the same as an XString character, except that it has been truncated into one 8-bit byte.

To pass an ASCII string into the GLOBALVIEW environment, it must be converted from ASCII to an XString format. The reverse is true when going from XString to ASCII. There are seven XString functions that either generate XChars or convert strings between ASCII and XCCS formats. The following table is a brief synopsis of those XString functions.

Table 3-1. **XString usage**

XString	Use
XCharset	Determine the character set to which an XChar belongs
XCharcode	Determine the code of an XChar
XCharmake	Make an XChar of a specific character set and code
XStrfromASC	Convert an ASCII string into an XString
XStrtoASC	Convert an XString into an ASCII string
XStrfromXCC8	Convert an XCC8 XString into a 16-bit XString
XStrtoXCC8	Convert a 16-bit XString into an XCC8 XString

The first three functions, **XCharset()**, **XCharcode()**, and **XCharmake()** are used to read and generate XChars. These functions are described in the section immediately following this one.

The two most important conversion functions are **XStrfromASC()** and **XStrtoASC()**. **XStrfromASC()** is used to convert an ASCII string to an XString format. **XStrtoASC()** is called to convert text from an XString to an ASCII format.

The syntax of **XStrfromASC()** is:

```
XStrfromASC(xs, s)
```

where **s** is a pointer to the ASCII string to be converted and **xs** is the return value which contains the XString equivalent of **s**. For example, to convert an ASCII text string into an XString text string, the following code segment may be written:

```
main(argc, argv)
int argc;
char *argv[];
...
XString name;
char *ascii;
...
ascii = argv[1];
name = (XString) malloc((strlen( ascii) + 1) * 2);
XStrfromASC(name, ascii);
```

The result of the preceding code segment is an XString variable, called **name**, that may be passed as an argument to other Document Interfaces Toolkit functions, such as **dsktp_movedoc()**.

The format of **XStrtoASC()** is:

```
XStrtoASC(xs, s)
```

where **xs** is the XString to be converted and **s** is the return value which contains the equivalent ASCII text.

An example of converting from XString back to ASCII is shown below:

```
...
XString    text_work;
XString    text;
ascii_text = (char *) malloc (XStrlen(text) + 1);
text_work = (XString) malloc (XStrlen(text))*2 + 2);
XStrtoASC(text_work, ascii_text);
...
```

XCC8 conversion

Two functions, **XStrfromXCC8()** and **XStrtoXCC8()** are used to convert strings between XString and XCC8 formats. The primary advantage of converting strings to an XCC8 format is that it compresses the strings into compact, 8-bit encoded strings.

XStrfromXCC8() is called to convert an XCC8 string to an XString format. The syntax of **XStrfromXCC8()** is:

```
XStrfromXCC8(xs, xcc8, len, prefix)
```

where **xs** is the storage area in which the converted XCC8 string will be placed, **xcc8** is the 8-bit encode string to be converted, **len** is the length in bytes of **xcc8**, and **prefix** specifies the character set of **xcc8**. **prefix** should be -1 if the first character of **xcc8** begins with a 16-bit code.

XStrtoXCC8() is called to convert an XString string to XCC8 format. The syntax of **XStrtoXCC8()** is:

```
XStrtoXCC8(xs, xcc8)
```

where **xs** is the XString to be converted and **xcc8** is the return value which contains the XCC8 equivalent of **xs**.

The following example shows a simple usage of **XStrtoXCC8()**:

```
...
#define BUFLEN 256
XChar       xs [BUFLEN];
unsigned char       xcc8 [BUFLEN*2];
...
len = XStrtoXCC8(xs, xcc8);
```

In the above example, the value of **xs** is translated into XCC8 format and is placed into the argument **xcc8**. **len**, the return value, returns the byte length of **xcc8**.

Note that converting strings directly between ASCII and XCC8 formats is not allowed. A string must be in an XString format before it may be converted to either ASCII or XCC8.

Character identification

When reading an XString, one of the first steps may be to determine the character set and the character code of the XChars in that XString. To determine the Character Set and the Character Code, calls are to be made to **XCharset()** and **XCharcode()** for each XChar in the XString. **XCharset()** is used to determine the Character Set (the most significant 8-bits). **XCharcode()** is used to determine the Character Code (the least significant 8-bits). The syntax of **XCharset()** is:

```
XCharset(xc);
```

The syntax of **XCharcode()** is:

```
XCharcode(xc);
```

The argument **xc** is of the type XChar, the value of which is the XChar for which the set or code is to be determined.

The following example illustrates one way of determining the Character Set and Character Code of a character:

```
#include <stdio.h>
#include <doctk/DocICProps.h>
...
main()
{
    ret_getpagedel ret;
    char cs,cc;
    if (dp_getpagedel(&ret)) {
        error_display("dp_getpagedel",0);
        return(-1);
    };
    cs = XCharset(ret.del);
    cc = XCharcode(ret.de);
    printf("Left (Set, Code) = (%x,%x) ", cs, cc);
...
```

XString functions

There are 18 XString functions that manipulate XStrings in a way similar to the way conventional C manipulates regular strings.

Table 3-2 is a brief synopsis of the XString functions that manipulate XStrings in a conventional C fashion. Note that the conventional C function listed is not necessarily identical to the corresponding XString function, but may be a close approximation.

Table 3-2. **C and XString function similarities**

C	XString	Use
strcat	XStrcat	Concatenate two XStrings
strncat	XStrncat	Concatenate a specific number of characters from one XString to the end of another
strcmp	Xstrcmp	Compare two XStrings for any variation, not taking into account the character case
strncmp	XStrncmp	Compare a specific number of characters in one XString against the same number of characters in another XString
	XStrcasecmp	Compare two XStrings, while taking into account the character case
	XStrncasecmp	Compare a specific number of characters in one XString against the same number of characters in another XString, while taking into account the character case
strcpy	XStrcpy	Copy an XString into a buffer
strncpy	XStrncpy	Copy a specific number of characters in an XString into a buffer
	XStrdup	Copy an XString into a buffer and return a pointer to the buffer
strlen	XStrlen	Determine the logical length of an XString
strcmp	XStrlexcmp	Lexicographically compare two XStrings using a sort order parameter to support multinational comparisons
strncmp	XStrnlexcmp	Lexicographically compare a specific number of characters in one XString against the same number of characters in another XString not accounting for character case

Table 3-2. **C and XString function similarities**

C	XString	Use
	XStrcaselexcmp	Lexicographically compare two XStrings, while taking into account the character case
strchr	XStrchr	Return a pointer to the first occurrence of a specified XChar in an XString
strrchr	XStrrchr	Return a pointer to the last occurrence of a specified XChar in an XString
strpbrk	XStrpbrk	Find the first occurrence of an XChar in an XString that also occurs in another XString and returns a pointer to it
strstr	XStrsch	Determine if an XString is wholly contained within another XString
strtok	XStrsep	Section an XString into tokens based upon a specified delimiter

Concatenating XStrings

XChars may be concatenated with other XChars to make an XString. The **XStrcat()** function is used to concatenate XStrings, much like the **strcat()** function in conventional C. **XStrcat()** takes two arguments: The first is the XChar or XString to which another XChar or XString is to be added. The second argument is the XChar or XString to be added. The syntax for **XStrcat()** is:

```
XStrcat(xs1, xs2);
```

where **xs2** is the XString to be appended to the end of XString **xs1**.

The following example appends the characters ".CV" to an XString:

```
...
XString name;
ascii1 = argv[1];
/*
The following line uses (strlen(ascii1) + 4) instead of
(strlen(ascii1) + 1) in order to allocate memory for the
additional 3 character extension
*/
name = (XString)malloc((strlen(ascii1) + 4) * 2);
name = XStrfromASC(name, ascii)
ascii = ".CV";
ext = (XString)malloc((strlen(ascii) + 1) * 2);
ext = XStrfromASC(ext, ascii);
name = XStrcat(name, ext);
...
```

The **XStrncat()** function is used to concatenate a specific number of XStrings in one XString to the end of another XString, like the conventional C function **strncat()**. The syntax for **XStrncat()** is:

```
XStrncat(xs1, xs2, n);
```

where **n** is the number of XStrings from **xs2** to be appended to the end of **xs1**.

XString comparison

Many occasions occur where an XString returned by a function must be evaluated by comparing it against another XString.

Some Document Interfaces Toolkit functions may be used to compare XChars or XStrings based upon a specified criteria. The criteria may notice or ignore character case, compare whole XStrings or partial XStrings, and/or compare XStrings lexicographically. A lexicographic comparison compares XStrings based upon the nationality of the language used to define the XChars in the respective XStrings.

Basic XString comparison

To initiate an exact comparison of two XStrings, the **XStrcmp()** function should be called. The syntax of **XStrcmp()** is:

```
XStrcmp(xs1, xs2);
```

where the XChars in **xs1** are compared the XChars in **xs2**.

To compare one XString, "abcdef", against another, "abcxyz", the following function call may be made:

```
...
char ascii1;
char ascii2;
XString xstr1;
XString xstr2;
int results;
ascii1 = 'abcdef';
ascii2 = 'abcxyz';
xstr1 = (XString)malloc((strlen( ascii1) + 1) * 2);
xstr2 = (XString)malloc((strlen(ascii2) + 1) * 2);
results = XStrcmp(xstr1, xstr2);
```

Once converted to an XString format, the XChars comprising the first XString, "abcdef", are compared on a one-by-one basis against the XChars in the second XString, "abcxyz". The fourth XChar in the second XString, "x", is noted as being different than the corresponding XChar in the first XString, "d". The value of XChar "d" is 0x0064(100) and that of "x" is 0x78(120). **XStrcmp()** then returns "d - x", which is -20 in decimal.

Referring to the Latin Alphabet Character Set 000_8 shown in the XCCS book, you will notice the octal and hexadecimal equivalents of each XChar. However, XChar comparisons are performed in decimal. The value returned by a call to **XStrcmp()** is the difference of subtracting the decimal value of an XChar in the second string from that of an XChar in the first XString. In the preceding example, subtracting "x", 120_{10}, from "d", 100_{10}, results in -20_{10}.

Note that whenever the return value is a positive integer, the first XString is numerically greater than the second XString. When the return value is negative, the second XString is numerically greater. When the return value is 0, the two XStrings are identical.

Partial XStrings may be compared. That is, a specific number of XChars in the first XString may be compared against an equivalent number of XChars in the second XString. The function **XStrncmp()** is used to compare a specified number of XChars in one XString against those of another XString.

Comparison is performed in the same manner as described for **XStrcmp()**, except that the process initiated by this function call stops when the specified number of XChars have been compared. For example, to compare the first 5 XChars of one XString, "abcdef", against the first 5 of another XString, "abcxyz", the following segment of code may be used:

```
...
char ascii1;
char ascii2;
XString xstr1;
XString xstr2;
int results;
int num;
ascii1 = 'abcdef';
ascii2 = 'abcxyz';
num = 5
xstr1 = (XString)malloc((strlen( ascii1) + 1) * 2);
xstr2 = (XString)malloc((strlen(ascii2) + 1) * 2);
results = XStrncmp(xstr1, xstr2, num);
```

The decimal sum of the first five XChars of the second XString, 535_{10}, is subtracted from that of the first five XChars in the first XString, 495_{10}. A difference of -40_{10} is returned. Thus, indicating that the sum total of the first five XChars in the second XString is greater than that of the first.

Ignoring case during comparison

The preceding functions are case-sensitive when comparing XStrings. Two additional functions are available in the Document Interfaces Toolkit that perform the same task as **XStrcmp()** and **XStrncmp()** but do so while ignoring the case of XChars. These functions are **XStrcasecmp()** and **XStnrcasecmp()**. The usage of these two functions is identical to that of **XStrcmp()** and **XStrncmp()**.

Like **XStrcmp()** and **XStrncmp()**, **XStrcasecmp()** and **XStrcasecmp()** compare the decimal equivalents of XChars and return the difference. When converting XChars to their decimal equivalents; however, lowercase XChars are assigned the decimal value of their uppercase equivalents. Thus, the value of the XChar "x" is no longer 120_{10}, but is 88_{10}, the value of the uppercase "X". So, when the return value of comparing two XStrings is 0, the two XStrings may not be identical but will be semantically the same.

Lexicographic comparison

Variations of the preceding functions are available which allow comparisons that take into account particular national sort orders used in some countries. They operate in the same manner as the other XString compare functions already described, except that the lexicographic compare functions take an additional argument, **sortorder**. The other functions use a simple numeric comparison.

The lexicographic compare functions are **XStrlexcmp()**, **XStrnlexcmp()**, and **XStrcaselexcmp()**. The syntax for these compare functions is:

```
XStrlexcmp(xs1, xs2, sortorder);

XStrnlexcmp(xs1, xs2, sortorder);

XStrncaselexcmp(xs1, xs2, sortorder, n);
```

where the XChars in **xs1** are compared against the XChars in **xs2**, based upon the value of **sortorder. n** is the number of XChars in **xs1** to be compared against the same number of XChars in **xs2**. **sortorder** is an enumerated type that specifies the nationality of interest. Valid nationalities are Standard, Danish, Swedish, and Spanish. Standard is the most commonly used of the sort orders. **sortorder** is explained in more detail in the *Document Interfaces Toolkit Reference Manual.*

4. Document interfaces

This chapter presents a general workflow for using the Document Interfaces Toolkit libraries. It describes how to create and read documents, graphics, and tables. The XNS toolkit library is described in Chapter 6 of this manual.

Document interface library

The document-specific portion of the Document Interfaces Toolkit consists of the C library *libdoctk.a* and the following header files:

Desktop.h
DocIC.h
DocICProps.h
GraphicsIC.h
Signals.h
TableIC.h
XString.h

The *Desktop.h* header file contains functions to enumerate, copy, delete, or create documents or folders on the GlobalView desktop. Those functions or properties prefixed with "dsktp__ " may be found in the *Desktop.h* header file.

The *DocIC.h* header file contains functions to create and enumerate document contents (e.g., text, fields, headings, footings, etc.). Those functions or properties prefixed with "di__" may be found in the *DocIC.h* header file.

The *DocICProps.h* header file contains functions and data types used to set the properties within GlobalView documents. Those functions or properties prefixed with "dp__" may be found in the *DocICProps.h* header file.

The *GraphicsIC.h* header file contains functions to create and enumerate CUSP button frames, anchored graphic frames, and nested graphic frames. Those functions or properties prefixed with "gi__" may be found in the *GraphicsIC.h* header file.

The *Signals.h* header file contains the declaration of **getsigno()** as well as error text descriptions of errors encountered during document manipulation. **getsigno()** is used to return error codes.

The *TableIC.h* header file contains functions to create a new table, enumerate the contents of a table, and add rows to a new or existing table. Those functions or properties prefixed with a "ti__" may be found in the *TableIC.h* header file.

The *XString.h* header file contains functions to manipulate characters and character strings, such as string copy, string comparison, string search, etc. Those functions or properties prefixed with an "X" may be found in the *XString.h* header file.

Error handling

Determining the cause of an error is initiated by a call to **getsigno()**, as shown in the example below.

```
if (di_apaframe(&to, type, &frame, contents, FALSE, FALSE,
    FALSE, FALSE, NULL, FALSE, &anc_ret)) {
    printf("Error during appending frame: Error number
        %d\n", getsigno())
    exit (-1);
};
```

The **getsigno()** function will return an integer number. It is up to the programmer to develop a routine that automatically appends the corresponding text associated with each error number. Otherwise, each time an error is encountered, the user must manually locate the text associated with each error number from the *Signals.h* header file or the reference manual. A general procedure for handling errors is:

```
...
if (di_XXX(...)) {print_error(getsigno()); exit(-1);}
...
void print_error(number)
int number;
{
switch(number){
    case 0x1000: printf("Doc_Container Full\n");break;
    case 0x1001: printf("Doc_Document Full\n");break;
    ...
    default: break;
    }
}
```

In addition, some of the functions return a status code that may be used to check for additional errors. For example, **di_start()** returns information into the structure **ret_sc**, which contains the following members:

```
di_doc doc;
di_heading lhead;
di_heading rhead;
di_footing lfoot;
di_footing rfoot;
di_numbering num;
di_scstat stat;
```

stat is the status code, which may have any of the following values:

SC_OK	Everything was fine
SC_DSKSP	There isn't enough disk space to perform the operation
SC_VM	There isn't enough contiguous virtual memory to create the document

You can use this status code in your program to check for other errors as in the example below:

```
di_start(PAGINATE, FALSE, FALSE, FALSE, ifoprops, iprops,
    ipgprops, NULL, &ret)
```

```
if (ret.stat != SC_OK) {
    fprintf (stderr, "ret_sc status = %d /n", ret.stat);
}
```

The preceding example checks to see if the **di__start()** function returned successfully. If not, then the code will print the error that **ret.stat** returned. The reference manual gives a complete description of the status code values, and which functions return them.

Using the *DocIC.h* header file

The *DocIC.h* header file provides functions to create or read the basic document structures, such as text, text tiles, fields, headers, footers, or frames. *DocIC.h* functions may be used to manipulate the contents of text frames only. Use *GraphicsIC.h* and *TableIC.h* functions to manipulate graphics frames or table frames, respectively.

Creating a document

To create a new document, the first step is to call either **di__start()** or **di__startap()**. This sets up the data structures for the new document and returns a pointer to an opaque type that represents the document, called **doc**.

The next step is to append information to the document by using various **di__ap*()** functions. Please refer to the chart in Table 4-1 for a listing of available **di__ap*()** functions. To append graphics or tables to the document, you must call the appropriate functions listed in *GraphicsIC.h* or *TableIC.h* before using the **di__apaframe()** function. If you want to use **di__starttext()** to append text to an anchored text frame, call **di__apaframe()** first, then call **di__starttext()**. **di__starttext()** will return a text handle to which you can append information.

When you have finished appending information to your document, call **di__finish()** to get a reference for your new file.

A template for creating documents is shown below:

```
main()
{
    di__start()
    ...
    di__ap*()
    ...
        [gi__*() or ti__*()]
        ...
        [di__apaframe()]
        ...
        [di__starttext()]
        ...
    di__finish()
    ...
    dsktp__movedoc()
}
```

Example of creating a document

The example below creates a document, and places it on the lower right hand corner of the desktop. The document's contents contain the line "This Document's Name is <document>" where <document> is a filename given as a variable in the invocation of this program.

```
#include    <stdio.h>
#include    <malloc.h>
#include    <string.h>
#include    <doctk/DocICProps.h>
#include    <doctk/DocIC.h>
/*
The following define statements set the text characteristics
to be 18 points, and Classic font. The DocICProps.h header
file will show how the font families are numbered.
*/
#define     FONT  0
#define     POINT  18

main(argc, argv)
int argc;
char *argv[];
{

    ret_sc ret;
    XString name, star_text;
    char insert_text[256];
    ret_fc ret2;
    unsigned short vers;
    di_tcont tcontainer;
    int i, ok, cr;
    dp_fontprops foprops;
    dp_paraprops prprops;
    dp_paraprops prprops2;
    dp_pageprops pgprops;
    dp_breakprops bkprops;
    dp_modeprops mdprops;
    dp_modesel modeselect;
/*
The following lines allocate memory and retrieve the
document's name from the dsktp_movedoc function. name
will now contain the document's name.
*/
    name = (XString)malloc((strlen(argv[1]) + 1) * 2);
    XStrfromASC(name, argv[1]);
/*
The following lines retrieve the default font properties, page
properties, page break properties and mode properties.
*/
    dp_getfontdef(&foprops);
    dp_getparadef(&prprops);
    dp_getparadef(&prprops2);
    dp_getpagedef(&pgprops);
    dp_getbreakdef(&bkprops);
    dp_getmodedef(&mdprops);
/*
The following lines set the font to be 18 point Classic
underlined with 1 line. FONT and POINT were set by the
define statements above.
```

```
*/
    foprops.fontdesc.family = FONT;
    foprops.fontdesc.size = POINT;
    foprops.udlines = 1;
/*
The following lines set the prprops paragraph aligned to the
left, line height of 15, and pre-leading set to 0
*/
    prprops.basprops.lnh = 15;
    prprops.basprops.prelead = 0;
    prprops.basprops.paralign = PA_LEFT;
/*
The following lines set the prprops2 paragraph to be
centered, line height of 24, pre- and post-leading to be 6
*/
    prprops2.basprops.lnh = 24;
    prprops2.basprops.prelead = 6;
    prprops2.basprops.poslead = 6;
    prprops2.basprops.paralign = PA_CENTER;
/*
The following line creates a new document with no header,
no footer and no page numbers. The document will have
font, paragraph, and page properties defined by foprops,
prprops, and pgprops above. The first new paragraph
character and page format characters will have default
values. The finished document is to have simple pagination.
*/
    if ( di_start(1, FALSE, FALSE, FALSE, &foprops, &prprops,
          &pgprops, NULL, &ret )) exit (-1)
/*
The following lines create the text string "This Document's
Name is <value of argv[1]> and place it into the character
string insert_text. A carriage return is also appended to the
text string.
*/
    strcpy(insert_text, "This Document's Name is ");
    strcat(insert_text, argv[1]);
    cr = strlen (insert_text);
    insert_text[cr] = 0x0d;
    insert_text[cr + 1] = '\0' ;
/*
The following lines allocate memory for the variable
star_text and then copies the value of insert_text into
star_text.
*/
    star_text = (XString)malloc(strlen(insert_text) * 2 + 2);
    XStrfromASC (star_text, insert_text);
/*
The following lines set the text container type to be a
document, and set handle to be ret.doc.
*/
    tcontainer.type = TC_DOC;
    tcontainer.h.doc = ret.doc;
/*
The following lines append the text string contained in
star_text to the text container. star_text memory is then
freed.
*/
    if (di_aptext(&tcontainer, star_text, &foprops ))
         exit( -1 );

    free(star_text);
```

```
/*
The following line sets the current paragraph to those
properties defined by prprops2.
*/
    if (di_setpara(&tcontainer, &prprops2)) exit( -1 );
/*
The following line appends five new paragraph characters
with page and paragraph properties specified by prprops,
and foprops.
*/
    if (di_apnewpara(&tcontainer, &prprops, &foprops, 5 ) )
            exit( -1 );
/*
The following line appends a page break character using
break properties specified in bkprops, and font properties
specified in foprops,
*/
    if (di_apbreak(&tcontainer, &bkprops, &foprops ))
            exit( -1 );
/*
The following lines set the document to display the structure
and non-printing characters.
*/
    mdprops.strct = TRUE;
    mdprops.nonprint = TRUE;
    if (di_setmode( ret.doc, &mdprops, modeselect ))
            exit( -1 );
/*
The following lines close the document, and release the
document handle.
*/
    if (di_finish(&ret.doc, NULL, NULL, &ret2 ))
            exit( -1 );
/*
The following lines move the document to the desktop, and
give the document the name specified by name. The memory
for name is then deallocated.
*/
    if (dsktp_movedoc(&ret2.ref[0], NULL, name, &vers))
            exit( -1 );

    free(name);
}
```

When creating and naming new documents, you must adhere to the legal character set of the GlobalView environment. The legal character set of the GlobalView environment is defined in the Xerox Character Code Standard.

Setting properties

In the preceding example, some font properties are set to non-default values. Specifically, the type is to be 18-point Century and underlined with 1 line. This was accomplished through the statements

```
#define  FONT 0
#define  POINT 18

dp_fontprops foprops;

dp_getfontdef (&foprops);
```

```
foprops.fontdesc.family = FONT;
foprops.fontdesc.size = POINT;
foprops.udlines = 1;

if(di_start(1, FALSE, FALSE, FALSE, &foprops, &prprops,
    &pgprops, NULL, &ret)) exit(-1);
```

Looking in the **dp__props** section of the *Document Interfaces Toolkit Reference Manual*, you'll notice that **dp__fontprops** is the main data type used to describe the font properties. **dp__fontprops** defines the font characteristics, whether the characters are underlined, subscript, superscript, etc. Setting the members associated with this property affects the visual aspects of the text string accordingly. In the preceding example, visual aspects of the text were changed based upon the values assigned to the members of **dp__fontprops**. The reference manual, and the *DocICProps.h* header file, list the members of **dp__fontprops**. The members are:

```
dp__fontdesc fontdesc;
unsigned udlines;
dp__bool stkout;
dp__place place;
dp__bool tobedel;
dp__bool revised;
dp__width width;
XString stylename;
dp__fontelmarr ntrelm;
dp__bool tranpare;
dp__color txtcol;
dp__color hlcol;
```

In addition, **fontdesc** contains the members

```
dp__family family;
dp__dvariant dvariant;
dp__weight weight;
unsigned short size;
```

Now, to set the font properties, the following line:

```
dp_getfontdef(&foprops);
```

initializes all the font properties to the following values:

```
dp__fontdesc fontdesc;
unsigned udlines;       /* 0 - no underlines */
dp__bool stkout;        /* FALSE - do not strikeout
                           characters */
dp__place place;        /* PL__NULL - standard position, i.e.
                           text is not superscript nor
                           subscript */
dp__bool tobedel;       /* FALSE - text is not marked for
                           deletion in redlining mode*/
dp__bool revised;       /* FALSE - text is not typed while
                           redlining enabled */
dp__width width;        /* WD__PROP - proportional, i.e.
                           normal spacing */
XString stylename;      /* NULL - no style sheet name */
dp__fontelmarr ntrelm; /* TRUE - all elements of ntrelm are
                           set to true, meaning neutral
                           elements of the style property */
```

```
dp__bool tranpare;        /* TRUE - text is painted transparent,
                             not solid*/
dp__color txtcol;         /* 0,0,0 text color is black */
dp__color hlcol;          /* 10000,0,0 highlighted text color
                             is white */
```

dp__getfontdescdef() is used to set its members to the following values:

```
dp__family family;        /* FMY__FRUT - frutiger, also known
                             as modern */
dp__dvariant dvariant;    /* DV__ROMAN - numbers
                             displayed in roman */
dp__weight weight;        /* WT__MEDIUM - text weight is
                             medium */
unsigned short size;      /* 12 - 12 point text */
```

In the example, all the default values except for **dp__family, size,** and **udlines** are kept. Once again, by referring to the reference manual or to the *DocICProps.h* header file, you will see that the example must set **dp__family** to 0 to get Classic font, **size** to 18 to get 18-point text, and **udlines** to 1 to get underlining with a single line. The default values of these variables are overridden by the following lines:

```
#define  FONT 0
#define  POINT 18
foprops.fontdesc.family = FONT;
foprops.fontdesc.size = POINT;
foprops.udlines = 1;
```

Now, when **di__start()** is called, the text font within the document will be 18-point Classic font with a single underline.

```
if(di__start(1, FALSE, FALSE, FALSE, &foprops, &prprops,
    &pgprops, NULL, &ret)) exit(-1);
```

Page properties, paragraph properties, and other properties are set in a similar manner. Properties and their default values may be found in the *Document Interfaces Toolkit Reference Manual.*

Enumerating a document

Document enumeration and creation are two separate activities. Functions and handles associated with one activity should not be used with the other. Enumeration is only the listing or displaying of text or properties. As such, it is a read-only process. No editing should be attempted while enumeration is in progress. Likewise, no enumeration should be attempted while creating a document.

The first step in enumerating a document is to call the **dsktp__getdocref()** function. This function requires the name and version of the document to open. **dsktp__getdocref()** then returns a file handle for that document. This file handle may then be passed as an argument to other functions. You may then call **di__open()** to open the document. This causes a specific handle to be returned. In this case, the handle returned not only identifies the files to be manipulated, but indicates that the appropriate data structures have been set. The next step is to pass the **doc** handle and a **di__enumprocs** structure into the **di__enumerate()** function. The **di__enumprocs** structure contains a set of call-back procedures, one for each of the following types of structures: anchored frame, break character, field, footnote, index, new paragraph, page format character, text or tile. These call-back procedures are not provided in the Document Interfaces Toolkit, and must be written by the programmer.

di__enumerate() proceeds sequentially through the document, starting at the beginning of the document. As **di__enumerate()** encounters different structures within the document, it calls the appropriate call-back procedure. If no call-back procedure has been written for a specific structure, that structure will be ignored. Each of these call-back procedures returns a Boolean value. If any one of these procedures returns **TRUE**, the enumeration process will terminate immediately. If the return value is never **TRUE**, the enumeration will continue to the end of the document.

When the enumeration process is complete, call **di__close()** to free all associated data structures and close any open file handles associated with the document.

A simple template for enumerating documents is shown below:

```
main()
{
    dsktp__getdocref()
    ...
    di__open()
    ...
    di__enumerate()
    ...
    di__close()
    ...
    dsktp__movedoc()
}
```

Example of enumerating a document

```
#include   <stdio.h>
#include   <malloc.h>
#include   <string.h>
#include   <doctk/DocICProps.h>
#include   <doctk/DocIC.h>

FILE       *fp;
short      ufileflag = FALSE;

extern dp_bool find_text();
extern dp_bool find_newpara();
extern dp_bool find_field();
extern dp_bool find_index();
/*
The following lines create an error display routine used in
this program to display errors and error numbers if
encountered
*/
int error_display( str, no )
char *str;
```

```
int no;
{
    if (no == 0) no = getsigno();
    fprintf(stderr,"Error: %s \n Error No. = %X \n", str, no );
};
/*
The following routine is a call-back procedure to find the
index
*/
dp_bool find_index(cdat, foprops, ixprops, index)
void(*cdat);
dp_fontprops *foprops;
dp_indexprops *ixprops;
di_index index;
{
    di_tcont inindex;
    di_enumprocs procs;
    dp_bool stops;
/*
The following line sets all elements of di_enumprocs to NULL
*/
    di_getenumprocsdef(&procs);
/*
The following lines identify the routines find_newpara and
find_text as the routines to find a new paragraph and text
respectively
*/
    procs.newpara = find_newpara;
    procs.text = find_text;
/*
The following lines identify the object type to be TC_INDEX,
and the handle to be index
*/
    inindex.type = TC_INDEX;
    inindex.h.index = index;
/*
The following lines enumerate the text container inindex,
using the call-back procedures defined by procs, and will not
include page numbering patterns, as indicated by the FALSE
value for the mrgnum argument
*/
    if (di_enumerate(&inindex, &procs, cdat, FALSE, &stops))
{
        error_display( "Enumerate", 0 );
    return( TRUE );
};

return( FALSE );
}
/*
The following routine is used to find fields within the
document
*/
dp_bool find_field(cdat, foprops, fiprops, field)
void(*cdat);
dp_fontprops *foprops;
dp_fldprops *fiprops;
di_field field;
{
    di_tcont infield;
    di_enumprocs procs;
    dp_bool stops;
```

```
/*
The following line sets all elements of di_enumprocs to NULL
*/
di_getenumprocsdef(&procs);
/*
The following lines identify the routines find_newpara and
find_text as the routines to find a new paragraph and text
respectively
*/
    procs.newpara = find_newpara;
    procs.text = find_text;
/*
The following lines identify the object type to be TC_FIELD,
and the handle to be field
*/
    infield.type = TC_FIELD;
    infield.h.field = field;
/*
The following lines enumerate the text container using the
call-back procedures defined by procs, and will not include
page numbering patterns, as indicated by the FALSE value for
the mrgnum argument
*/
    if (di_enumerate(&infield, &procs, cdat, FALSE, &stops)) {
        error_display( "Enumerate", 0 );
        return( TRUE );
    }

    return(FALSE );
}
/*
The following routine adds a newline character to the
output file
*/
dp_bool find_newpara(cdat, foprops, prprops )
void *cdat;
dp_fontprops *foprops;
dp_paraprops *prprops;
{
/*
The following lines place a newline character in the output
file if it exists. ufileflag is set to TRUE in the main program if
an output file is specified in the invocation of the program
*/
    if (ufileflag = = TRUE) {
        if (fputs ( "\n", fp ) < 0) {
            fprintf(stderr, "output file write error.\n");
            return(TRUE);
        }
    }
    else {
        fprintf(stdout, "\n");
    }
    return ( FALSE );
}

dp_bool find_text( cdat, foprops, text )
void(*cdat);
dp_fontprops *foprops;
XString text;
{
    short i = 0, j = 0;
    char *ascii_text;
    XString text_work;
/*
The following lines allocate memory for the text string
variables
*/
    ascii_text = (char *)malloc(XStrlen(text) + 1);
    text_work = (XString)malloc (XStrlen(text) * 2 + 2);
/*
The following lines add the document character strings to
the text_work variable
*/
    while ( i < XStrlen(text) ) {
         if (*(text + i) < 0x0080 ) {
              *(text_work + j) = *(text + i);
              j + + ;
         }
         i + + ;
    }
/*
The following line adds the null character to the end of the
text string as per the C requirements
*/
    *(text_work + j) = '\0';
/*
The following lines convert the XString text to ASCII format
so that it may be appended to the output file
*/
    if (XStrtoASC(text_work, ascii_text)) {
         fprintf(stderr, "Bad string occurred !\n");
         return(TRUE);
    }
/*
The following lines add the text string to the output file if it
exists. ufileflag is set to TRUE in the main program if an
output file is specified in the invocation of the program
*/
    if (ufileflag = = TRUE) {
         if (fputs(ascii_text, fp ) < 0) {
              fprintf(stderr, "output file write error.\n");
              return (TRUE);
         }
    }
    else {
         fprintf(stdout, "%s",ascii_text);
    }
/*
The following lines use the UNIX memory deallocation call to
free the memory assigned to the text string variables
*/
    free(ascii_text);
    free(text_work);
    return(FALSE );
}
/*
The following routine is the main program
*/
main(argc, argv)
int argc;
char *argv[];
{
    dsktp_docref ref;
    di_doc sdoc;
    di_enumprocs procs;
    ret_open ret;
    XString name[256], star_text[256];
    char insert_text[256];
    unsigned short vers;
    di_tcont tcontainer;
    dp_bool mrgnum, stops;
    int i, ok;
/*
The following lines check for invocation syntax errors, and if
the output file can be opened successfully
*/
    if (argc < 2) {
            fprintf(stderr, "usage: %s document_name
                [UNIX_filename] \n", argv[0]);
            exit(1);
    }
    if (argc > 2) {
            ufileflag = TRUE;
            if ((fp = fopen (argv[2], "w")) = = NULL ) {
                fprintf(stderr, "Destination file Open Error
                    !!\n");
                exit(1);
            }
    }

    XStrfromASC (name, argv[1]);
/*
The following lines retrieve the latest version of the
document from the desktop
*/
    ok = dsktp_getdocref (name, LASTVERS, NULL, &ref[0]);
    if (ok) {
            fprintf (stderr, "Can't get document
                reference.\n");
            exit (1);
    }
/*
The following line opens the document
*/
    if (di_open(ref, &ret )) error_display( "Open", 0 );
/*
The following lines set the text container to be document,
and set text handle to be ret.doc and sdoc
*/
    tcontainer.type = TC_DOC;
    tcontainer.h.doc = ret.doc;
    sdoc = ret.doc;

    mrgnum = FALSE;
/*
The following line sets all elements of di_enumprocs to NULL
*/
di_getenumprocsdef( &procs );
/*
The following lines set the callback procedures to the
following routines defined earlier in this program
*/
    procs.newpara = find_newpara;
    procs.text = find_text;
```

```
        procs.field = find_field;
        procs.index = find_index;
    /*
    */
        if ( di__enumerate (&tcontainer, &procs, &sdoc, mrgnum,
              &stops)) {
              error_display( "Enumerate", 0 );
              di__close( &ret.doc );
              exit( 1 );
        };
    /*
    The following line finalizes and closes the document
    */
        if (di__close(&ret.doc )) error_display( "Close", 0 );
        fclose(fp);
    }
```

di__*() functions

Use the following charts may be used as an aid to the selection and application of **di__*()** functions. Detailed information regarding each function may also be found in the *Document Interfaces Toolkit Reference Manual.*

Table 4-1. **Usage of di__*() functions**

Object	Creating	Reading
	Function	Function
Common		di__enumerate
Document	di__start	di__open
	di__finish	di__close
	di__abort	
Text	di__aptext	di__textproc
	di__apchar	di__reltext
	di__reltext	
Anchored Text Frame	di__starttext	di__aframeproc
	di__apaframe	di__textforaframe
	di__relcap	

Table 4-1. **Usage of di__*() functions**

Object	Creating	Reading
	Function	Function
Anchored Footnote	di__setfnprops	di__aframeproc
	di__apaframe	di__fnpropsproc
	di__fntile	di__getfnprops
	di__relcap	di__fntileproc
Other Anchored Frames	di__apaframe	di__aframeproc
	di__relcap	
Break	di__apbreak	di__breakproc
Field	di__field	di__fieldproc
	di__relfield	di__getfieldfromname
		dp__enumfrun
Index	di__apindex	di__indexproc
	di__relindex	
New paragraph	di__apnewpara	di__newparaproc
	di__setpara	
Page (Header, Footer, Numbering)	di__appfc	di__pfcproc
	di__relhead	di__docproc
	di__relfoot	
	di__relnum	
SoftPageBreak		di__sfbrkproc
FillInOrder	di__aptofillin	di__fillinproc
	di__clearfillin	di__enumfillin

Table 4-1. **Usage of di__*() functions**

Object	Creating	Reading
	Function	Function
Style	di__start	di__enumstyle
	di__styleproc	di__fstyleproc
	di__apfstyleproc	di__pstyleproc
	di__appsytleproc	
	di__apfstyle	
	di__appsytle	
TextLink	di__aptotxtlnk	di__txtlnkproc
	di__cleartxtlnk	di__enumtxtlnk
Mode	di__setmode	di__getmode

Using the *GraphicsIC.h* header file

The *GraphicsIC.h* header file contains functions and properties used in the creation and enumeration of graphic frames.

Generating graphics

Graphics may be created and added to a document which is currently open for editing. Graphics creation is initiated by a call to **gi__startgr()**. This function creates a graphics container and returns a **handle**.

The returned **handle** from **gi__startgr()** identifies the graphics frame in which graphics data is to be placed. The handle is then passed as an argument to various **gi__ad*()** functions when adding new graphic objects. Depending on the function called, you may add graphic objects like curves or rectangles, bitmap frames, or text frames to the graphics frame.

Nested frames, such as non-anchored graphics frames, CUSP buttons, or graphics clusters may also be added to the graphics frame. A nested frame is simply a frame placed within a larger frame. **gi__startgframe()**, **gi__startnbtn()**, **gi__startcluster()** functions create the respective nested frame listed above.

When everything has been added to the graphics container, the final step is a call to **gi__finish*()**. The following is a template of the graphics creation process:

```
main()
```

```
{
    di_start()
    ...
    gi_startgr()
    ...
    gi_ad*(h)
    ...
    gi_finish*()
    ...
    di_apaframe()
    ...
    di_finish()
    ...
    dsktp_movedoc()
}
```

Example of generating graphics

An example of the graphics generation process is shown below. In this example, a new document file is created, an anchored graphic frame is added to the document, and a curve is inserted within the graphic frame. The document is then placed on the lower right corner of the desktop. The following calls are made in this example:

```
di_start()
gi_startgr()
gi_get*()
gi_ad*()
gi_finishgr()
di_apaframe()
di_finish()
```

The example program is listed below:

```
#include <stdio.h>
#include <doctk/DocICProps.h>
#include <doctk/DocIC.h>
#include <doctk/XString.h>
#include <doctk/Desktop.h>
#include <doctk/GraphicsIC.h>

main (argc,argv)
int argc;
char *argv[];

{
    ret_sc ret;
    ret_fc ret2;
    XString name;
    di_tcont to;
    di_aframetype type;
    gi_handle h;
    gi_box box;
    gi_curveprops props;
    di_ins contents;
    dp_frameprops frame;
    ret_apaframe anc_ret;
    char *ascii;
    unsigned short vers;
/*
```

```
The following line creates a document with no header,
footer, or page numbers and uses default values for the
page, font, and mode properties
*/
    if (di_start(PO_COMPRESS, FALSE, FALSE, FALSE,NULL,
        NULL, NULL, NULL, &ret)) exit(-1)
/*
The following line creates a graphics frame, where h is the
handle to the frame
*/
    if (gi_startgr(ret.doc,&h)) exit (-1)
/*
The following lines initialize graphics frame box and curve to
their default property values
*/
    gi_getboxdef(&box);
    gi_getcurvepropsdef(&props);
/*
The following line adds a curve with properties defined by
&props
*/
    if (gi_adcurve(h,&box,&props)) exit (-1)
/*
The following line frees the graphics handle
*/
    if (gi_finishgr(h,&contents)) exit (-1)
/*
The following lines set the text container to be a document,
and set text handle to be ret.doc
*/
    to.type = TC_DOC;
    to.h.doc = ret.doc;
/*
The following line defines the type of frame to be appended
as a graphics anchored frame
*/
    type = AF_GRAPH;
/*
The following line retrieves the default properties of the
anchored frame
*/
    if (dp_getframedef(&frame)) exit (-1)
/*
The following line appends the graphics frame to the
document with contents from gi_finishgr. No captions are
placed; use default font properties. Frame size automatically
adjusted to fit existing frame
*/
    if (di_apaframe(&to, type, &frame, contents, FALSE,
        FALSE, FALSE, FALSE, NULL, FALSE, &anc_ret)) exit
        (-1)
/*
The following line finalizes the document, and releases the
document handle
*/
    if (di_finish(&ret.doc,NULL,NULL,&ret2)) exit (-1)
/*
The following lines retrieve the name of the document, and
place it onto the desktop
*/
    ascii = argv[1];
    name = (XString)malloc((strlen(ascii) + 1) * 2);
```

```
        XStrfromASC(name,ascii);
        dsktp_movedoc(&ret2.ref[0],NULL,name,&vers);
    }
```

Enumerating graphics

The **gi_enumerate()** function is used to read the contents and properties of a graphic frame. It reads the contents of the graphic frame, and calls the appropriate call-back procedure for each object encountered. The graphic frame and user-defined call-back procedures are supplied as arguments to **gi_enumerate()**. If an object is encountered for which a call-back procedure has not been defined, that object will be ignored.

Example of enumerating graphics

```
#include <stdio.h>
#include <doctk/DocIC.h>
#include <doctk/DocICProps.h>
#include <doctk/GraphicsIC.h>
#include <doctk/XString.h>
#include <doctk/Desktop.h>
/*
The following routine, curveproc, simply prints out all the
properties associated with any encountered curves
*/
dp_bool curveproc(cdat, box, props)
void *cdat;
gi_box *box;
gi_curveprops *props;
{
    printf("**** curve's box & a part of curve props
            values ****\n");
    printf("box.place.[x,y] = [%d, %d] (mica)\n",
            box->place.x, box->place.y);
    printf("box.dims.[w,h] = [%d, %d] (mica)\n",
            box->dims.w, box->dims.h);
    printf("curveprops.brsh.wth = %d\n",
            props->brsh.wth);
    printf("curveprops.brsh.stylebrush = %d\n",
            props->brsh.stylebrush);
    printf("curveprops.brsh.brushcolor.y = %d\n",
            props->brsh.brushcolor.y);
    printf("curveprops.brsh.brushcolor.e = %d\n",
            props->brsh.brushcolor.e);
    printf("curveprops.brsh.brushcolor.s = %d\n",
            props->brsh.brushcolor.s);
    printf("curveprops.lnenw = %d\n", props->lnenw);
    printf("curveprops.lnese = %d\n", props->lnese);
    printf("curveprops.lnhnw = %d\n",
            props->lnhnw);
    printf("curveprops.lnhse = %d\n", props->lnhse);
    printf("curveprops.plnw.x = %d\n",
            props->plnw.x);
    printf("curveprops.plnw.y = %d\n",
            props->plnw.y);
    printf("curveprops.plapx.x = %d\n",
    props->plapx.x);
    printf("curveprops.plapx.y = %d\n",
          props->plapx.y);
    printf("curveprops.plse.x = %d\n", props->plse.x);
    printf("curveprops.plse.y = %d\n", props->plse.y);
    printf("curveprops.plpek.x = %d\n",
          props->plpek.x);
    printf("curveprops.plpek.y = %d\n",
          props->plpek.y);
    printf("curveprops.eccentric = %d\n",
          props->eccentric);
    printf("curveprops.eccentricity = %d\n",
          props->eccentricity);
    printf("curveprops.fixangle = %d\n",
          props->fixangle);

}
/*
The following routine, aframeproc, is the call-back
procedure that is called when an anchored graphics frame is
encountered within a document
*/
dp_bool aframeproc(cdat, type, font, frame, props, cont,
    tcap, bcap, lcap, rcap)
void *cdat;
di_aframetype type;
dp_fontprops *font;
di_ins frame;
dp_frameprops *props;
di_ins cont;
di_caption tcap;
di_caption bcap;
di_caption lcap;
di_caption rcap;
{
    dp_bool stops;
    gi_enumprocs gprocs;
/*
The following line checks to see if an an anchored graphics
frame has been encountered, and has some contents
*/
    if (type = = AF_GRAPH && cont ! = NULL) {
/*
The following line sets all elements of gi_enumprocs to NULL
*/
          gi_getenumprocsdef(&gprocs);
/*
The following line identifies curveproc as the call-back
procedure if a curve is encountered
*/
          gprocs.curve = curveproc;
/*
The following line is the enumeration process, using the call-
back procedures as defined by gprocs
*/
          if (gi_enumerate(frame, gprocs, NULL, &stops)
                        != 0) {
                        printf(" Error(gi_enumerate) getsigno =
                                %x \n", getsigno() );
                        return(TRUE);
                }
        }
        return(FALSE);
}

/*
The following routine is the main program for enumerating
graphics
*/
main( argc, argv)
int  argc;
char        *argv[];
{
        ret_open ret_op;
        dsktp_docref ref;
        di_tcont to;
        XString name;
        di_enumprocs procs;
        dp_bool stops;
/*
The following lines allocate memory for the document,
translate the ASCII name into an equivalent XString format,
and then retrieve the document from the desktop
*/
        name = (XString)malloc((strlen(argv[1]) + 1) * 2);
        XStrfromASC(name, argv[1]);
        if (dsktp_getdocref(name, LASTVERS, NULL,
                &ref[0])) {
                printf(" Error(dsktp_getdocdef)
                        getsigno = %x \n", getsigno());
                exit(-1);
        }
/*
The following lines open the document, and returns an error
message if the open is unsuccessful
*/
        if (di_open(ref, &ret_op)) {
                printf(" Error(di_open) getsigno = %x \n",
                        getsigno());
                exit(-1);
        }
        if (ret_op.status != OP_OK) {
                printf("error status = %d\n", ret_op.status);
                exit(-1);
        }
/*
The following lines set the text container to be a document,
and the text handle to be ret_op.doc
*/
        to.type = TC_DOC;
        to.h.doc = ret_op.doc;
/*
The following line sets all elements of di_enumprocs to NULL
```

```
*/
    di_getenumprocsdef(&procs);
/*
The following line identifies aframeproc as the call-back
procedure if an anchored frame is encountered
*/
    procs.aframe = aframeproc;
/*
The following line enumerates the document, using the call-
back procedures defined by &procs
*/
    if (di_enumerate(&to, &procs, NULL, TRUE, &stops)) {
        di_close(&ret_op.doc);
        exit( 1 );
    }
/*
The following lines close the document, and returns an error
message if the close function is unsuccessful
*/
    if (di_close(&ret_op.doc)) {
        printf(" Error(di_close) getsigno = %x \n",
            getsigno());
        exit(-1);
    }
}
```

gi_*() functions

The following charts may be used as an aid to the selection and application of **gi_*()** functions. Detailed information regarding each function may also be found in the *Document Interfaces Toolkit Reference Manual.*

Table 4-2. **Anchored graphic and anchored button frame functions**

Category	Creating	Reading
	Function Name	Function Name
Common	di__apaframe	di__enumerate
		di__aframeproc
Anchored Graphics Frame	gi__startgr	gi__getgframeprops
	gi__finishgr	
	gi__setgframeprops	
Anchored Button Frame	gi__startbtn	gi__btnforaframe
	gi__finishbtn	gi__enumbtnprog
	gi__relbtnprog	
	gi__apchartobtnprog	
	gi__apnparatobtnprog	
	gi__aptexttobtnprog	

Table 4-3. **Graphic objects and related functions**

Category	Objects	Creating	Reading
		Function Name	Function Name
Common			gi__enumerate
Primitive Objects	Point	gi__adpoint	gi__pointproc
	Line	gi__adline	gi__lineproc
	Curve	gi__adcurve	gi__curveproc
	Ellipse	gi__adellipse	gi__ellipseproc
	Rectangle	gi__adrectangle	gi__rectangleproc
	Triangle	gi__adtriangle	gi__triangleproc
	Pie Slice	gi__pislce	gi__pislceproc

Table 4-3. **Graphic objects and related functions**

Category	Objects	Creating Function Name	Reading Function Name
Frame	Bitmap Frame	gi_adbm	gi_bmproc
	Text Frame	gi_adtframe	gi_tframeproc
	Form Field	gi_adffield	gi_ffieldproc
	Nested Graphics Frame	gi_startgframe	gi_frameproc
		gi_finishgframe	
	Nested Table	gi_adtable	gi_tableproc
	Nested Button Frame	gi_startnbtn	gi_buttonproc
		gi_finishnbtn	gi_enumbtnprog
		gi_relbtnprog	
		gi_apchartobtnprog	
		gi_apnparatobtnprog	
		gi_aptexttobtnprog	
Chart	Bar Chart	gi_adbacht	gi_bachtproc
	Line Chart	gi_adlncht	gi_lnchtproc
	Pie Chart	gi_adpicht	gi_pichtproc
		gi_finishcht	
Others	Cluster	gi_startcluster	gi_clusterproc
		gi_finishcluster	

Using the *TableIC.h* header file

A table is defined by three types of properties: table, column, and row. Table properties specify the name of the table, a description of the table headers, and the number of columns and rows in the table. Column properties determine how the columns are to be divided, and how text is to be aligned within the columns. Row properties determine how the text is to be aligned within each row. The actual contents of a table are included as a member of the row properties.

Table generation

A new table is initiated upon calling **ti__starttable()**. Table properties and column properties are passed to **ti__starttable()**, which returns a table handle, **Handle**. The table handle is a pointer to **Object**. **Object** is an enumerated type that contains table-related data and a pointer to the actual table contents. Initially, the row properties have default values so the table will have no contents. After calling **ti__starttable()**, you may add contents and properties to each row via calls to other **ti*__()** functions.

To add contents to the table, pass the table handle to **ti__appendrow()**. When all the rows have been added, finalize the table by calling **ti__finishtable()**.

ti__finishtable() returns an instance of the table called **di__ins**, which you can then pass to **di__apaframe()** or **gi__adtable()**. The remaining table frame properties, like captions and border style, are handled by calls to *DocIC* and *GraphicsIC* functions.

To add information to an existing table, you should call **ti__startextable()** instead of **ti__starttable()**. **ti__startextable()** also returns a table handle, which may then be passed to **ti__appendrow()** and later to **ti__finishtable()**. **ti__startextable()** requires a **di__ins** as an argument which may be obtained from **di__enumerate()**. The handle of the document to be enumerated must come from **di__startap()**.

Example of generating a table

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <doctk/DocICProps.h>
#include <doctk/DocIC.h>

#define PAGINATE PO_COMPRESS

extern int createTable();
/*
The following lines create an error display routine used in
this program to display errors and error numbers if
encountered
*/
int error_display(str, no )
char *str;
int no;
{
    if (no = = 0 ) no = getsigno();
    fprintf (stderr,"%s Error error No. = %X
          \n", str, no);
    exit(1);
};

main(argc, argv)
int argc;
char *argv[];
{
        XString iDocName;
        ret_sc ret;
        ret_fc ret2;
        ret_apaframe ret3;
        unsigned short vers;
        int i, numChar, ok;
/*
The following lines set the font, paragraph, and page
properties  to their default values
*/
        dp_fontprops foprops, *ifoprops = NULL;
        dp_paraprops prprops, *iprprops = NULL;
        dp_pageprops pgprops,*ipgprops = NULL;
        dp_breakprops bkprops;

        dp_modeprops mdprops;
        dp_modesel mdselect;
/*
The following lines check for length of document name,
memory allocation problems, and invocation syntax errors
*/
    switch (argc) {
            case 1:
                    iDocName = NULL;
                    break;
            case 2:
                    numChar = strlen(argv[1] );
                    if (numChar > 100 ) {
                            fprintf(stderr, " too long
                            document_name \n" );
                            exit( 1 );
                    };
                    iDocName = (XString)malloc(
                            sizeof (XChar) * (numChar + 1) );
                    if ( iDocName) XStrfromASC
                            (iDocName, argv[1] );
                    else error_display ("Malloc", 1 );
                    break;
            default:
                    fprintf(stderr, "usage: %s
                            document_name \n", argv[0] );
                    exit( 1 );
            };

/*
The following line creates a document with no header,
footer, or page numbers.  The font, paragraph and page
properties are defined by ifoprops, iprprops, and ipgprops
above. The first new paragraph character and page format
character will have default values
*/
        if ( di_start( PAGINATE, FALSE, FALSE,
                FALSE, ifoprops, iprprops, ipgprops, NULL, &ret ))
                error_display( "Start", 0 );
/*
```

```
The following lines check to see if the document was created
or opened successfully.  If not, then, give error status
number and abort
*/
    if (ret.stat ! = SC_OK ) {
            fprintf(stderr, " ret_sc status = %d\n",
            ret.stat );
            if (di_abort (&ret.doc )) exit( -1 );
            exit( 1 );
    };

    for ( i = 0; i < = (int)ME_PROMPT; i + + )
            mdselect[i] = TRUE;
/*
The following lines set the document to show structure,
show non-printing characters, not to display cover sheet, and
not to prompt for field
*/
    mdprops.strct = TRUE;
    mdprops.nonprint = TRUE;
    mdprops.cover = FALSE;
    mdprops.prompt = FALSE;

    if (di_setmode ( ret.doc, &mdprops,
            mdselect)) exit ( -1 );
/*
The following line calls the createTable routine to create the
table for this document
*/
    if (createTable (&ret.doc, &ret3))
            error_display ( "CreateTable", 0 );
/*
The following line finalizes the document
*/
    if (di_finish(&ret.doc, NULL, NULL, &ret2))
            error_display ( "Finish", 0 );
/*
The following line moves the document to the lower-right-
hand corner of the desktop
*/
    ok = dsktp_movedoc(ret2.ref, NULL,
            iDocName, &vers );
    fprintf(stderr," End of dsktp_movedoc
            status = %d, version = %d\n", ok, vers);
/*
The following line uses the UNIX free command  to
deallocate the memory set aside for the document
*/
    free(iDocName);
    if (ok) exit( 1 );

}
/*
The following lines show the code for createTable.
createTable is the routine that actually creates the table. The
units of measurement are in micas, where 1 mica = 1/100
mm
```

```
*/
#include  <stdio.h>
#include  <malloc.h>
#include  "DocICProps.h"
#include  "DocIC.h"
#include  "TableIC.h"
/*
The following lines define the table to have two columns,
and two rows
*/
#define  COLUMN_NUM   2
#define  ROW_NUM      2
/*
The following lines define the caption to appear at the
bottom of the table
*/
#define  WTCAP      FALSE
#define  WBCAP      TRUE
#define  WLCAP      FALSE
#define  WRCAP      FALSE
/*
The following line is used in di_apaframe and specifies that
the frame will be adjusted to fit the existing frame
*/
#define  TRSTS      FALSE

extern int error_display()

int createTable(doc, ret)
di_doc *doc;
ret_apaframe *ret;
{
    int i, ok;
    di_tcont tCont;
    dp_fontprops *afoprops = NULL;
    dp_frameprops frameRec;
    ti_tableprops propsRec;
    ti_colinfoseq colInfoSeq;
    ti_rowcontseq rowContSeq;
    ti_handle ret_h;
    ret_ft ret_finTable;

    tCont.type = TC_DOC;
    tCont.h.doc = *doc;
/*
The following lines allocate memory for the table columns
and rows
*/
    colInfoSeq.seq = (ti_colinforec*)
          malloc (sizeof (ti_colinforec) * COLUMN_NUM);
    if (colInfoSeq.seq = = NULL) error_display ("Malloc",
          1);
    rowContSeq.rowdat = (ti_rowent
          *)malloc (sizeof (ti_rowent) * COLUMN_NUM);
    if (rowContSeq.rowdat = = NULL) {
          free(colInfoSeq.seq);
          error_display("Malloc", 1);
    };
/*
The following lines retrieve the default properties of an
anchored frame to which the table will be placed
*/
    if (dp_getframedef(&frameRec)) {
        free(colInfoSeq.seq);
        free(rowContSeq.rowdat);
        error_display("GetFrameDefault", 0);
    };
/*
The following lines set the frame top margin to 18, and the
frame bottom margin to 36 micas
*/
    frameRec.tmgn = 18;
    frameRec.bmgn = 36;
/*
The following lines get the default properties for the table
*/
    if (ti_gettablepropsdef(&propsRec)) {
        free(colInfoSeq.seq);
        free(rowContSeq.rowdat);
        error_display("GetTablePropsDef",0);
    };
/*
The following line defers the table frame to the next page if
it cannot fit on the current page
*/
    propsRec.deferon = TRUE;
/*
The following lines set the amount of white space above and
below each header element to 329 micas and the line for the
table to two pixels in width
*/
    propsRec.thdmgn = 329;
    propsRec.bhdmgn = 329;
    propsRec.dvrline.width = LW_W2;
/*
The following line sets the default column property
definitions for the table
*/
    if (ti_getcolinforecdef(colInfoSeq.seq)) {
        free(colInfoSeq.seq);
        free(rowContSeq.rowdat);
        error_display("GetColInfoRecDef", 0);
    };
/*
The following lines set the column to have a width of 3000
micas, the left and right margin to 150 micas, and to be
subdivided into two columns
*/
    colInfoSeq.seq->width = 3000;
    colInfoSeq.seq->lmgn = 150;
    colInfoSeq.seq->rmgn = 150;
    colInfoSeq.length = COLUMN_NUM;
/*
```

```
The following line assigns each column in the table to the
properties defined above
*/
    for (i = 1; i < COLUMN_NUM; i++)*(colInfoSeq.seq +
          i) = *colInfoSeq.seq;
/*
The following lines set the default row property definitions
for the table
*/
    if(ti_getrowentdef(rowContSeq.rowdat)) {
          free(colInfoSeq.seq);
          free(rowContSeq.rowdat);
          error_display("GetRowEntDef", 0);
    };

/*
The following lines set the top and bottom margin to 139
micas, a solid line two pixels in width, text alignment is
centered within each row, and the number of rows to two
*/
    rowContSeq.tmgn = 139;
    rowContSeq.bmgn = 139;
    rowContSeq.line.style = LS_SOLID;
    rowContSeq.line.width = LW_W2;
    rowContSeq.valign = VA_FCENTER;
    rowContSeq.length = ROW_NUM;
/*
The following line assigns each row in the table to the
properties defined above
*/
    for (i = 1; i <ROW_NUM; i++)
          *(rowContSeq.rowdat + i) =
                *rowContSeq.rowdat;
/*
The following lines create a table with the properties
defined by &propsRec and &colinfoSeq
*/
    if (ti_starttable(*doc, &propsRec, &colInfoSeq, &ret_h))
    {
          free(colInfoSeq.seq);
          free(rowContSeq.rowdat);
          error_display("StartTable", 0);
    };
/*
The following lines add rows to the table. The rows will have
contents as defined by &rowContSeq
*/
    for (i = 0; i < ROW_NUM; i++)
          if (ti_appendrow (ret_h, &rowContSeq)) {
                free(colInfoSeq.seq);
                free(rowContSeq.rowdat);
                error_display("AppendRow", 0);
          };
/*
The following lines use the UNIX free function to free the
memory allocated to the table columns and rows
*/
```

```
    free(colInfoSeq.seq);
    free(rowContSeq.rowdat);
/*
The following line finalizes the table creation
*/

    if (ti_finishtable(ret_h, &ret_finTable))
        error_display("FinishTable", 0);
/*
The following line appends the anchored table frame to the
text container &tCont.
*/
    ok = di_apaframe(&tCont, AF_TABLE, &frameRec,
            ret_finTable.table, WTCAP, WBCAP, WLCAP,
            WRCAP, afoprops, TRSTS, ret);
/*
The following lines release the caption resources previously
allocated
*/
    if (WTCAP) di_relcap(&ret->tcap);
    if (WBCAP) di_relcap(&ret->bcap);
    if (WLCAP) di_relcap(&ret->lcap);
    if (WRCAP) di_relcap(&ret->rcap);

    return (ok);
}
```

Table enumeration

The **ti_enumtable()** function reads the contents of a table. **ti_enumtable()** takes as arguments an instance of a table, **di_ins**, and three call-back procedures:

ti_tableproc()
ti_columnproc()
ti_rowproc()

ti_enumtable() will call **ti_tableproc()** and **ti_columnproc()** once while enumerating a given table. These call-back procedures obtain the table and column properties. Since the table contents are stored in the row properties, **ti_enumtable()** will call **ti_rowproc()** once for each row in the table.

Example of enumerating tables

```
#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>
#include    <doctk/DocICProps.h>
#include    <doctk/DocIC.h>
#include    <doctk/Desktop.h>
#include    <doctk/Signals.h>
#include    <doctk/XString.h>
#include    <doctk/TableIC.h>

ti_stop tableproc();
ti_stop rowenum();
ti_stop gettableprops();
ti_stop getcolumnprops();
dp_bool readtext();
void errorexit();
int micatospace();
dp_bool paraproc();
dp_bool addtexttofile();
void printhorizontalline();

struct cdata {
FILE *fp;
di_doc doc;
ti_tableprops *tableprops;
ti_colinfo colprops;
int curcol, currow, *tablewidth;
};

main(argc, argv)
int argc;
char *argv[];
{
    char *ascii;
    XString docname;
    ret_open reto;
    di_tcont tcont;
    di_enumprocs docenums;
    dp_bool stop;
    unsigned short vers;
    dsktp_docref ref;
    struct cdata *mydata;
/*
The following lines allocate memory to hold the client data
*/
    if ((mydata = (struct cdata *)malloc (sizeof(struct cdata)))
            = = NULL) {
            fprintf (stderr, "Malloc error\n");
            exit (-1);
    }
/*
The following lines initialize the table row and column
pointers and properties
*/
    mydata->curcol = 0;
    mydata->currow  = -1; /* row 0 = header*/
    mydata->tablewidth = NULL;
    mydata->colprops = NULL;
    mydata->tableprops = NULL;
/*
The following lines check that a document name was given
during the invocation of this program
*/
    if (argc ! = 3) {
            fprintf(stderr, "You must provide a name for the
            document and output file i.e. the correct form is
            %s docname outputfile\n", argv[0]);
    exit(1);
    }
/*
```

```
The following line gets the document name from the
invocation argument, and allocates memory for the
document
*/
    ascii = argv[1];

    if ((docname = (XString)malloc((strlen(ascii) + 1) * 2))
            = = NULL) {
            fprintf(stderr, "Malloc error\n");
            exit( -1 );
    }
/*
The following line converts the ASCII file name into an
XString and stores it in the variable docname
*/
    XStrfromASC(docname, ascii);

    vers = 1;
/*
The following line retrieves the document from the desktop
*/
    if (dsktp_getdocref(docname, vers, NULL, ref))
            errorexit (NULL, NULL);
/*
The following lines open the document, and print an error if
problems occur while opening the document
*/
    if (di_open(ref, &reto)) errorexit(NULL, NULL);

    if (reto.status != OP_OK) {
    fprintf (stderr, "Problem opening named document\n");
            exit ( 1 );
    }

    mydata->doc = reto.doc;
/*
The following lines create a UNIX ASCII output file to contain
the enumerated table
*/
    if ((mydata->fp = fopen(argv[2], "w")) = = NULL) {
            fprintf(stderr, "Can't open file called %sn",
                    argv[2]);
            di_close(&reto.doc);
            exit( 1 );
    }

/*
The following lines set the text container type to be a
document, and the text container handle to be reto.do
*/
    tcont.type = TC_DOC;
    tcont.h.doc = reto.doc;
/*
The following line sets all elements of di_enumprocs to NULL
*/
    if (di_getenumprocsdef(&docenums))
            errorexit(&reto.doc, mydata->fp);
```

```
/*
The following line sets the call-back procedure for an
anchored frame to be tableproc
*/
    docenums.aframe = tableproc;
/*
The following lines enumerate the document using the call-
back procedures defined in docenums. No page numbering
patterns will be included in the heading or footing as
indicated by the FALSE value for mrgnum.
*/
    if (di_enumerate(&tcont, &docenums, mydata, FALSE,
          &stop)) errorexit(&reto.doc, mydata->fp);
/*
The following lines finalize the document and close it
*/
    if (di_close(&reto.doc)) errorexit(NULL, mydata->fp);

    if (fclose(mydata->fp)) {
          fprintf(stderr, "Problem closing text file\n");
          exit( 1 );
    }

 fprintf (stdout, "finished reading table\n");

 }
/*
The following routine is tableproc, and is the main program
for enumerating a table
*/
ti_stop tableproc(cdat, type, font, frame, props, cont, tcap,
    bcap, lcap, rcap)
void (*cdat);
di_aframetype type;
dp_fontprops *font;
di_ins frame;
dp_frameprops *props;
di_ins cont;
di_caption tcap;
di_caption bcap;
di_caption lcap;
di_caption rcap;
{
    ti_enumprocs tableenumprocs;
    struct cdata *mydata;

    mydata = (struct cdata *)cdat;
/*
The following line returns to the main program if the type
encountered is not an anchored table frame
*/
    if (type != AF_TABLE) return(FALSE);
/*
The following lines define the call-back procedures for the
table, column, and row enumeration procedures
*/
    tableenumprocs.table = gettableprops;
    tableenumprocs.column = getcolumnprops;
    tableenumprocs.row = rowenum;
/*
The following line enumerates the table using the call-back
procedures defined in tableenumprocs
*/
    if (ti_enumtable(cont, &tableenumprocs, mydata))
        return(TRUE);

    printhorizontalline(mydata, FALSE);
    return(FALSE);

}

/*
The following routine, gettableprops, retrieves the table
properties and places them into mydata->tableprops
*/
ti_stop gettableprops(cdat, props)
void *cdat;
ti_tableprops *props;
{
/*
The following lines assign the table properties to the client
data structure, and initialize the table properties
*/
    struct cdata *mydata;
    mydata = (struct cdata *)cdat;
    mydata->tableprops = NULL;
/*
The following line allocates memory to hold the table
properties
*/
    if ((mydata->tableprops = (ti_tableprops *)
           malloc (sizeof(ti_tableprops))) == NULL) {
           fprintf(stderr, "Malloc error\n");
           exit(-1);
    }
/*
The following line adds the table properties as defined by
*props to the client data structure *mydata-> tableprops
*/
    *mydata->tableprops = *props;

    return(FALSE);

};

/*
The following routine, getcolumnprops, retrieves the table
column properties and places them into the client data
structure
*/
ti_stop getcolumnprops(cdat, columns)
void (*cdat);
ti_colinfo columns;
{
    int i;
    struct cdata *mydata;
    di_tcont tcont;
    di_enumprocs hdrenums;
    dp_bool stop = FALSE;
    char *hdrstring = NULL;
    int vpcolwidth, unixcolwidth;

    mydata = (struct cdata *)cdat;
/*
The following lines allocate memory for the table column
properties
*/
    if ((mydata->colprops = (ti_colinfoseq *)malloc
            (sizeof(ti_colinfoseq))) == NULL) {
            fprintf(stderr, "Malloc error\n");
            return(TRUE);
     }

    mydata->colprops->length = columns->length;
    mydata->colprops->seq = NULL;
/*
The following lines allocate memory for the data stored
within the columns as well as the table width
*/
    if ((mydata->colprops->seq = (ti_colinforec *) malloc
            (sizeof (ti_colinforec)*columns->length)) ==
            NULL) { fprintf(stderr, "Malloc error\n");
                    return(TRUE);
    }

    if ((mydata->tablewidth = (int *)malloc(sizeof(int) *
            columns->length)) == NULL) {
            fprintf(stderr, "Malloc error\n");
            return(TRUE);
     }
/*
The following lines copy the column properties plus convert
the column widths to characters and store it in the main data
structure
*/
    for (i = 0;i < columns->length; i++) {

      if (columns->seq[i].divid) {
            fprintf(stderr, "Divided columns not supported,
                    operation aborted\n");
            return(TRUE);
      }

      mydata->colprops->seq[i] = columns->seq[i];
/*
The following lines get the column widths in characters and
put them into the array in the main data structure
*/
      vpcolwidth = mydata->colprops->seq[i].width;

      if (micatospace(vpcolwidth, &unixcolwidth) )
```

```
                return(TRUE);

        mydata->tablewidth[i] = unixcolwidth;
        }
/*
The following lines print a horizontal line to separate the
header from the table contents
*/
        printhorizontalline(mydata, FALSE);
        putc('|', mydata->fp);
/*
The following lines set the type to be text and set all
elements of di_enumprocs to NULL
*/
        tcont.type = TC_TEXT;
        if (di_getenumprocsdef (&hdrenums)) return(TRUE);

        hdrenums.text = readtext;
        for (i = 0;i < columns->length; i + +) {

                tcont.h.text = mydata-> colprops->
                        seq[i].hdentry.cont.u.text;

                mydata->curcol = i;
/*
The following lines enumerate the header information
*/
                if (di_enumerate(&tcont, &hdrenums, &hdrstring,
                        FALSE, &stop)) return(TRUE);
/*
The following lines print the header text in the ASCII output
file
*/
                if (addtexttofile(hdrstring, mydata)) return(TRUE);

        }
/*
The following line places a newline character into the ASCII
output file
*/
        putc('\n', mydata->fp);
        return(FALSE);
}
/*
The following routine, rowenum, enumerates the table rows
and prints the row contents to the ASCII output file
*/
ti_stop rowenum(cdat, cont)
void (*cdat);
ti_rowcont cont;
{

        di_enumprocs rowenums;
        dp_bool stop;
        int i, j, k;
        struct cdata *mydata;
        di_tcont tcont;
```

```
    char *curstring = NULL;
    int vpcolwidth, unixcolwidth, len;

    mydata = (struct cdata *)cdat;

    tcont.type = TC_TEXT;
/*
The following line sets all elements of di_enumprocs to NULL
*/
    if (di_getenumprocsdef (&rowenums)) return(TRUE);
/*
The following lines identify the call-back procedures
readtext and paraproc for the row enumeration process
*/
    rowenums.text = readtext;
    rowenums.newpara = paraproc;
/*
The following line prints a horizontal line in the output ASCII
file to separate the row
*/
    printhorizontalline(mydata, TRUE);
    putc('|', mydata->fp);
/*
The following lines go through the row column by column
and enumerates the textual content of each cell
*/
    for (i = 0; i < cont->length; i++) {

      mydata->curcol = i;
     tcont.h.text = cont->rowdat[i].cont.u.text;

     if (di_enumerate(&tcont, &rowenums, &curstring,
           FALSE, &stop))
            return (TRUE);
/*
The following line adds the row contents to the output ASCII
file
*/
      if (addtexttofile(curstring, mydata)) return (TRUE);
    }
/*
The following line adds a newline character to the output
ASCII file
*/
    putc ('\n', mydata->fp);
    mydata->currow++;

    return(FALSE);
}
/*
The following routine, readtext, reads the table contents
into the ASCII string passed in cdat
*/
dp_bool readtext(cdat, foprops, text)
void (*cdat);
dp_fontprops *foprops;
XString text;
```

```
{

    char **curstring, *ascii;
    char dummy = '?';

    curstring = (char **)cdat;
/*
The following lines allocate memory to hold the text string
*/
    if ((ascii = (char *)malloc(XStrlen(text) + 1)) = = NULL) {
            fprintf(stderr, "Malloc error\n");
            exit (-1);
    }

    XStrtoASC(text, ascii, dummy);

    if (*curstring = = NULL)
            *curstring = ascii;
    else {
/*
The following lines allocate more memory since an existing
string is being appended to
*/
            *curstring = (char *)realloc (*curstring, strlen
                    (*curstring) + strlen(ascii) + 1);
            strcat(*curstring, ascii);

    }

    return(FALSE);

}
/*
The following routine, addtextofile, writes text to the
output ASCII file
*/
dp_bool addtexttofile(text, mydata)
char *text;
struct cdata *mydata;
{
    int len, colwidth;

    colwidth = mydata->tablewidth[mydata->curcol];
/*
The following lines check the length of the text string, and
truncates the text string if it is longer than the column width
*/
    if ((len = strlen (text)) > = colwidth ) {
            fprintf(stdout, "The cell content in column %d,
                    row %d is too long for cell width. The text
                    has been clipped\n", mydata->curcol + 1,
                    mydata->currow + 1);
            text[colwidth] = '\0';
    }

    fprintf(mydata->fp, "%-*s|", colwidth, text);
    if (len > 0) {
            free(text);
            text = NULL;
    }

    return (FALSE);
}
/*
The following routine, paraproc, does nothing, but it makes
sure that FALSE is returned from the enumerate procs
*/
dp_bool paraproc(cdat, foprops, prprops)
void (*cdat);
dp_fontprops *foprops;
dp_paraprops *prprops;
{
return (FALSE);
}
/*
The following routine, errorexit, is a general routine for
printing error messages and exiting when an error is
encountered
*/
void errorexit(vpdoc, unixdoc)
di_doc *vpdoc;
FILE *unixdoc;
{
    fprintf(stderr, "Error reading table: signo =
            %x\n",getsigno());

    if (vpdoc != NULL) di_close(vpdoc);
    if (unixdoc != NULL) fclos (unixdoc);

    exit(1);

}
/*
The following routine, printhorizontalline, prints a
horizontal line in the output ASCII file to separate rows,
headers, etc.
*/
void printhorizontalline(mydata, withvert)
struct cdata *mydata;
dp_bool withvert;
{
    int i,j;

    if (withvert) putc('|', mydata->fp);
    else putc('-', mydata->fp);

    for (i = 0;i < mydata->colprops->length; i++) {
    for (j = 0; j < mydata->tablewidth[i]; j++)
                putc('-', mydata->fp);

            if (withvert) putc('|', mydata->fp);
            else putc('-', mydata->fp);
```

```
        }
        putc('\n', mydata->fp);

        }
    /*
    The following routine, micatospace, converts the width in
    micas into the width in characters.  This is not accurate and is
    shown here merely as an example.
    */
    int micatospace(mica, space)
    int mica, *space;
    {
        if (mica < = 0) return(-1);
        else *space = mica/214;
        return(0);
    }
```

Linking programs with the Document Interfaces Toolkit library

Typically, document-specific programs generated with the Document Interfaces Toolkit will only need to be linked with the *libdoctk.a* library. An example of a makefile is shown below:

```
#File:MyProgram.mk

CC = cc -g $(CFLAGS)
T = MyProgram
TOOLKITLIB = /usr/new/lib/libdoctk.a

all: program
program: $(T)

$(T): $(T).c
    $(CC)$(T).c $(TOOLKITLIB) -o $(T)

#end of Makefile
```

You will need to include the appropriate header files in your program. The *Document interface library* section of this manual lists the header files and which functions are associated with each header file.

Document interface function calls

An alphabetical listing of all functions provided by the Document Interfaces Toolkit specific to documents is listed below.

di_abort()	terminate the document generation process
di_apaframe()	append an anchored frame
di_apbreak()	append a page break character
di_apchar()	append one or more instances of a specified text character to a string

di__apfield()	append a document field
di__apfntile()	append a Footnote Reference Tile
di__apfstyle()	append font properties to the styles in a document
di__apindex()	append an index character
di__apnewpara()	append one or more new paragraph characters
di__appfc()	append a page format character
di__appstyle()	append paragraph style properties to the styles in a document
di__aptext()	append the text string specified
di__aptofillin()	append an item to the fill-in order of fields and tables
di__aptotxtlnk()	append an item to the end of the text frame link order
di__clearfillin()	cancel the previously specified fill-in order for the entire document
di__cleartxtlnk()	clear the text frame link order of a document
di__close()	release the document handle of a document being enumerated
di__enumerate()	parse the contents of a document
di__enumfillin()	enumerate the fill-in order of fields and tables
di__enumstyle()	enumerate all the font and paragraph style rules in a document
di__enumtxtlnk()	enumerate the link order of a text frame
di__finish()	finalize a document
di__getfieldfromname()	search for a named field and list the properties of that field
di__getfnprops()	obtain the footnote properties of a document
di__getmode()	get the mode of properties for a document
di__open()	open a file
di__relcap()	release caption resources
di__relfield()	release field resources
di__relfoot()	release footer resources
di__relhead()	release header resources
di__relindex()	release index resources
di__relnum()	release number resources
di__reltext()	release text resources
di__setfnprops()	set the footnote properties of a document
di__setmode()	set the mode properties of a document
di__setpara()	modify the paragraph properties of paragraphs in a specific text container
di__start()	initiate the document generation process
di__startap()	acquire a file handle for appending information to an existing document
di__starttext()	initiate the process of appending text to the body of an anchored text frame

di__textforaframe()	extract text from an anchored frame during enumeration
dp__colfromname()	retrieve the integer equivalent of a well-known color
dp__enumfrun()	perform an action defined by the client for each fontrun in an XString
dp__getbaspropsdef()	get default basic properties
dp__getbreakdef()	get the default properties of page break objects
dp__getcolwidthdef()	get the default column width properties
dp__getfielddef()	get the default properties of a field
dp__getfnnumdef()	get the default footnote numbering properties
dp__getfontdef()	get the default font properties
dp__getfontelmarralltrue()	initialize all font element properties to TRUE
dp__getfontstyledef()	get the default font style properties
dp__getframedef()	get the default anchored frame properties
dp__getindexdef()	get the default index properties
dp__getmodedef()	get the default properties of the commands in the document and auxiliary menus of a VP document
dp__getpagedef()	retrieve the default properties associated with a specific page in a VP document
dp__getpagedel()	get the default properties of the page number delimiter
dp__getrundef()	get the default fontrun properties
dp__gettabstopdef()	get the default tab stop as follows
dp__getparaelmarralltrue()	initialize all paragraph element properties to TRUE
dp__getparadef()	get the default paragraph properties
dp__getparastyledef()	get the default paragraph style properties
dp__gettframedef()	get the default text frame properties
dp__gettoc()	get the default table of content character properties
dp__namefromcol()	retrieve the name of a color based upon the data that defines the well known color
dp__wkcolfromcol()	retrieve a well known color from color
dsktp__checkuser()	verify the identity of a user accessing the Desktop
dsktp__copydoc()	copy a document and return a reference, or pointer, to the duplicate document
dsktp__deletefolder()	remove a folder from within another folder or from the desktop
dsktp__deletedoc()	remove a document from within a folder, a nested folder, or on the desktop
dsktp__enumerate()	enumerate the documents in a folder, a nested folder, or on the desktop
dsktp__getaccess()	ascertain the status and access permissions of the Desktop
dsktp__getdocref()	acquire a reference, or pointer, to a document on a desktop
dsktp__getdocpropsref()	obtain the properties of a document on the Desktop

dsktp__makefolder()	create a folder on the desktop or within an existing folder on the desktop
dsktp__movedoc()	move a document to a folder, to a nested folder, or onto the desktop
getsigno()	retrieve the number of an error
gi__adbacht()	add a bar chart to a graphic container
gi__adbm()	add a bitmap graphic to a graphic container
gi__adcurve()	add a curve to a graphic frame
gi__adellipse()	add an ellipse to a graphic container
gi__adffield()	add a form field to a graphic frame
gi__adline()	add a line to a graphic container
gi__adlncht()	add a line chart to a graphic container
gi__adpicht()	add a pie chart to a graphic container
gi__adpislce()	add a pieslice object to a graphic container
gi__adpoint()	add a point to a graphic container
gi__adrectangle()	add a rectangle to a graphic container
gi__adtable()	add a table frame to a graphic container
gi__adtframe()	add a text frame to a graphic container
gi__adtriangle()	add a triangle to a graphic container
gi__apchartobtnprog()	add a character to a CUSP button program
gi__apnparatobtnprog()	add a new paragraph to a CUSP button program
gi__aptexttobtnprog()	add a string to a CUSP button program
gi__btnforaframe()	extract the properties of a button in an anchored CUSP button frame during enumeration
gi__enumbtnprog()	enumerate the properties and text contents of a CUSP button
gi__enumerate()	read the contents of a graphics frame
gi__finishbtn()	terminate the creation of an anchored button
gi__finishcht()	indicate that no more objects are to be added to a chart
gi__finishcluster()	indicate that no more objects are to be added to a cluster and free the allocated resources
gi__finishgframe()	indicate that no more objects are to be added to a graphic container
gi__finishgr()	indicate that no more objects are to be added to a graphic container and free the allocated resources
gi__finishnbtn()	indicate that no more objects are to be added to a graphic container and free the allocated resources
gi__getbachtpropsdef()	get default bar chart properties
gi__getbmdispdef()	get default bit map display properties
gi__getbmpropsdef()	get the default bit map properties
gi__getbmscalpropsdef()	get the default bit map scale properties

gi__getboxdef()	get the default box properties
gi__getchtappdef()	get the default chart appearance properties
gi__getchtdatdef()	get the default chart data properties
gi__getcurvepropsdef()	get the default curve properties
gi__getellipsepropsdef()	get the default ellipse properties
gi__getframepropsdef()	get the default frame properties
gi__getgframeprops()	retrieve the name, description, and grid properties of an anchored graphic frame
gi__getgframepropsdef()	get the default graphic frame properties
gi__getgridpropsdef()	get the default grid properties
gi__getlinepropsdef()	get the default line properties
gi__getlnchtappdef()	get the default line chart appearance properties
gi__getlnchtpropsdef()	get the default line chart properties
gi__getpichtpropsdef()	get the default pie chart properties
gi__getpislcepropsdef()	get the default pie slice properties
gi__getpointpropsdef()	get the default point properties
gi__getrectanglepropsdef()	get the default rectangle properties
gi__gettframepropsdef()	get the default text frame properties
gi__gettrianglepropsdef()	get the default triangle properties
gi__relbtnprog()	release handles obtained by calls to **gi__startbtn()** or **gi__startnbtn()**
gi__setgframeprops()	set the properties of a graphics frame
gi__startbtn()	create an anchored CUSP button
gi__startcluster()	create a graphic cluster
gi__startgframe()	nest a graphic frame within a graphics container
gi__startgr()	create a graphic frame in a document
gi__startnbtn()	create a CUSP button in an anchored frame
ti__appendrow()	add a row to a table
ti__deffont()	assign default font property values to table elements
ti__defpara()	assign default paragraph property values to table elements
ti__enumtable()	parse the contents of a table
ti__finishtable()	close table being edited
ti__getcolinforecdef()	get default column properties
ti__gethdentrydef()	get default header entry properties
ti__getlinedef()	get default line properties
ti__getrowentdef()	get default row properties
ti__getsortkeydef()	get default sort key properties
ti__gettablepropsdef()	get default table properties

ti_gettableprops()	extract the properties of a named table
ti_maxelm()	estimate the maximum number of cells that may be added to a table in a document
ti_startextable()	prepare table to receive new data
ti_starttable()	add a new table to a document
XCharcode()	retrieve the XChar character code (the lower 8 bits)
XCharmake()	create characters having specific visual qualities, based upon the character set defined in the Xerox Character Set Standard and the character codes defined in Xerox Character Code Standard
XCharset()	retrieve the XChar character set (the higher 8 bits) of a character
XStrcasecmp()	compare two strings, while ignoring the case of ASCII characters
XStrchr()	parse a string in search of a specific character
XStrcaselexcmp()	compare two strings, while ignoring the case of ASCII characters and while sorting the character strings in the order specified in **sortorder**
XStrcat()	concatenate one string to the end of another string
XStrcmp()	compare two strings
XStrcpy()	copy a specific string into a VM storage area
XStrdup()	copy a text string into a storage area and return a pointer to that area
XStrfromASC()	convert an ASCII string to an XString
XStrfromXCC8()	convert an XCCS 8-bit encoded string into an XString string
XStrlen()	determine the logical length of an XString character
XStrlexcmp()	compare two strings according to a sort order
XStrncasecmp()	compare a portion of one string against another string, while ignoring the case of ASCII characters
XStrncaselexcmp()	compare a portion of one string against another string, while ignoring the case of ASCII characters
XStrncat()	copy a specific number of characters from one string and append them to the end of another string
XStrncmp()	compare a portion of one string against another string
XStrncpy()	copy a specific number of characters in a text string
XStrnlexcmp()	compare two strings, while ignoring the case of ASCII characters and while sorting the character strings in the order specified in **sortorder**
XStrpbrk()	search a string for the occurrence of any character contained within another string
XStrrchr()	search for a character within a string, starting from the end of the string and proceeding forward towards the beginning of the string
XStrsch()	determine if a string is contained within another string
XStrsep()	separate a string into tokens based upon one or more delimiter characters

XStrtoASC()	convert the XString into an ASCII string
XStrtoXCC8()	convert an XString into a compact XCCS 8-bit encoded string

5. Document function examples

This chapter present example code fragments for the document-specific functions to illustrate their usage. Only the major functions have been listed. See chapter 4 for more detailed examples for creating and enumerating documents, tables, and graphics. Consult the *Document Interfaces Toolkit Reference Manual* for a complete description of all the available XNS functions.

The examples are organized alphabetically and have the following format:

- Function name
- Function brief description
- Function syntax
- Example code fragment
- Description of the example

di__abort()

The **di__abort()** function is used to terminate the document generation process and deallocate the storage resources allocated to that document.

di__abort(doc)

Example

```
if (ret.stat != SC_OK) {
    fprintf(stderr, "ret_sc status = %d\n", ret.stat);
    if (di_abort(&ret.doc)) exit(-1);
    exit(1);
}
```

In the preceding example, the return status from a **di__start()** function is checked to see if the call returned successfully. If the status does not match **SC__OK**, then an error message will be printed, and the document generation process will be aborted. **ret.doc** is the file handle returned by the **di__start()** function, and is used by the **di__abort()** function to determine which document to finalize.

di__apaframe()

The **di__apaframe()** function is used to append an anchored frame to a text container.

di__apaframe(to, type, frame, cont, wtcap, wbcap, wlcap, wrcap, font, trustsize, ret)

Example

```
type = AF_TEXT;
contents = "Appending anchored text frame"
if (di_apaframe(&to, type, NULL,contents, FALSE, TRUE,
FALSE, FALSE, NULL, FALSE, &anc_ret)) exit (-1);
```

In the preceding example, an anchored text frame is appended to the document container specified by **&to**. **type** specifies the type of anchored frame to be appended, and **contents** specifies the contents to be appended. In this example, **type** is defined as **AF_TEXT** or a text frame. Setting the argument **frame** to NULL specifies that the default **dp_frameprops** variables should be used. The arguments **wtcap**, **wbcap**, **wlcap**, and **wrcap** specify the location of the caption. In the above example, the caption will be placed at the bottom, as defined by the TRUE value of **wbcap**. **anc_ret** will return a handle to the caption so that its contents can be defined and appended. Setting **font** to NULL means that the anchored frame will have default font properties. Setting **trustsize** to FALSE means that the frame size specified by **frame** will be ignored and the frame will be adjusted to fit the existing frame.

di_apfield()

The **di_apfield()** function is used to append a document field to a text container.

di_apfield(to, fiprops, foprops, ret)

Example

```
void (*cdat);
dp_fontprops *fontprops;
dp_fldprops *fieldprops;
di_tcont *to;
di_field ret;
...
to = (di_tcont*)cdat;
if (di_apfield(to, fieldprops, fontprops, &ret)) return (TRUE);
```

In the preceding example, a doc field is appended to a text container specified by **to**. The field and font properties are specified by **fieldprops** and **fontprops**, respectively. The default values for **fieldprops** and **fontprops** can be obtained through calls to the **dp_getfielddef()** and **dp_getfontdef()** functions. **ret** can be passed to other **di_ap*()** functions in order to add data to the appended field.

di_aptext()

The **di_aptext()** function is to append a text string to a text container.

di_aptext(to, text, foprops)

Example

```
dp_fontprops *foprops;
char insert_string[256];
XString text_to_add;
di_tcont *to;
insert_string = "This is the Document Interfaces Toolkit User
    Guide"
...
XStrfromASC(text_to_add, insert_string);
to = (di_tcont*)cdat;
```

```
if (di_aptext(to, text, foprops)) return (TRUE);
```

In the preceding example, the text string "This is the Document Interfaces Toolkit User Guide" is appended to the text container specified by **to**. The text string is translated into an XString format through the **XStrfromASC** function.The font properties of the text string are specified by **foprops**. The default properties of **foprops** can be obtained through a call to **dp_getfontdef()**.

di__aptofillin()

The **di__aptofillin()** function is used to append an item to the fill-in order of fields and tables.

di__aptofillin(doc, name, type)

Example

```
void (*cdat);
char * insert_string = "Field1";
XString name;
di_fillintype type;
di_doc target;
...
XStrfromASC(name, insert_string);
type = FI_FIELD;
target = (di_doc)cdat;
if (di_aptofillin(target, name, type)) {
     error_display ("AppendToFillin", 0);
     return(TRUE);
};
```

In the preceding example, the string specified by **insert_string** is appended to the fill-in order of the table specified by **target**. **FI_FIELD** signifies that a field is the object whose fill-in order will be appended to. **target** is returned from an earlier call to either **di_start()** or **di_startap()**. If the **di_aptofillin()** function does not return successfully in the preceding example, the user-defined procedure **error_display** is then called.

di__close()

The **di__close()** function is used to release the document handle of an enumerated document.

di__close(docptr)

Example

```
if (di_enumerate(&to, &procs, &data, mrgnum, &stops)) {
     error_display("Enumerate", 0);
     di_close(&ret.doc);
     exit(1);
};
```

In the preceding example, the document specified by the document handle **&ret** will be closed if the **di__enumerate()** function does not return successfully. **di__close()** will release the document handle and free the storage space originally allocated to it.

di__enumerate()

The **di__enumerate()** function is used to parse the contents of a document.

di__enumerate(to, procs, cdat, mrgnum,ret)

Example

```
to.type = TC__DOC;
to.h.doc = ret.doc;
data.source = ret.doc;
mrgnum = FALSE;
di__getenumprocsdef(&procs);
procs.text = textproc;
procs.field = fieldproc;
procs.index = indexproc;
...
if (di__enumerate(&to, &procs, &data, mrgnum, &stops)) {
    ...
    errorexit();
};
```

In the preceding example, the document specified by the document handle **&to** is to be enumerated. **to.type** has the value of **TC__DOC**, meaning that the text container is a document. The **di__getenumprocsdef()** function is used to set all elements of **di__enumprocs** to NULL so that the user can define the required call-back procedures. **&procs** defines the user-defined call-back procedures used to enumerate the document. In the above example, the call-back procedures **textproc**, **fieldproc**, and **indexproc** will be called when a text, field, and index object is encountered. If any of these call-back procedures returns a TRUE, then the enumeration process is halted. **mrgnum** has a value of FALSE, meaning that a page numbering pattern will not be included in the heading or footing during enumeration. **&stops** is a pointer to the user-defined data that is passed to the call-back procedures. **&stops** is set to TRUE if **di__enumerate()** encounters an object that it can not recognize, or if it encounters an object for which a client call-back procedure was not supplied.

di__enumstyle()

The **di__enumstyle()** function is used to enumerate all the font and paragraph style properties of a document, such as mode, fill-in order,and text-link.

di__enumstyle(doc, fstyleproc, pstyleproc, cdat)

Example

```
di__style style;
void (*cdat);
di__apfstyleproc (*fstyleproc);
di__appstyleproc (*pstyleproc);
{
    myStyle mydat;
    myData *data;
    ...
    data = (myData *)cdat;
```

```
        mydata.style = style;
        mydata.fstyleproc = fstyleprocedure;
        mydata.pstyleproc = pstyleprocecure;
        di_enumstyle(data->source, fstyle_callback,
              pstyle_callback, &mydat);
    }
```

In the preceding example, the font and paragraph style properties of the document specified by **data->source** will be enumerated. This document handle is obtained from an earlier call to **di__open()** or **di__startap()**. **fstyle__callback** and **pstyle__callback** are user-defined call-back procedures to be invoked once for each object in the style. If any of the call-back procedures returns TRUE, the enumeration will be halted.

di__setmode()

The **di__setmode()** function is used to set the mode properties of a document.

di__setmode(doc, props, select)

Example

```
    di_doc source;
    di_doc target;
    {
        dp_modeprops props;
        dp_modesel select;
        unsigned int i;

        for (i = 0; i< = ME_PROMPT; i + + ) {
              select[i] = TRUE;
        };
        if (di_getmode(source, &props)) return(-1);
        ...
        if (di_setmode(target, &props, select)) return(-1);
        return(0);
    }
```

In the preceding example, the mode properties of the document specified by the document handle **target** are set. Mode properties display the structure, non-printing characters, cover sheets, and prompt fields in a document. In the above example, the **for** statement defines which properties will be affected. Those properties which are designated by TRUE will be changed. **di__getmode()** was called to get the default mode properties which can then be modified to suit the user's needs. **di__setmode()** is then called to set the mode properties of the document. **&props** defines the display characteristics of the document.

di__start()

The **di__start()** function is used to initiate the document generation process.

di__start(pagiops, whead, wfoot, wnum, ifoprops, iprprops, ipgprops, styledat, ret)

Example

```
    if (di_start(PO_COMPRESS,TRUE, FALSE, FALSE, NULL,
        NULL, NULL, NULL, &ret)) {
        printf("Error in di_start, getsigno = %x\n", getsigno());
        exit(-1);
    };
```

In the preceding example, an empty document will be created with the following properties: compressed pagination, header properties inserted into the first page format character, no footing or numbering properties inserted into the first page format character, default initial font, paragraph, and page properties, default values for the first new paragraph and page format characters. The **&ret** handle can be passed to other **di_ap*()** functions to add information to the document. More information on default properties can be found in the *Document Interfaces Toolkit Reference Manual.*

di_starttext()

The **di_starttext()** function is used to initiate the process of appending text to the body of an anchored text frame.

di_starttext(doc, frame, props, ret)

Example

```
    di_text desttext;
    dp_gettframedef (&tframepropsref);
    if (di_starttext(data->target, ret.frame, &tframepropsref,
    &desttext)) {
        error_display("Start Text", 0);
        return(TRUE);
    };
```

In the preceding example, the text frame defined by the **ret.frame** is "readied" to accept new text. **ret.frame** is the frame handle returned by an earlier call to **di_apaframe()**. The properties of the text frame are defined by **&tframepropsref**, which in turn is defined by **dp_gettframedef()**. **&desttext** returns a handle which then can be passed to various **di_ap*()** functions to add the actual data. **data->target** is the document handle returned by an earlier call to **di_start()** or **di_startap()**. In the above example, if **di_starttext()** does not return successfully, then a user-defined procedure, **error_display()**, will return an error status.

dp_colfromname()

The **dp_colfromname()** function is used to retrieve the integer equivalent of a well known color.

dp_colfromname(name, ret)

Example

```
    ret_wkcolfromname retcolor;
    dp_fontprops foprops;
    dp_color *color;
    ...
    if (dp_getfontdef(&foprops)) errorexit();
    if (dp_colfromname(CL_GREEN, &retcolor)) errorexit();
    foprops.txtcol = retcolor.color;
```

In the preceding example, the color of a text string is defined to be green. **CL__GREEN** defines the desired color, and **&retcolor** returns a structure whose member is an array of three integers that specifies the color green. This member is passed to **foprops.txtcol** to assign the text the color of green. A complete list of available colors is shown in the *Document Interfaces Toolkit Reference Manual.*

dsktp__movedoc()

The **dsktp__movedoc()** function is used to move a document to a folder, a nested folder, or the desktop.

dsktp__movedoc(ref, dstpath, name, vers)

Example

```
    ascii = argv[1];
    name = (XString)malloc((strlen(ascii) + 1) *2);
    XStrfromASC(name, ascii);
    dsktp_movedoc(&ret2.ref[0], NULL, name, &vers);
```

In the preceding example, the document defined by its handle &**ret2.ref[0]**, and its name, defined by **name** is moved to the desktop. A value of NULL for **dstpath** signifies that the document is to be placed onto the desktop, not a folder or a nested folder. The name of the document in this example was taken from the parameter **argv[1]** and translated into an XString format so that it may be understood by the GLOBALVIEW environment.

getsigno()

The **getsigno()** function is used to determine the cause for the failure.

getsigno()

Example

```
    if (di_start(PO_COMPRESS, FALSE, FALSE, FALSE, NULL,
        NULL, NULL, NULL, &ret)) {
        printf("Error in di_start, getsigno = %x\n", getsigno());
        exit(-1);
    };
```

In the preceding example, if **di__start()** does not complete successfully, then **getsigno()** is called to retrieve the error number returned by **di__start()**. The text description of the error number may be obtained through the *Signals.h* header file or the *Document Interfaces Toolkit Reference Manual.*

gi__apnparatobtnprog()

The **gi__apnparatobtnprog()** function is used to add a new paragraph character to the button program.

gi__apnparatobtnprog(to, paprops, foprops, num)

Example

```
    void (cdat;
    dp_fontprops *foprops;
```

```
dp__paraprops *prprops;
...
gi__buttonprog to;
to = (gi__buttonprog)cdat;
if (gi__apnparatobtnprog(to, prprops, foprops, 1))
    return(TRUE)
```

In the preceding example, one new paragraph character is added to the CUSP button defined by its handle **to**. **to** is returned from an earlier call to **gi__startbtn()** or **gi__startnbtn()**. The properties of this new paragraph character is defined by prprops and foprops. These properties may be defined by calls to **dp__getfontdef()** and **dp__getparadef()**. **num** has the value of 1, meaning only one new paragraph character is to be added.

ti__appendrow()

The **ti__appendrow()** function is used to add a row to a table.

ti__appendrow(h, rc)

Example

```
if (ti__getrowentdef(rowContSeq.rowdat)) {
...
}
rowContSeq.tmgn = 139;
rowContSeq.valign = VA__CENTER
for (i = 0; i <ROW__NUM; i + + ) if (ti__appendrow(ret__h,
    &rowContSeq)) {
    free(colInfoSeq.seq);
    free(rowContSeq.rowdat);
    errorexit();
};
```

In the preceding example, the default row entry properties are retrieved, and then added to the table defined by the table handle **ret__h**. Information on retrieving default row entry properties can be found in the *Document Interfaces Toolkit Reference Manual*. **ret__h** may be obtained by an earlier call to either **ti__starttable()** or **ti__startextable()**. The number of rows to be added is defined by the parameter **ROW__NUM**. If the **ti__appendrow()** function is not successful, then the memory allocated to the column and row data is freed via the UNIX free command, and a user-defined **errorexit()** procedure is called.

ti__getcolinforecdef()

The **ti__getcolinforecdef()** function is used to get the default column properties.

ti__getcolinforecdef(col)

Example

```
if (ti__getcolinforecdef(colInfoSeq.seq)) {
    ...
    errorexit();
}
colInfoSeq.seq->width = 3000;
colInfoSeq.seq->lmgn = 150;
```

In the preceding example, the default column properties are retrieved and placed into the argument **colInfoSeq.seq**. This example also shows how some of the column properties may be modified from their default values. The width of the column, defined by **colInfoSeq.seq->width**, has been changed to 3000 micas. The left margin of the column, as defined by **colInfoSeq.seq->lmgn**, is set to 150 micas. A complete list of default column properties is defined in the *Document Interfaces Toolkit Reference Manual.*

ti_getrowentdef()

The **ti_getrowentdef()** function is used to get the default row entry properties.

ti_getrowentdef(rowentry)

Example

```
if (ti_getrowentdef(rowContSeq.rowdat)) {
    ...
    errorexit();
}
rowContSeq.tmgn = 120;
rowContSeq.line.style = LS_SOLID;
rowContSeq.valign = VA_CENTER;
```

In the preceding example, the default row entry properties are retrieved and placed into the argument **rowContSeq.rowdat**. This example also shows how some of the row entry properties may be modified from their default values. The top margin, line style, and text alignment as specified by **rowContSeq.tmgn**, **rowContSeq.line.style** and **rowContSeq.valign** respectively, are changed. The top margin is set to 120 micas. The line separating the rows is defined to be a solid line. The text within each row will be centered. A complete list of default row entry properties is defined in the *Document Interfaces Toolkit Reference Manual.*

ti_gettablepropsdef()

The **ti_gettablepropsdef()** function is used to get default table properties.

ti_gettablepropsdef(props)

Example

```
if (ti_gettablepropsdef(&propsRec)) {
    free(colinfoSeq.seq);
    free(rowContSeq.rowdat);
    errorexit();
}:
propsRec.bdrline = LW_SOLID;
propsRec.bdrline.width = LW_W3;
propsRec.dvrline.width = LW_W2
```

In the preceding example, the default table properties are retrieved, and are placed into the argument **&propsRec**. If the function call does not return successfully, then the memory allocated to the column and row properties are freed, as defined by the UNIX **free** command. **errorexit()** is a user-defined procedure in case of an unsuccessful return. The above example also shows how some properties may be changed. The table border, **propsRec.bdrline**, and the divider line (the line between the header row and the rest of the table), **prosRec.dvrline**, are modified. The border line now has a solid line, and a width of three pixels. The divider line has been changed to have a line width of two pixels. A complete listing of the default table properties and their values can be found in the *Document Interfaces Toolkit Reference Manual.*

ti_starttable()

The **ti_starttable()** function is used to add a new table to a document.

ti_starttable(doc, props, col, ret)

Example

```
if (ti_starttable(*doc, &propsRec, &collnfoSeq, &ret_h)) {
    free(collnfoSeq.seq);
    free(rowContSeq.rowdat);
    errorexit();
};
```

In the preceding example, a table will be added to the document specified by the ***doc** document handle. The table properties are defined by the **&propsRec** argument. Default table properties may be obtained from the **ti_gettablepropsdef()** function, and later modified if required. **&collnfoSeq** defines the column properties for the table. The default column properties may be obtained from the **ti_getcolinforecdef()** function. The returned argument, **&ret_h** is a pointer to an opaque variable that contains the table handle. If the **ti_starttable()** does not return successfully in the preceding example, the memory allocated for the column and row data will be freed, and a user-defined procedure, **errorexit()**, will be called.

XStrcat()

The **XStrcat()** function is used to concatenate one string to the end of another string.

XStrcat(xs1, xs2)

Example

```
ascii = ".CV"
ext = (XString)malloc(strlen(ascii) + 2) * 2);
XStrfromASC(ext,ascii);
name = XStrcat(name, ext);
}
```

In the preceding example, the string ".CV" is concatenated to the contents of name. Memory is allocated for ".CV" before it is translated to an XString format and later concatenated. The procedure for allocating memory is defined in the ***Document Interfaces Toolkit Reference Manual***.

XStrfromASC()

The **XStrfromASC()** function is used to convert an ASCII string to an XString.

XStrfromASC(xs, s)

Example

```
ascii = "This is the Document Interfaces Toolkit User Guide";
name = (XString)malloc((strlen(ascii) + 1)*2);
name = XStrfromASC(name, ascii);
```

In the preceding example, the string"This is the Document Interfaces Toolkit User Guide", defined by **ascii**, is converted to the XString **name**. Memory is allocated for the XString before **XStrfromASC()** is called.The procedure for allocating memory is defined in the *Document Interfaces Toolkit Reference Manual*.

6. XNS services

This chapter presents guidelines for using the *libxnstk.a* library from the Document Interfaces Toolkit. It describes using the appropriate XNS header file, common XNS function arguments, handling courier errors, and special linking considerations.

The XNS functions of the Document Interfaces Toolkit are not as order-dependent as the document-specific functions. Each XNS function calls a specific service request. This service request may be dependent on only one other call preceding it. As such, a generalized template cannot be generated for each task.

This chapter will discuss the common elements of the XNS functions such as: the **__Connection** and **__BDTprocptr** argument found in all XNS functions, trapping for errors, linking procedures, and the recommended include files for each XNS service. Examples for the major XNS functions may be found in Chapter 7.

XNS services header files

The *libxnstk.a* library allows you access to the following XNS services:

- Printing
- Authentication
- Clearinghouse
- Gap
- Mailing
- Filing

Each XNS service has two header files associated with it, [service]__de.h and [service].h, where [service] represents the name of the service, such as *Printing3__de.h* and *Printing3.h.* The [service]__de.h header files have define statements that eliminate the need for prefixing functions and error statements with the service name. [service].h is the "raw" header file. If you include the [service].h header file, you must prefix the name of the header file to each function or error name. Using the function **Printing3__GetPrinterStatus** as an example, **Printing3__** is the prefix that you may or may not need to include. The following two examples compare these two approaches:

If you include *Printing3__de.h,* you may use the following method of coding:

```
# include "Printing3__de.h"
    ...
    StatusResult = GetPrinterStatus(
          getprintstatusconnected,NULL);
    ...
    switch (Exception.Code) {
```

```
    case ServiceUnavailable:
            msg = "GetStat: Service unavailable";
            break;
    ...
```

If you include *Printing3.h,* your call to these same procedures must look as follows:

```
# include "Printing3.h"
    ...
    StatusResult = Printing3__GetPrinterStatus(
            getprintstatusconnected,NULL);
    ...
    switch (Exception.Code) {
    case Printing3__ServiceUnavailable:
            msg = "GetStat: Service unavailable";
            break;
    ...
```

In the second case, **Printing3__** must precede both **GetPrinterStatus** and **ServiceUnavailable**. If, during compilation of your program, you encounter errors that report functions or error not found, you have probably included the wrong header file or you did not prefix your functions.

Note that you should include either the [service]__de.h or the [service].h header file, but not both.

__Connection and __BDTprocptr parameters

Every XNS function has two common parameters, **__Connection** and **__BDTprocptr**.

The **__Connection** parameter is the courier connection number for the XNS server the application is trying to reach. By using a particular number, a C application may, for example, talk to printer1 instead of printer2. This connection number may be obtained by using the following code in your C program:

```
COURIERFD   connected;
char        *hostnameptr;

if (!(connected = cour__establish__conn(hostnameptr))) {
    fprintf(stderr, "\t\tCOURIER CONNECTION FAILED!!!!\n");
    return;
}
```

In the above example **connected** is the return value of **cour__establish__conn**. **cour__establish__conn** is a function provided by the XNS basic software that checks the name of a specified pointer, hostnameptr in the preceding example. If the pointer is valid, then **cour__establish__conn** will try to connect to it. See the *XNS for UNIX System V.3 Programming Guide* for more information on this function. The **hostnameptr** parameter is a string that contains the name of the server you want to communicate with. You can now replace **connected** for **__Connection** in each applicable function. If the connection to the host does not occur, the above code will return an error message.

The **__BDTprocptr** parameter is the name of a function that performs the bulk data transfer. Bulk data is an arbitrarily long, ordered sequence of zero or more eight-bit bytes. Although any data may be modeled as bulk data, bulk data transfer is intended primarily for use in transporting objects that may be too large to be reasonably modeled as parameters or results of remote procedures. For example, directory listings and the contents of files are among the entities that may be modeled as bulk data. Not every XNS function requires the use of bulk data transfer. If a function does not require bulk data transfer, **__BDTprocptr** should be set to **NULL**. The following list shows the functions that require bulk data transfer:

- **Clearing2__List()**
- **Clearing2__ListAliases()**
- **Clearing2__ListAliasesOf()**
- **Clearing2__ListDomains()**
- **Clearing2__ListDomainsServed()**
- **Clearing2__RetrieveMembers()**
- **Filing6__Deserialize()**
- **Filing6__Serialize()**
- **Filing6__Store()**
- **Inbasket2__ChangeBodyPartsStatus()**
- **Printing3__Print()**

The bulk data function must be created by you, the C programmer. For example, suppose you want to print an interpress document which resides in a UNIX directory. Your C code might contain the following XNS print call:

```
printresult = Printing3__Print(printconnected, SendSource,
    BulkData1__immediateSource, attributes, options);
```

For the purpose of explaining bulk data transfer, it is necessary to review the syntax for the print function used above.

```
Printing3__Print(__Connection, __BDTprocptr, master,
    printAttributes, printOptions);
```

The second and third arguments, **__BDTprocptr** and **master**, are the two parameters related to bulk data transfer. The **master** argument describes the bulk data type. It tells the function that data will be sourced from the UNIX environment or sinked into the UNIX environment. To specify source, set the value of **master** to be **BulkData1__immediateSource**. To specify sink, set the value of **master** to be **BulkData1__immediateSink**. The XNS print example above sourced data from the UNIX environment and therefore used **BulkData1__immediateSource**. **__BDTprocptr** is the name of your C program that sends print data from your UNIX environment to the XNS printer described by **__Connection**, the courier connection number. In the XNS print example above, "SendSource" is the name of the C program that sends the print data. An example of the SendSource program is shown below:

```
    int
    SendSource (bdtconnection)
        COURIERFD
{
        char      * buf;
        int       buflen;
        int       count;
        extern    int errno;
        int         len;
        char        local__buf[BUFSIZ];

        len = sizeof(local__buf) < < 3;
        if (len < = 0  | |  !(buf = malloc(len))) {
              buf = local__buf;
              len = sizeof(buf);
        }
        while ((count = read(ipfile, buf, len)) > 0) {
              if (cour__bdt__write(bdtconnection, buf, count) <
                  count) {
                  if (buf ! = local__buf)
                      free(buf);
                  return BDT__WRITE__ABORT;
              }
        }
        if (buf ! = local__buf)
              free(buf);
        return (count > = 0) ? BDT__WRITE__FINISHED :
              BDT__WRITE__ABORT;
}
```

If you want to retrieve data from the XNS server use **BulkData1__immediateSink** as in the following example:

```
Filing6__Retrieve(connected, bulkretrieveproc, remotehandle,
                  BulkData1__immediateSink, session);
```

In the above example, the bulk data transfer procedure that must be supplied is called **bulkretrieveproc**. It might look as follows:

```
int
bulkretrieveproc (connected)
    COURIERFD   connected;
{
    int    chars__to__save = 0;
    int    charset = 0;
    int    count;
    int    do__translate;
    int    residule__count;

    residule__count = 0;
    DURING {
          do__translate = (filetypevalue = =
              TYPE__VPMailNote | | filetypevalue = =
              TYPE__A);
          errno = 0;
          while ((count = cour__bdt__read(connected,
              &file__buf[chars__to__save], sizeof(file__buf) -
              chars__to__save)) > 0) {
              bytessent + = count;
              if (do__translate)
                  count = translate__and__write(fout,
                  file__buf, count + chars__to__save,
                  &charset, &chars__to__save);
              else
                  count = write(fout, file__buf, count);
              if (count < 0) {
                  perror("write");
```

```
                    signal(SIGINT, SIG__IGN);
                    E__RETURN(BDT__READ__ABORT);
                }
                if (hash) {
                    count + = residule__count;
                    for ( ; count > = 1024; count -= 1024)
                        putchar(' # ');
                    residule__count = count;
                    fflush(stdout);
                }
            }
            if (count < 0)
                perror("netin");
            else if (charset = = 0 && chars__to__save = = 1
                && file__buf[0] = = '\r') {
                if (write(fout, "\n", 1) < 1)
                    perror("write");
            }
        } HANDLER {
            signal(SIGINT, SIG__IGN);
            E__RETURN(BDT__READ__ABORT);
        } END__HANDLER;

        signal(SIGINT, SIG__IGN);
        return BDT__READ__FINISHED;
    }
```

This example does more than is necessary for bulk data retrieval, because it handles various types of Xerox documents. What is important here is that this procedures reads a specified file from the XNS server and writes it to the UNIX file given by **fout**. **cour__bdt__read**, provided by the XNS basic software, is the lower-level bulk data transfer read function. See the *XNS for UNIX System V.3 Programming Guide* for information on this function. More details on bulk data transfer are described in the *Xerox Filing Protocol* manual.

XNS error handling

You must write your own error handling routines to catch errors returned by the XNS library functions. The XNS service reference manuals for each of the six respective service explains the error. Both the *Document Interfaces Toolkit Reference Manual* and the respective XNS service reference manuals show which errors are reported. Two errors that do not show up in the XNS service reference manuals are courier generated errors, **REJECT__ERROR** and **SYSTEM__ERROR**. These two errors may be generated when calling any XNS function. The example code below will show how to catch these errors.

Using the PrintingService **GetPrinterStatus** function as an example, the code fragment below switches on **Exception.Code**, **REJECT__ERROR**, and **SYSTEM__ERROR** to print the appropriate error message:

```
#include <courier/except.h>
#include <Printing3__de.h>
```

```
int         secondlevelerror, syserror;
Cardinal    probnum;

secondlevelerror= 0;
syserror = 0;
DURING
    StatusResult =
        GetPrinterStatus(getprintstatusconnected,NULL);
HANDLER {
    char * msg;
    switch (Exception.Code) {
    case ServiceUnavailable:
        msg = "GetStat: Service unavailable";
        break;
    case SystemError:
        msg = "GetStat: System Error";
        break;
    case Undefined:
        msg = "GetStat: Undefined error";
        probnum = CourierErrArgs(UndefinedArgs,problem);
        secondlevelerror = 1;
        break;

    case REJECT__ERROR:
        switch (CourierErrArgs(rejectionDetails, designator)) {
        case 0:
            msg = "GetStat: REJECT: noSuchProgramNumber";
            break;
        case 1:
            msg = "GetStat: REJECT: noSuchVersionNumber";
            break;
        case 2:
            msg = "GetStat: REJECT: noSuchProcedureValue";
            break;
        case 3:
            msg = "GetStat: REJECT: invalidArgument";
            break;
        default:
            msg = "GetStat: REJECT: unknown error";
            secondlevelerror = 1;
            probnum = CourierErrArgs(rejectionDetails,
                designator);
            break;
        }
        break;
    case SYSTEM__ERROR:
        msg ="GetStat: Connection Error";
        syserror = 1;
        break;
    default:
        msg = "GetStat: Unknown Error";
        secondlevelerror = 1;
        probnum = Exception.Code;
        break;
    }
    fprintf (stderr,"\t\t\tError: %s\n", msg);
```

```
        if (syserror) {
            syserror = 0;
            fprintf (stderr,"\t\t\t%s\n", Exception.Message);
        }
        if (secondlevelerror) {
            secondlevelerror = 0;
            fprintf (stderr,"\t\t\tProblem number: %d\n", probnum);
        }
    } END__HANDLER;
}
```

The **DURING**, **HANDLER**, and **END__HANDLER** macros are defined in the *except.h* header file. When an error occurs, the XNS function will return a code number and sometimes a problem number. The above example switches on the error code in order to print out the corresponding error message as defined in the case statements. If the error can also show which problem occurred, you can get the problem number by using the **CourierErrArgs** function. **CourierErrArgs** is a function provided by the basic XNS software that returns arguments for errors encountered. Refer to the *XNS for UNIX System V.3 Programming Guide* manual for more information on this function.

Linking with libraries and including headers

The following libraries should be linked when compiling your XNS program:

- *libxnstk.a*
- *libcourier.a*

The *libxnstk.a* library contains the XNS functions provided by the Document Interfaces Toolkit.

The *libcourier.a* library contains lower-level courier definitions. This library should be linked with all programs using the XNS portion of the Document Interfaces Toolkit.

An example of a make file that can be used to link your XNS programs is listed below:

```
#File:MyProgram.mk

CC = cc -g $(CFLAGS)
T = MyProgram

TOOLKITLIB = /usr/include/tk/libxnstk.a
MISC = /usr/new/lib/libcourier.a

all: program

program: $(T)

$(T): $(T).c
    $(CC) $(T).c $(TOOLKITLIB) $(MISC) -o $(T)

#end of Makefile
```

The following sections list the recommended include files for each XNS service.

Authentication service

```
#include <stdio.h>        /* contains standard I/O definitions */
#include <sys/types.h>    /* contains "system" types definitions
                             */
#include <errno.h>          /*  contains integer expression
                                definition for errors */
#include <signal.h>       /* contains facilities for handling
                             exceptional conditions */
#include <courier/except.h>   /* contains macro definitions
                                 for the exception handler */
#include <courier/Authenti2.h> /* contains Toolkit
                                 Authentication functions,
                                 can use Authenti2__de.h
                                 instead */
#include <courier/FilingSu1.h> /* contains the definition for
                                 the CH__StringToName
                                 function */
```

GAP service

```
#include <stdio.h>        /* contains standard I/O definitions */
#include <sys/types.h>    /* contains "system" types definitions
                             */
#include <errno.h>        /* contains integer expression
                             definition for errors */
#include <signal.h>       /* contains facilities for handling
                             exceptional conditions */
#include <courier/except.h>   /* contains macro definitions
                                 for the exception handler */
#include <courier/Gap3.h>     /* contains Toolkit Gap
                                 functions, can use
                                 Gap3__de.h instead */
#include <courier/gapcontrol.h>  /* contains service types for
                                    GAP TTY services */
```

Clearinghouse service

```
#include <stdio.h>        /* contains standard I/O definitions */
#include <sys/types.h>    /* contains "system" types definitions
                             */
#include <errno.h>        /* contains integer expression
                             definition for errors */
#include <signal.h>       /* contains facilities for handling
                             exceptional conditions */
#include <courier/except.h>   /* contains  macro definitions
                                 for the exception handler */
#include <courier/Clearing2.h> /* contains Toolkit
                                 Clearinghouse functions, can
                                 use Clearing2__de.h instead
                                 */
```

```
#include <courier/FilingSu1.h>  /* contains the
                                   CH__StringToName function
                                   definition*/
#include <courier/Authenti2.h> /* contains Toolkit
                                   Authentication functions,
                                   can use Authenti2__de.h
                                   instead */
#include <courier/filetypes.h>  /* contains definitions for
                                   various Xerox file types */
#include <courier/CHEntrie0.h> /* contains definitions for
                                   Clearinghouse properties */
```

Mailing service

```
#include <stdio.h>         /* contains standard I/O definitions */
#include <sys/types.h>     /* contains "system" types definitions
                              */
#include <errno.h>         /* contains integer expression
                              definition for errors */
#include <signal.h>        /* contains facilities for handling
                              exceptional conditions */
#include <courier/except.h>     /* contains macro definitions
                                   for the exception handler */
#include <courier/Inbasket2.h> /* contains Toolkit Inbasket
                                   functions, can use
                                   Inbasket2__de.h instead */
#include <courier/MailTran5.h> /* contains Toolkit Mail
                                   Transport functions, can use
                                   MailTran5__de.h instead  */
#include <courier/FilingSu1.h>  /* contains the
                                   CH__StringToName function
                                   definition*/
#include <courier/xnstime.h>    /* contains the difference
                                   between XNS and UNIX
                                   times */
#include <ctype.h>         /* contains functions for testing
                              characters */
```

Filing service

```
#include <stdio.h>         /* contains standard I/O definitions */
#include <sys/types.h>     /* contains "system" types definitions
                              */
#include <errno.h>         /* contains integer expression
                              definition for errors */
#include <signal.h>        /* contains facilities for handling
                              exceptional conditions */
#include <courier/except.h>     /* contains macro definitions
                                   for the exception handler */
#include <ctype.h>         /* contains functions for testing
                              characters */
#include <sys/times.h>     /* contains types and functions for
                              manipulating date and time */
#include <courier/FilingSu1.h> /* contains
#include <courier/Authenti2.h> /* contains Toolkit
                                   Authentication functions,
                                        can use Authenti2__de.h
                                        instead */
#include <courier/Filing6.h>         /* contains Toolkit Filing
                                        functions, can use
                                        Filing6__de.h instead */
#include <courier/filetypes.h>       /* contains definitions for
                                        various Xerox file types */
#include <courier/xnstime.h>         /* contains the difference
                                        between XNS and UNIX
                                        times */
#include <dirent.h>            /* contains filesystem-independent
                                  directory information */
```

Printing service

```
#include <stdio.h>             /* contains standard I/O definitions */
#include <sys/types.h>         /* contains "system" types definitions
                                  */
#include <errno.h>             /* contains integer expression
                                  definition for errors */
#include <signal.h>            /* contains facilities for handling
                                  exceptional conditions */
#include <courier/except.h>          /* contains  macro definitions
                                        for the exception handler */
#include <courier/Printing3.h>       /* contains Toolkit Printing
                                        functions, can use
                                        Printing3__de.h instead */
#include <pwd.h>               /* contains the passwd structure */
#include "papersize.h"         /* contains paper dimension
                                  descriptions */
```

XNS filing service

Remote files must be represented in the following form before the file can be accessed on the XNS file server:

/path![version]/filename![version]

For example, suppose the remote file you wish to access is called "MyFile" and it's located on the remote directory called "RemoteDirect". If you specify the path as

/RemoteDirect/MyFile

you'll get an access error message indicating that the file does not exist. Instead, the path must be:

/RemoteDirect!1/MyFile!1

where "RemoteDirect" and "MyFile" both have a version number of 1 in this example.

In order to access a file, a directory handle must first be obtained and opened for the entire path. For example, to access the file "/directory1/directory2/MyFile", you must acquire the handle for "/directory1" and open this directory using the **Open()** function. Next, you must acquire the handle for "/directory1/directory2" and open it. With this handle you can now access "MyFile".

Available XNS functions

This sections gives an alphabetical listing of all functions calls available within the XNS *libxnstk.a* library. The headers associated with each XNS function are listed within the square brackets ([ ]) following its description. Note that the prefix for each function has been left off. You must include the prefix associated with each function if you use the [service].h header.

AbortRetrival()	direct the mail service to retain a message until the client is ready to accept [*MailTran5.h, MailTran5__de.h*]
AddGroupProperty()	add a unique group type property to an object [*Clearing2.h, Clearing2__de.h*]
AddItemProperty()	add a property of a specified value to an object [*Clearing2.h, Clearing2__de.h*]
AddMember()	add a new member to a group type property of an object [*Clearing2.h, Clearing2__de.h*]
AddSelf()	add user to the Clearinghouse database [*Clearing2.h, Clearing2__de.h*]
BeginPost()	initiate the posting of a message with a mail service [*MailTran5.h, MailTran5__de.h*]
BeginRetrieval()	initiate the retrieval of one or more messages from a specified delivery slot [*MailTran5.h, MailTran5__de.h*]
ChangeAttributes()	modify the access-related attributes of a specific file [*Filing6.h, Filing6__de.h*]
ChangeBodyPartsStatus()	update the status of one or more message body parts [*Inbasket2.h, Inbasket2__de.h*]
ChangeControls()	modify specific controls associated with a file [*Filing6.h, Filing6__de.h*]
ChangeItem()	assign a new value to an item type property [*Clearing2.h, Clearing2__de.h*]
ChangeMessageStatus()	update a specified range of messages from new to known [*Inbasket2.h, Inbasket2__de.h*]
ChangeSimpleKey()	change a simple key that is registered with the Authentication Service [*Authenti2.h, Authenti2__de.h*]
ChangeStrongKey()	change a strong key that is registered with the Authentication Service [*Authenti2.h, Authenti2__de.h*]
Close()	indicate to the File Service that a specific file handle is no longer wanted for the remainder of the current session [*Filing6*.h, *Filing6__de.h*]
Continue()	determine the duration, in seconds, permitted for an inactive session before it is terminated by the File Service [*Filing6.h, Filing6__de.h*]
Copy()	duplicate an existing file or directory [*Filing6.h, Filing6__de*.h]
Create()	make a new file [*Filing6.h, Filing6__de.h*]

Create()	initiate a terminal emulation session with a mainframe computer system [*MailTran5.h, MailTran5__de.h*]
CreateAlias()	add a new alias to an object in the Clearinghouse database [*Clearing2.h, Clearing2__de.*h]
CreateObject()	create a unique object in the Clearinghouse database [*Clearing2.h, Clearing2__de.h*]
CreateSimpleKey()	register a new simple key with the Authentication Service [*Authenti2.h, Authenti2__de.h*]
CreateStrongKey()	register a strong key with the Authentication Service [*Authenti2.h, Authenti2__de.h*]
CheckSimpleCredentials()	verify the identity of the initiator [*Authenti2.h, Authenti2__de.h*]
Delete()	remove an existing file [*Filing6.h, Filing6__de.*h]
Delete()	remove one or more contiguous messages from the inbasket [*Inbasket2.h, Inbasket2__de.h]*
DeleteAlias()	remove an alias of an object in the Clearinghouse database [*Clearing2.h, Clearing2__de.h*]
DeleteMember()	delete a member from a group type property of an object [*Clearing2.h, Clearing2__de.h*]
DeleteObject()	delete a unique object from the Clearinghouse database [*Clearing2.h, Clearing2__de.h*]
DeleteProperty()	remove a specific property from an object [*Clearing2.h, Clearing2__de.h*]
DeleteSelf()	deletes the user [*Clearing2.h, Clearing2__de.h*]
DeleteSimpleKey()	delete a simple key that is registered with the Authentication Service [*Authenti2.h, Authenti2__de.h*]
DeleteStrongKey()	delete a strong key that is registered with the Authentication Service [*Authenti2.h, Authenti2__de.h*]
Deserialize()	reconstruct a file and its descendants from a previously serialized file [*Filing6.h, Filing6__de.h*]
EndPost()	signal the mail service that the message is complete and no more data is to follow [*MailTran5.h, MailTran5__de.h*]
EndRetrieval()	end the current delivery slot retrieval sequence [*MailTran5.h, MailTran5__de.h*]
Find()	locate and open a file in a directory [*Filing6.h, Filing6__de.h*]
GetAttributes()	retrieve the attribute and attribute value pairs of a specific file [*Filing6.h, Filing6__de.h*]
GetControls()	determine the file access associated with a specific file [*Filing6.h, Filing6__de.h*]
GetPrinterProperties()	query the print service for the static properties of a printer [*Printing3.h, Printing3__de.h*]
GetPrintRequestStatus()	ascertain the status of a document that was sent to a printer [*Printing3.h, Printing3__de.h*]
GetPrinterStatus()	determine the availability of the print service [*Printing3.h, Printing3__de.h]*

GetSize()	retrieve a tally of the disk space occupied by all the messages in an inbasket [*Inbasket2.h, Inbasket2__de.h*]
GetStrongCredentials()	acquire privileged user authority from the Authentication Service [*Authenti2.h, Authenti2__de.h*]
IsMember()	determine if a named object is a member of a group type property [*Clearing2.h, Clearing2__de.h*]
List()	enumerate the files in a directory and return desired attributes [*Filing6.*h, *Filing6__de.h*]
ListAliases()	list the objects in a specific domain which are aliases [*Clearing2.h, Clearing2__de.h*]
ListAliasesOf()	list the aliases of an object [*Clearing2.h, Clearing2__de.h*]
ListDomains()	list domains within an organization [*Clearing2.h, Clearing2__de.h*]
ListDomainsServed()	obtain a list of the domains served by the Clearinghouse [*Clearing2.h, Clearing2__de.h*]
ListOrganizations()	list the names of organizations in the Clearinghouse database [*Clearing2.h, Clearing2__de.h*]
ListObjects()	list objects in a domain [*Clearing2.h, Clearing2__de.h*]
ListProperties()	list the ID number of every property associated with an object [*Clearing2.h, Clearing2__de.h*]
Logon()	initiate access to a File Service [*Filing6.h, Filing6__de.h*]
Logon()	initiate a new inbasket session with the mail service [*Inbasket2.h, Inbasket2__de.h*]
Logoff()	end the current File Service session [*Filing6.h, Filing6__de.h*]
Logoff()	end an inbasket session with the mail service [*Inbasket2.h, Inbasket2__de.h*]
LookupObject()	ascertain the complete name of an object in the Clearinghouse database [*Clearing2.h, Clearing2__de.h*]
MailCheck()	determine the state of an inbasket [*Inbasket2.h, Inbasket2__de.h*]
MailPoll()	quickly determine the state of an inbasket [*Inbasket2.h, Inbasket2__de.h*]
MailPoll()	determine if messages are present in a delivery mail slot [*MailTran5.h, MailTran5__de.h*]
Move()	change the directory structure of the filing service without creating or deleting files [*Filing6.h, Filing6__de.h*]
Open()	make a file available for use [*Filing6.h, Filing6__de.h*]
PostOneBodyPart()	submit data to a mail service [*MailTran5.h, MailTran5__de.h*]
Print()	access bulk transfer data in a source and send it to the print service queue [*Printing3.h, Printing3__de.h*]
Replace()	remove the contents of a file, and then replace it with data received from a specific source [*Filing6.h, Filing6__de.h*]
ReplaceBytes()	overwrite the contents of a file or to append new data to a file[*Filing6.h, Filing6__de.h*]
Retrieve()	read the contents of an existing file and transfer them to the client [*Filing6.h, Filing6__de.h*]

RetrieveAddresses()	query the clearinghouse server for a list of all the network addresses it recognizes [*Clearing2.h, Clearing2__de.h*]
RetrieveBodyParts()	copy specific parts of an inbasket message [*Inbasket2.h, Inbasket2__de.h*]
RetrieveBytes()	read a range of bytes within a file [*Filing6.h, FIling6__de.h*]
RetrieveContent()	extract the message in the envelope that is at the head of the delivery slot queue [*MailTran5.h, MailTran5__de*.h]
RetrieveEnvelope()	extract an envelope's header information from the delivery slot queue [*MailTran5.h, MailTran5__de*.h]
RetrieveEnvelopes()	extract a particular message (envelope) from the inbasket [*Inbasket2.h, Inbasket2__de.h*]
RetrieveItem()	determine the value of an item type property that is associated with an object [*Clearing2.h, Clearing2__de.h*]
RetrieveMembers()	extract, or retrieve, the value of a group type property associated with an object [*Clearing2.h, Clearing2__de.h*]
Serialize()	compress all the descriptive information and data of a file and its descendants into a series of eight-bit bytes [*Filing6.h, Filing6__de.h*]
ServerPoll()	determine if the mail service currently in use will accept additional messages for posting [*Mailtran5.h, Mailtran5__de.h*]
Store()	create a file that contains specific data [*Filing6.h, Filing6__de.h*]
UnifyAccessLists()	assign the access list attributes (i.e., permissions) of a directory to all its descendants [*Filing6.h, Filing6__de.h*]

Duplicate functions

There are some duplicate function names in the Document Interfaces Toolkit libraries, as outlined in Table 6-1.

Table 6-1. **Duplicate XNS functions**

Function	Header	Description
Delete()	*Filing6.h*	Remove an existing file
	Inbasket2.h	Remove messages from Mail
Logon()	*Filing6.h*	Initiate access to File Service
	Inbasket2.h	Initiate inbasket Mail session
Logoff()	*Filing6.h*	End current File Service
	Inbasket2.h	End inbasket Mail session
Mailpoll()	*Inbasket2.h*	Determine state of inbasket
	MailTran5.h	Determine if messages are in delivery mail slot

If you must combine some duplicate functions into one program, use the appropriate header file and prefix your functions to specify their full name. See the *Using the libxnstk.a library* section at the beginning of this chapter, or the *Document Interfaces Toolkit Reference Manual* for more details on using appropriate header files.

7. XNS function examples

This chapter presents example code fragments for the XNS functions to illustrate their usage. Only the major functions have been listed. Consult the *Document Interfaces Toolkit Reference Manual* for a complete description of all the available XNS functions.

The examples are organized alphabetically according to their header file name. Each example has the following format:

- Function name
- Function brief description
- Function syntax
- Example code fragment
- Description of the example

Authenti2.h

Authentication2__CheckSimpleCredentials()

The **Authentication2__CheckSimpleCredentials()** function is used to verify that the correct password has been submitted to the Authentication Service.

Authentication2__CheckSimpleCredentials(__Connection, __BDTprocptr, credentials, verifier)

Example

```
Authentication2__CheckSimpleCredentialsResults
    checksimplecredresult;

Authentication2__Credentialscredentials;
Authentication2__Verifier     verifier;

char        *checksecnameptr;

checksecnameptr = MakeSimpleCredsAndVerifier(
    secnameptr, 0, &credentials, &verifier);

checksimplecredresult =
    Authentication2__CheckSimpleCredentials(connected,
    NULL, credentials, verifier);
```

In the preceding example, **MakeSimpleCredsAndVerifier** creates the simple credentials and verifier for the user. **Authentication2__CheckSimpleCredentials** compares the simple key in verifier against the simple key that is registered for the user. **credentials** contains information regarding the user's name and password. **checksimplecredresult** has a value of **TRUE** if the simple key registered for the user and the simple key in verifier match. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. Bulk data transfer is not required, so **__BDTprocptr** is set to **NULL**.

Authentication2__DeleteSimpleKey()

The **Authentication2__DeleteSimpleKey()** function is used to delete a strong key that is registered with the Authentication Service.

Authentication2__DeleteSimpleKey(__Connection, __BDTprocptr, credentials, verifier, name)

Example

```
Authentication2__Credentials    credentials;
Authentication2__Verifier       verifier;
char                            *deletenameptr;

deletenameptr =
    MakeSimpleCredsAndVerifier(nameptr, 0, &credentials,
    &verifier);

(void)Authentication2__DeleteSimpleKey
    (connected, NULL, credentials, verifier,
    *CH__StringToName(nameptr));
```

In the preceding example, the strong key registered with **nameptr** will be deleted from the Authentication Service. **MakeSimpleCredsAndVerifier** creates the strong credentials and verifier for the user. **nameptr** contains the name of the user whose strong key will be deleted. **CH__StringToName()** translates this name into a three-part name so that it can be recognized by the Clearinghouse. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. Bulk data transfer is not required, so **__BDTprocptr** is set to **NULL**.

Authentication2__GetStrongCredentials()

The **Authentication2__GetStrongCredentials()** function is used to retrieve credentials in order to prove one's strong identity to a specified communications recipient.

Authentication2__GetStrongCredentials(__Connection, __BDTprocptr, initiator, recipient, nonce)

Example

```
Authentication2__GetStrongCredentialsResults
    strongcredresult;
char        authserver[128];

strcpy(authserver, "Authentication
    Service:CHServers:CHServers");

strongcredresult =
    Authentication2__GetStrongCredentials(connected,
        NULL, *CH__StringToName(nameptr),
        *CH__StringToName(authserver), nonce);
```

In the preceding example, strong credentials are created for the user defined by **nameptr** in order to prove the user's identity to the recipient, **authserver**. The well-known name of the Authentication Service is "Authentication Service:CHServers:CHServers" which is copied into **authserver**. **CH__StringToName()** translates both **nameptr** and **authserver** into three-part names so that they may be recognized by the Clearinghouse. **strongcredresult** has one member, **credentialsPackage** which is encrypted with the user's key. Once decrypted, it will contain the user's credentials. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. Bulk data transfer is not required, so **__BDTprocptr** is set to **NULL**.

Clearing2.h

Clearinghouse2__AddItemProperty()

The **Clearinghouse2__AddItemProperty()** function is used to add a property of a specified value to an object.

Clearinghouse2__AddItemProperty(__Connection, __BDTprocptr, name, newProperty, value, agent)

Example

```
Clearinghouse2__AddItemPropertyResults additemprtyrslt;
char       *nametoaddpropertyto;
Clearinghouse2__Item value;
char       *val[3];

nametoaddpropertyto = "Jones:Sunnyvale:USA";

val[0] = "ABCD Division";

value.length = 1;
value.sequence = (Unspecified *)&val[0];

additemprtyrslt = Clearinghouse2__AddItemProperty(
    connected, NULL,
    *CH__StringToName(nametoaddpropertyto),
    CHEntries0__user, value, *agent);
```

In the preceding example, the user property is added to "Jones:Sunnyvale:USA" in the Clearinghouse database. Properties for Clearinghouse databases include an ID number and a value, or descriptive field. In this example, **CHEntries0__user** returns an ID of 10003 as defined in the *CHEntries0.h* header file, indicating to the Clearinghouse that a user property is to be added. The Clearinghouse property value is defined by the **value** argument, or "ABCD Division". **CH__StringToName()** is used to translate **nametoaddpropertyto** into a three-part name so that it will be recognized by the Clearinghouse. **agent** contains the client's credentials and verifier, as defined by the Authentication protocol. The returned argument **additemprtyrslt** contains the full name of "Jones:Sunnyvale:USA". **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. No bulk data transfer is required, so **__BDTprocptr** is set to **NULL**.

Clearinghouse2__AddMember()

The **Clearinghouse2__AddMember()** function is used to add a new member to a group type property of an object.

Clearinghouse2__AddMember(__Connection, __BDTprocptr, name, property, newMember, agent)

Example

```
Clearinghouse2__AddMemberResults addmemberresult;
char              *grouptoaddto;
char              *newmember;

grouptoaddto = "xnstoolkit:green1:fermat";
newmember = "Jones:Sunnyvale:USA";

addmemberresult = Clearinghouse2__AddMember(
    connected, NULL, *CH__StringToName(grouptoaddto),
    CHEntries0__members,
    *CH__StringToName(newmember), *agent);
```

In the preceding example, a new member, "Jones:Sunnyvale:USA" is added to the "xnstoolkit:green1:fermat" group within the Clearinghouse database. **CH__StringToName()** translates both "Jones:Sunnyvale:USA" and "xnstoolkit:green1:fermat" to three-part names so that they may be recognized by the Clearinghouse. **CHEntries0__members** indicates to the Clearinghouse that a member group property is being added. No Bulk Data Transfer is required, so **__BDTprocptr** is set to **NULL**. The returned parameter **addmemberresult** has one member, **distinguishedObject**, which contains the distinguished name of "xnstoolkit:green1:fermat". **connected** contains the courier connect number. A method for obtaining this connection number is shown in Chapter 6.

Clearinghouse2__AddSelf()

The **Clearinghouse2__AddSelf()** function is used to add the user identified by the **agent** argument to a group property of an object.

Clearinghouse2__AddSelf(__Connection, __BDTprocptr, name, property, agent)

Example

```
Clearinghouse2__AddSelfResults     addselfresult;
char       *grouptoaddto;

grouptoaddto = "xnstoolkit:green1:fermat";

addselfresult = Clearinghouse2__AddSelf(connected,
    NULL, *CH__StringToName(grouptoaddto),
    CHEntries0__members, *agent);
```

In the preceding example, the user is added to the group property in the Clearinghouse database. **CHEntries0__members** indicates to the Clearinghouse that a Clearinghouse name, or member, is being added to a group property. **agent** identifies the user and verifies the user's credentials. The user is then added to the group property of **grouptoaddto**, or "xnstoolkit:green1:fermat". **CH__StringToName()** translates **grouptoaddto** to a three-part name so that it will be recognized by the Clearinghouse. **addselfresult** will contain the distinguished name of "xnstoolkit:green1:fermat". **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. No bulk data transfer is required, so **__BDTprocptr** is set to **NULL**.

Clearinghouse2__ChangeItem()

The **Clearinghouse2__ChangeItem()** function is used to assign a new value to an item type property.

Clearinghouse2__ChangeItem(__Connection,
__BDTprocptr,name, property, newValue, agent)

Example

```
Clearinghouse2_ChangeItemResults changeitemresult;
Clearinghouse2_Item     newValue;
char        *newVal[3];
char        *objectwhoseitemwillbechanged;

objectwhoseitemwillbechanged =
    "Jones:Sunnyvale:USA";

newVal[0] = "XYZ Division";

newValue.length = 1;
newValue.sequence = (Unspecified *)&newVal[0];

changeitemresult = Clearinghouse2_ChangeItem(
    connected, NULL,
    *CH_StringToName(objectwhoseitemwillbechanged),
    CHEntries0_user, newValue, *agent);
```

In the preceding example, the value for the user property will be changed for "Jones:Sunnyvale:USA" in the Clearinghouse database. Properties in the Clearinghouse are defined by a ID number followed by a value field. **CHEntries0__user** returns 10003 as defined by the *CHEntries0.h* header file. 10003 is the code for user property ID, indicating to the Clearinghouse that a user property is to be changed. The new value for the user property is defined by the **newValue** argument in this example, or "XYZ Division". **CH__StringToName()** translates **objectwhoseitemwillbechanged** into a three-part name so that it can be recognized by the Clearinghouse. The returned argument **changeitemresult** contains the full path name of "Jones:Sunnyvale:USA". **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. No bulk data transfer is required, so **__BDTprocptr** is set to **NULL**.

Clearinghouse2__CreateAlias()

The **Clearinghouse2__CreateAlias()** function is used to add a new alias to an object in the Clearinghouse database.

Clearinghouse2__CreateAlias(__Connection, __BDTprocptr, alias, sameAs, agent)

Example

```
Clearinghouse2__CreateAliasResults createaliasresult;
char       *alias;
char       *nametoalias;

nametoalias = "Johnson:Sunnyvale:USA";
alias = "JJ:San Jose:USA";

createaliasresult = Clearinghouse2__CreateAlias(
    connected, NULL, *CH__StringToName(alias),
    *CH__StringToName(nametoalias), *agent);
```

In the preceding example, an alias is created for "Johnson:Sunnyvale:USA". The alias will be "JJ:San Jose:USA". **CH__StringToName()** translates **alias** and **nametoalias** into three-part names, so that they can be recognized by the Clearinghouse. **agent** contains the credentials and verifier of the client as defined in the Authentication protocol. No Bulk Data Transfer is required, so **__BDTprocptr** is set to **NULL**. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6.

Clearinghouse2__CreateObject()

The **Clearinghouse2__CreateObject()** function is used to create a new distinguished object in the Clearinghouse database.

Clearinghouse2__CreateObject(__Connection, __BDTprocptr, name, agent)

Example

```
char       * object;
object = "Jones:Sunnyvale:USA";

(void)Clearinghouse2__CreateObject(connected, NULL,
    *CH__StringToName(object), *agent);
```

In the preceding example, the object "Jones:Sunnyvale:USA" is created in the Clearinghouse database. **CH__StringToName()** translates **object** into a three-part name so that it can be recognized by the Clearinghouse. **agent** contains the client's credentials and verifier as defined in the Authentication protocol. No Bulk Data Transfer is required, so **__BDTprocptr** is set to **NULL**. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6.

Clearinghouse2__DeleteAlias()

The **DeleteAlias()** function is used to remove an alias of an object in the Clearinghouse database.

DeleteAlias(__Connection, __BDTprocptr, alias, agent)

Example

```
Clearinghouse2__DeleteAliasResults  deletealiasresult;
char        *aliastodelete;

aliastodelete = "JJ:San Jose:USA";

deletealiasresult = Clearinghouse2__DeleteAlias(
    connected, NULL, *CH__StringToName(aliastodelete),
    *agent);
```

In the preceding example, the alias "JJ:San Jose:USA" is deleted from the Clearinghouse database. **CH__StringToName()** translates **aliastodelete** into a three-part name so that it can be recognized by the Clearinghouse. **agent** contains credentials and verifier of the client as defined in the Authentication protocol. The returned argument **deletealiasresult** contains the full name of the object to which "JJ:San Jose:USA" was aliased. No Bulk Data Transfer is required, so **__BDTprocptr** is set to **NULL**. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6.

Clearinghouse2__DeleteMember()

The **Clearinghouse2__DeleteMember()** function is used to delete a member from a group type property of an object.

Clearinghouse2__DeleteMember(__Connection, __BDTprocptr, name, property, member, agent)

Example

```
Clearinghouse2__DeleteMemberResults deletememrslt;
char        *grouptodeletefrom;
char        *membertodelete;

grouptodeletefrom = "xnstoolkit:green1:fermat";
membertodelete = "GBeichler:San Jose:USA";

deletememrslt = Clearinghouse2__DeleteMember(
    connected, NULL,
    *CH__StringToName(grouptodeletefrom),
    CHEntries0__members,
    *CH__StringToName(membertodelete), *agent);
}
```

In the preceding example, "GBeichler:San Jose:USA" is deleted from the group property of **xnstoolkit:green1:fermat**. **CHEntries0__members** returns the value of 3 as defined by the *CHEntries0.h* header file. The number 3 is the code for member, indicating to the Clearinghouse that a member is to be changed or deleted. **CH__StringToName()** translates **grouptodeletefrom** into a three-part name so that it can be recognized by the

Clearinghouse. The **agent** argument contains the client's credentials and verifier, as defined in the Authentication protocol. The returned argument **deletememrslt** contains the full path name of "xnstoolkit:green1:fermat". No bulk data transfer is required, so **__BDTprocptr** is set to **NULL. connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6.

Clearinghouse2__DeleteObject()

The **Clearinghouse2__DeleteObject()** function is used to delete an object from the Clearinghouse database.

Clearinghouse2__DeleteObject(__Connection, __BDTprocptr, name, agent)

Example

```
char        * objecttodelete;

objecttodelete = "Jones:Sunnyvale:USA";

(void)Clearinghouse2__DeleteObject(connected, NULL,
    *CH__StringToName(objecttodelete), *agent);
```

In the preceding example, "Jones:Sunnyvale:USA" is deleted from the Clearinghouse database. All aliases that point to "Jones:Sunnyvale:USA" are also deleted. **CH__StringToName()** translates **objecttodelete** into a three-part name so that it can be recognized by the Clearinghouse. **agent** contains the client's credentials and verifier, as defined in the Authentication protocol. Bulk data transfer is not required, so **__BDTprocptr** is set to **NULL. connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6.

Clearinghouse2__DeleteProperty()

The **Clearinghouse2__DeleteProperty()** function is used to remove a specific property from an object.

Clearinghouse2__DeleteProperty(__Connection, __BDTprocptr, name, property, agent)

Example

```
Clearinghouse2__DeletePropertyResults deleteproprtyrslt;
char        *objectwhosepropertywillbedeleted;

objectwhosepropertywillbedeleted =
    "Jones:Sunnyvale:USA";
deleteproprtyrslt = Clearinghouse2__DeleteProperty(
    connected, NULL,
    *CH__StringToName(objectwhosepropertywillbedeleted),
    CHEntries0__user, *agent);
```

In the preceding example, the user property will be deleted from "Jones:Sunnyvale:USA" in the Clearinghouse database. **CHEntries0__user** returns a value of 10003 as defined by the *CHEntries0.h* header file which indicates to the Clearinghouse database that a user property is to be modified.

CH__StringToName() translates **objectwhosepropertywillbedeleted** into a three-part name so that it can be recognized by the Clearinghouse. **agent** contains the client's credentials and verifier, as defined by the Authentication protocol. The returned argument **deleteproprtyrslt** contains the full path name of "Jones:Sunnyvale:USA". **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. Since bulk data transfer is not required, **__BDTprocptr** is set to **NULL**.

Clearinghouse2__DeleteSelf()

The **Clearinghouse2__DeleteSelf()** function deletes the user identified by the **agent** argument from a group type property of an object.

Clearinghouse2__DeleteSelf(__Connection, __BDTprocptr, name, property, agent)

Example

```
Clearinghouse2_DeleteSelfResults   deleteselfresult;
char        *grouptodeletefrom;

grouptodeletefrom = "xnstoolkit:green1:fermat";

deleteselfresult = Clearinghouse2_DeleteSelf(
    connected, NULL,
    *CH_StringToName(grouptodeletefrom),
    CHEntries0_members, *agent);
```

In the preceding example, the user, as defined by **agent**, is deleted from the group property in the Clearinghouse database. **agent** contains the client's credentials and verifier, as defined by the Authentication protocol. The user will be deleted from the "xnstoolkit:green1:fermat" group property. **CHEntries0__members** returns a value of 3 as defined by the *CHEntries0.h* header file. The number 3 is the code for member, indicating to the Clearinghouse that a group property is to be modified. **deleteselfresult** contains the full path name of "xnstoolkit:green1:fermat" from which the user was removed. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. Since bulk data transfer is not required, **__BDTprocptr** is set to **NULL**.

Clearinghouse2__IsMember()

The **Clearinghouse2__IsMember()** function determines if a named object is a member of a group type property.

Clearinghouse2__IsMember(__Connection, __BDTprocptr, memberOf, property, secondaryProperty, name, agent)

Example

```
Clearinghouse2_IsMemberResults ismemberresult;
char        *objectwithgroupproperty;
char        *lookingforthisname;
```

```
objectwithgroupproperty = "xnstoolkit:green1:fermat";
lookingforthisname = "Jones:Sunnyvale:USA";

ismemberresult = Clearinghouse2__IsMember(
    connected, NULL,
    *CH__StringToName(objectwithgroupproperty),
    Clearinghouse2__all, Clearinghouse2__all,
    *CH__StringToName(lookingforthisname), *agent);
```

In the preceding example, "xnstoolkit:green1:fermat" is queried to see if it contains the member "Jones:Sunnyvale:USA". **CH__StringToName()** translates "xnstoolkit:green1:fermat" and "Jones:Sunnyvale:USA" to three-part names, so that they may be recognized by the Clearinghouse. Both **property** and **secondaryProperty** contain the value **Clearinghouse2__all**, indicating that every object belonging to "xnstoolkit:green1:fermat" will be compared against "Jones:Sunnyvale:USA" to see if a match exists. The returned parameter **ismemberresult** contains two members, **isMember** and **distinguishedObject**. **IsMember** is a Boolean variable whose value indicates if "Jones:Sunnyvale:USA" was found. **distinguishedObject** contains the full path name of "xnstoolkit:green1:fermat". **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. Since bulk data transfer is not required, **__BDTprocptr** is set to **NULL**.

Clearinghouse2__ListAliasesOf()

The **Clearinghouse2__ListAliasesOf()** function is used to list the aliases of an object.

Clearinghouse2__ListAliasesOf(__Connection, __BDTprocptr, pattern, list, agent)

Example

```
Clearinghouse2__ListAliasesOfResults listaliasesofresult;
char        * local;
char        *aliasesoftolist;

aliasesoftolist = "Jones:Sunnyvale:USA";

local = "aliases.lst";
if ((fout = open(local, O__CREAT | O__TRUNC |
    O__WRONLY, 0664)) < 0) {
        printf("\t\tCAN'T OPEN local file\n");
        return;
}

listaliasesofresult = Clearinghouse2__ListAliasesOf(
    connected, bulkretrieveproc,
    *CH__StringToName(aliasesoftolist),
    BulkData1__immediateSink, *agent);
```

In the preceding example, all aliases for "Jones:Sunnyvale:USA" are retrieved. **CH__StringToName()** translates "Jones:Sunnyvale:USA" into a three-part name so that it may be recognized by the Clearinghouse. **agent** is a structure whose two members contain the credentials and verifier of the client. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. Bulk data transfer is required in this example. **bulkretrieveproc** is the C procedure defined by the C programmer that will perform the bulk data transfer. An example of such a procedure is shown in Chapter 6. In the preceding example, the aliases found for "Jones:Sunnyvale:USA" will be placed into the file **aliases.lst**. The transfer of data into **aliases.lst** is actually done within the **bulkretrieveproc** procedure. The opening of the file is illustrated here. **BulkData1__immediateSink** indicates that the data will be sinked into the UNIX environment.

Clearinghouse2__ListDomain()

The **Clearinghouse2__ListDomain()** function is used to list domain names within an organization.

Clearinghouse2__ListDomain(__Connection, __BDTprocptr, pattern, list, agent)

Example

```
Clearinghouse2__TwoPartName   pattern;
Clearinghouse2__ObjectName*nameresult;
char        * local;

nameresult = CH__StringToName(nameptr);
pattern.organization = nameresult->organization;
pattern.domain  = "xns*";

local = "domains.lst";
if((fout = open(local, O__CREAT|O__TRUNC|
    O__WRONLY, 0664)) < 0) {
        printf("\t\tCAN'T OPEN local file\n");
        return;
}

(void)Clearinghouse2__ListDomain(connected,
    bulkretrieveproc, pattern, BulkData1__immediateSink,
    *agent);
```

The preceding example will list all domains that start with "xns" in the organization defined by **pattern.organization**. **fout** is a pointer to the UNIX file that will contain the list of domains that match the pattern. Bulk data transfer is required in this example. **bulkretrieveproc** is the C procedure defined by the C programmer that will perform the bulk data transfer. An example of such a procedure is shown in Chapter 6. The transfer of data into the **fout** file is done within the **bulkretrieveproc** procedure. The opening of this file is illustrated here. Since data is to be sinked into the UNIX environment, **BulkData1__immediateSink** is used. **agent** contains two members: the client's credentials and the client's verifier. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6.

Clearinghouse2__ListDomainServed()

The **Clearinghouse2__ListDomainServed()** function is used to obtain a list of the domains served by a specific Clearinghouse service.

Clearinghouse2__ListDomainServed(__Connection, __BDTprocptr, domains, agent)

Example

```
char        * local;

local = "domains.lst";
if ((fout = open(local, O_CREAT | O_TRUNC |
    O_WRONLY, 0664)) < 0) {
            printf("\t\tCAN'T OPEN local file\n");
            return;
}

(void)Clearinghouse2_ListDomainServed( connected,
    bulkretrieveproc, BulkData1_immediateSink, *agent);
```

The preceding example lists all domains served by a specific Clearinghouse service and places that list into the UNIX file domains.lst. Bulk data transfer is required in this example. **bulkretrieveproc** is the C procedure defined by the C programmer that will perform the bulk data transfer. An example of such a procedure is shown in Chapter 6. The transfer of data into the file **domains.lst** is done within the **bulkretrieveproc** procedure. The opening of this file is illustrated here. Since data is to be sinked into the UNIX environment, **BulkData1__immediateSink** is used. **agent** contains two members, the client's credentials and verifier. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6.

Clearinghouse2__ListObjects()

The **Clearinghouse2__ListObjects()** function is used to list the objects in a domain that have a specific property associated with them.

Clearinghouse2__ListObjects(__Connection, __BDTprocptr, pattern, property, list, agent)

Example

```
char        * local;
char        * pattern;

local = "objects.lst";
if ((fout = open(local, O_CREAT | O_TRUNC |
    O_WRONLY, 0664)) < 0) {
            printf("\t\tCAN'T OPEN local file\n");
            return;
}

pattern = "gary*:Sunnyvale:USA";
```

```
(void)Clearinghouse2_ListObjects(connected,
    bulkretrieveproc, *CH_StringToName(pattern),
    Clearinghouse2_all, BulkData1_immediateSink, *agent);
```

In the preceding example, all objects starting with "gary" are listed in the domain USA. The results are placed into the UNIX file **objects.lst**. The argument **Clearinghouse2_all** indicates that all objects in the domain that match the pattern are to be listed. Bulk data transfer is required in this example. **bulkretrieveproc** is the C procedure defined by the C programmer that will perform the bulk data transfer. An example of such a procedure is shown in Chapter 6. The transfer of data into the file "objects.lst" is done within the **bulkretrieveproc** procedure. The opening of this file is illustrated here. Since data is to be sinked into the UNIX environment, **BulkData1_immediateSink** is used. **agent** contains two members, the client's credentials and verifier. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6.

Clearinghouse2_ListOrganizations()

The **Clearinghouse2_ListOrganizations()** function is used to list the names of organizations in the Clearinghouse database.

Clearinghouse2_ListOrganizations(__Connection,
__BDTprocptr, pattern, list, agent)

Example

```
char      *pattern;
char      * local;

local = "organization.lst";
if ((fout = open(local, O_CREAT | O_TRUNC
    |O_WRONLY, 0664)) < 0) {
        printf("\t\tCAN'T OPEN local file\n");
        return;
}

pattern = "*tool*";

(void)Clearinghouse2_ListOrganizations(connected,
    bulkretrieveproc, pattern, BulkData1_immediateSink,
    *agent);
```

The preceding example lists all organizations within the Clearinghouse database that contain the string "tool". The results are placed into a UNIX file "organization.lst". This example requires bulk data transfer. **bulkretrieveproc** is the C procedure defined by the C programmer that performs the bulk data transfer. An example of such a procedure is shown in Chapter 6. The transfer of data into the file "organization.lst" is done within the **bulkretrieveproc** procedure. The opening of this file is illustrated here. Since data is to be sinked into the UNIX environment, **BulkData1_immediateSink is used**. **agent** contains two members, the client's credentials and verifier. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6.

Clearinghouse2__ListProperties()

The **Clearinghouse2__ListProperties()** function is used to list the ID number of every property associated with an object.

Clearinghouse2__ListProperties(__Connection, __BDTprocptr, pattern, agent)

Example

```
Clearinghouse2__ListPropertiesResult listpropertiesresult;
char        *objectwithproperties;

objectwithproperties = "xnstoolkit:green1:fermat";

listpropertiesresult = Clearinghouse2__ListProperties(
    connected, NULL,
    *CH__StringToName(objectwithproperties), *agent);
```

The preceding example lists the ID number of every property associated with "xnstoolkit:green1:fermat". **CH__StringToName()** translates "xnstoolkit:green1:fermat" into a three-part name so that it may be recognized by the Clearinghouse. **agent** contains two members, the client's credentials and verifier. No bulk data transfer is required in this example, so **__BDTprocptr** is set to **NULL**. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6.

Clearinghouse2__LookupObject()

The **Clearinghouse2__LookupObject()** function is used to query the Clearinghouse database for the full name of an object that is contained within it.

Clearinghouse2__LookupObject(__Connection, __BDTprocptr, name, agent)

Example

```
Clearinghouse2__LookupObjectResults lookupobjectrslt;
char        *objecttolookup;

objecttolookup = "Jones:Sunnyvale:USA";

lookupobjectrslt = Clearinghouse2__LookupObject(
    connected, NULL,
    *CH__StringToName(objecttolookup), *agent);
```

The preceding example queries the Clearinghouse database for the full name of "Jones:Sunnyvale:USA". The results are placed into the **distinguishedObject** member of **lookupobjectrslt.** **agent** contains two members, the client's credentials and verifier. No bulk data transfer is required in this example, so **__BDTprocptr** is set to **NULL**. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6.

Clearinghouse2__RetrieveAddresses()

The **Clearinghouse2__RetrieveAddresses()** function is used to query the clearinghouse server for a list of all of its network addresses.

Clearinghouse2__RetrieveAddresses (__Connection, __BDTprocptr)

Example

```
Clearinghouse2__RetrieveAddressesResults rtrvaddrrslt;

rtrvaddrrslt = Clearinghouse2__RetrieveAddresses(
    connected, NULL);
```

In the preceding example, **rtrvaddrrslt** contains a list of the network addresses recognized by the Clearinghouse server. The Clearinghouse server accessed is based upon the value of **connected**. **connected** contains the courier connection number. No bulk data transfer is required so **__BDTprocptr** is set to **NULL**.

Clearinghouse2__RetrieveItem()

The **Clearinghouse2__RetrieveItem()** function is used to determine the value of an item type property that is associated with an object.

Clearinghouse2__RetrieveItem(__Connection, __BDTprocptr, pattern, property, agent)

Example

```
Clearinghouse2__RetrieveItemResults retrieveitemresult;
char        *objectwithproperties;

objectwithproperties = "xnstoolkit:green1:fermat";

retrieveitemresult = Clearinghouse2__RetrieveItem(
    connected, NULL,
    *CH__StringToName(objectwithproperties),
    Clearinghouse2__all, *agent);
    }
```

The preceding example returns all item properties of the first object encountered that matches "xnstoolkit:green1:fermat" in the Clearinghouse database. **CH__StringToName()** translates **objectwithproperties** into a three-part name so that it may be recognized by the Clearinghouse. **Clearinghouse2__all** indicates that all item properties are to be returned. **agent** contains the client's credentials and verifier. **retrieveitemresult** contains two members: **distinguishedObject** and **value**. **distinguishedObject** contains the full name of "xnstoolkit:green1:fermat". **value** contains the value of the item property. No bulk data transfer is required in this example, so **__BDTprocptr** is set to **NULL**. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6.

Clearinghouse2__RetrieveMembers()

The **Clearinghouse2__RetrieveMembers()** function is used to retrieve the value of a group type property associated with an object.

Clearinghouse2__RetrieveMembers(__Connection, __BDTprocptr, pattern, property, membership, agent)

Example

```
Clearinghouse2__RetrieveMembersResults retmemrslt;
Clearinghouse2__Property    membersproperty = 3;
char        *grouptoretrievefrom;
char        * local;

grouptoretrievefrom = "xnstoolkit:green1:fermat";

local = "propvalues.lst";
if ((fout = open(local, O__CREAT | O__TRUNC |
    O__WRONLY, 0664)) < 0) {
        printf("\t\tCAN'T OPEN local file\n");
        return;
}

retmemrslt = Clearinghouse2__RetrieveMembers(
    connected, bulkretrieveproc,
    *CH__StringToName(grouptoretrievefrom),
    membersproperty, BulkData1__immediateSink, *agent);
```

The preceding example returns the value of the group member property associated with "xnstoolkit:green1:fermat. **CH__StringToName()** translates **xnstoolkit:green1:fermat** into a three-part name so that it may be recognized by the Clearinghouse. **membersproperty** has a value of 3 indicating that the member group value is to be retrieved. The value of 3 is defined by the *CHEntries0.h* header file to be the member property. **agent** contains the client's credentials and verifier. Bulk data transfer is required in this example. **bulkretrieveproc** is the C procedure defined by the C programmer that will perform the bulk data transfer. An example of such a procedure is shown in Chapter 6. In the preceding example, the member properties found for "Jones:Sunnyvale:USA" will be placed into the file "propvalues.lst". The transfer of data into **propvalues.lst** is done within the **bulkretrieveproc** procedure. The opening of the file is illustrated here. **BulkData1__immediateSink** indicates that the data will be sinked into the UNIX environment. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6.

Filing6.h

Filing6__Logon()

The **Filing6__Logon** function is used to initiate access to a File Service.

Filing6__Logon(__Connection, __BDTprocptr, service, credentials, verifier)

Example

```
Filing6__Credentials  credentials
Filing6__LogonResultslogonresult;
Filing6__SecondaryCredentials    secondarycreds;
char  *hostnameptr;
int name;
int pwd;

hostnameptr = "Jones:Sunnyvale:USA"

name = 0;
pwd = 0;
name = MakeSimpleCredsAndVerifier (name, pwd,
    &credentials.primary, &verifier);

secondarycreds.designator = Filing6__strengthNone;
credentials.secondary = secondarycreds;

logonresult = Filing6__Logon(connected, NULL,
    *CH__StringToName(hostnameptr), credentials, verifier);

session = logonresult.session;
```

In the preceding example, a File Service session is initiated for "Jones:Sunnyvale:USA". Setting both **name** and **pwd** to 0 cause **MakeSimpleCredsAndVerifier** to find the logged on user and the logged on user's password. The **credentials** and **verifier** returned from **MakeSimpleCredsAndVerifier** are then passed to the **Filing6__Logon** function. **CH__StringToName()** translates "Jones:Sunnyvale:USA" into a three-part name so that it may be recognized by the Clearinghouse. The returned parameter, **logonresult**, contains one member, **session**, which identifies the session to the File Service. **session** may then be passed to other File Service calls. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. Since bulk data transfer is not required in this example, **__BDTprocptr** is set to **NULL**.

Filing6__Open()

The **Filing6__Open** function is used to make a file available for use.

Open(__Connection, __BDTprocptr, attributes, directory, controls, session)

Example

```
Filing6_OpenResults openresult;
Filing6_AttributeSequence  attrseq;
Filing6_Attributepathattr[1];

attrseq.length = 1;
attrseq.sequence = pathattr;
filename = "MyFile";

pathattr[0].type = Filing6_pathname;
StringToAttr(filename, &pathattr[0]);

openresult = Filing6_Open(connected, NULL, attrseq,
    Filing6_nullHandle, nullControls, session);
```

In the preceding example, the file "MyFile" will be opened for use. Since no other parameters for the **attrseq** argument are specified, the highest version number for "MyFile" will be opened. **Filing6__nullHandle** implies that the starting directory to begin the search for the file is the root directory. **nullControls** indicates that no controls are specified. **session** is the session handle returned from an earlier call to **Filing6__Logon()**. The returned parameter **openresult** has one member, **handle**, which may be passed as an argument to all calls that are to access the file during the current session. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. No bulk data transfer is required, so **__BDTprocptr** is set to **NULL**.

Inbasket2.h

Inbasket2__ChangeMessageStatus()

The **Inbasket2__ChangeMessageStatus()** function is used to update a specified range of messages from new to known.

Inbasket2__ChangeMessageStatus(__Connection, __BDTprocptr, range, changeUserDefinedStatus, newUserDefinedStatus, session)

Example

```
Inbasket2_Range    range;
Boolean          changeUserDefinedStatus;
LongCardinal          newUserDefinedStatus;

range.low = nullIndex;
range.high = 10;
changeUserDefinedStatus = 1;
newUserDefinedStatus = Inbasket2__known;

(void)Inbasket2__ChangeMessageStatus(connected,
    NULL, range, changeUserDefinedStatus,
    newUserDefinedStatus, inbasketsession);
```

In the preceding example, the first ten inbasket messages are updated to known. The range is specified by **range.low** and **range.high**. Setting **changeUserDefinedStatus** to a 1 causes the **existenceOfMessage** parameter to be set to **KNOWN** and **userDefinedStatus** to be updated with the value of **newUserDefinedStatus**, or "Inbasket2__known" in this example. **inbasketsession** is defined by **logonresult.session** from **Inbasket2__Logon()**. Bulk data transfer is not required, so **__BDTprocptr** is set to **NULL**. **connected** contains the courier connection number. A method for obtaining this connection number is show in Chapter 6.

Inbasket2__Delete()

The **Inbasket2__Delete()** function is used to remove one or more contiguous messages from the inbasket.

Inbasket2__Delete(__Connection, __BDTprocptr, range, session)

Example

```
Inbasket2__Range     range;

range.low = 3;
range.high = nullIndex;

(void)Inbasket2__Delete(connected, NULL, range,
    inbasketsession);
```

In the preceding example, the inbasket messages between the third and last will be deleted. The range of messages to be deleted is defined by the **range.low** and **range.high** arguments. **inbasketsession** is defined by **logonresult.session** from **Inbasket2__Logon**. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. Bulk data transfer is not required, so **__BDTprocptr** is set to **NULL**.

Inbasket2__GetSize()

The **Inbasket2__GetSize()** function is used to retrieve a tally of the disk space occupied by all the messages in an inbasket.

Inbasket2__GetSize(__Connection, __BDTprocptr, inbasket, credentials, verifier)

Example

```
Inbasket2__GetsizeResults       getsizeresult;
Authentication2__Credentials    credentials;
Authentication2__Verifier       verifier;
char        inbasketptr;
```

```
inbasketptr = "Jones:Sunnyvale:USA";
nameptr = MakeSimpleCredsAndVerifier(nameptr, 0,
    &credentials, &verifier);

getsizeresult = Inbasket2__Getsize(connected, NULL,
    * CH__StringToName(inbasketptr), credentials, verifier);
```

In the preceding example, the disk space occupied by all the messages for "Jones:Sunnyvale:USA" will be placed into **getsizeresult**. **CH__StringToName()** converts **inbasketptr** into a three-part name. **inbasketptr** represents the mail recipient, "Jones:Sunnyvale:USA" in this example. **credentials** and **verifier** are the credentials and verifier returned by the **MakeSimpleCredsAndVerifier()** function. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. Bulk data transfer is not required, so **__BDTprocptr** is set to **NULL**.

Inbasket2__Logoff()

The **Inbasket2__Logoff()** function is used to end an inbasket session with the mail service.

Inbasket2__Logoff(__Connection, __BDTprocptr, session)

Example

```
(void)Inbasket2__Logoff(connected, NULL, inbasketsession);
```

In the preceding example, an inbasket session is terminated. the session terminated, **inbasketsession**, is that session defined by **logonresult.session** from **Inbasket2__Logon**. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. No bulk data transfer is required, so **__BDTprocptr** is set to **NULL**.

Inbasket2__Logon()

The **Inbasket2__Logon()** function is used to initiate a new inbasket session with the mail service.

Inbasket2__Logon(__Connection, __BDTprocptr, inbasket, credentials, verifier)

Example

```
Inbasket2__LogonResults          logonresult;
Authentication2__Credentials     credentials;
Authentication2__Verifier        verifier;
char                   *inbasketofrecipient;

inbasketofrecipient = "Jones:Sunnyvale:USA";
nameptr = MakeSimpleCredsAndVerifier(nameptr, 0,
    &credentials, &verifier);

logonresult = Inbasket2__Logon(connected, NULL,
    * CH__StringToName(inbasketofrecipient),credentials,
    verifier);

inbasketsession = logonresult.session;
```

In the preceding example, a new inbasket session for "Jones:Sunnyvale:USA" is initiated with the mail service. **CH__StringToName()** converts **inbasketofrecipient** into a three-part name needed by the mail service. **credentials** and **verifier** arguments are the credentials and verifier returned by the **MakeSimpleCredsAndVerifier()** function. **inbasketsession** contains the inbasket session handle to be passed to other Inbasket2 functions, such as **Inbasket2__Logoff()**. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. No bulk data transfer is required, so **__BDTprocptr** is set to **NULL**.

Inbasket2__MailCheck()

The **Inbasket2__MailCheck()** function is used to determine the state of an inbasket.

Inbasket2__MailCheck(__Connection, __BDTprocptr, session)

Example

```
Inbasket2__MailCheckResults     mailcheckresult;

mailcheckresult = Inbasket2__MailCheck(connected,
    NULL, inbasketsession);
```

In the preceding example, the state of an inbasket session is checked. The session to be checked is defined by **inbasketsession**, which is in turned defined by the earlier call to **Inbasket2__Logon()**. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. No bulk data transfer is required, so **__BDTprocptr** is set to **NULL**.

Inbasket2__MailPoll()

The **Inbasket2__MailPoll()** function is used to determine the state of an inbasket without initiating an inbasket session.

Inbasket2__MailPoll(__Connection, __BDTprocptr, inbasket, credentials, verifier)

Example

```
Authentication2__Credentials    credentials;
Authentication2__Verifier       verifier;
char        *inbasketptr;

inbasketpt = "Jones:Sunnyvale:USA";
nameptr = MakeSimpleCredsAndVerifier(nameptr, 0,
    &credentials, &verifier);

mailpollresult = Inbasket2__MailPoll(connected, NULL,
    * CH__StringToName(inbasketptr), credentials, verifier);
```

In the preceding example, the state of the inbasket for "Jones:Sunnyvale:USA" is checked. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. No bulk data transfer is required, so **__BDTprocptr** is set to **NULL**. **CH__StringToName()** translates the **inbasketptr** string into a three-part name needed by the mail service. **credentials** and **verifier** are the credentials and verifier obtained by the **MakeSimpleCredsAndVerifier()** function.

Inbasket2__RetrieveEnvelopes()

The **Inbasket2__RetrieveEnvelopes()** function is used to extract a particular message (envelope) from the inbasket.

Inbasket2__RetrieveEnvelopes(__Connection, __BDTprocptr, index, whichMsg, session)

Example

```
Inbasket2__RetrieveEnvelopesResults
retrieveenvelopesresult;

retrieveenvelopesresult = Inbasket2__RetrieveEnvelopes(
    connected, NULL, Inbasket2__nullIndex, Inbasket2__next,
    inbasketsession);

message = retrieveenvelopesresult.index;
```

In the preceding example, all messages in the inbasket are enumerated. The inbasket session is defined by **inbasketsession** which is in turn defined by an earlier call to **Inbasket2__Logon()**. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. No bulk data transfer is required, so **__BDTprocptr** is set to **NULL**.

Mailing5.h

MailTransport5__AbortRetrival()

The **MailTransport5__AbortRetrival()** function is used to postpone the delivery of a message.

MailTransport5__AbortRetrival (__Connection, __BDTprocptr, session)

Example

```
(void) MailTransport5__AbortRetrival(connected, NULL,
    retsession);
```

In the preceding example, the mail service is directed to stop the retrieval process and to retain the remainder of the messages until the client is ready to accept them. **retsession** is the session handle returned from a previous call to **MailTransport5__BeginRetrieval()**. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. Since no bulk data transfer is required, **__BDTprocptr** is set to NULL.

MailTransport5__BeginRetrieval()

The **MailTransport5__BeginRetrieval()** function is used to initiate the retrieval of one or more messages from a delivery slot.

**MailTransport5__BeginRetrieval(__Connection,
    __BDTprocptr, deliverySlot, credentials, verifier)**

Example

```
MailTransport5__BeginRetrievalResults
                                        beginretrievalresul
                                        t;
Authentication2__Credentialscredentials;
Authentication2__Verifier    verifier;
char  *mailrecipientname;

mailrecipientname = "Jones:Sunnyvale:USA"

mailrecipientname = MakeSimpleCredsAndVerifier
    (mailrecipientname, 0, &credentials, &verifier);

beginretrievalresult =
MailTransport5__BeginRetrieval(connected,
    NULL, *CH__StringToName(mailrecipientname),
    credentials, verifier);

retsession = beginretrievalresult.session;
```

In the preceding example, **retsession** contains the mailtransport session handle that can be passed to the various ***Retrieval()** functions. **mailrecipientname** identifies the mail slot to be accessed. In this example, "Jones:Sunnyvale:USA" is the mail slot. **CH__StringToName()** translates **mailrecipientname** into a three-part name so that it may be recognized by the Clearinghouse. **credentials** and **verifier** are the credentials and verifier returned from the call to the **MakeSimpleCredsAndVerifier**. **connected** contains the courier connected number. A method for obtaining this connection number is shown in Chapter 6. Since bulk data transfer is not required, **__BDTprocptr** is set to NULL.

MailTransport5__EndPost()

The **MailTransport5__EndPost()** function is used to signal the mail service that the message initiated by **BeginPost()** is complete and no more data is to follow.

**MailTransport5__EndPost(__Connection, __BDTprocptr,
    session, abortPost)**

Example

```
MailTransport5__EndPostResults endpostresult;
Boolean  abortPost;

abortPost = 0;

endpostresult = MailTransport5__EndPost(connected, NULL,
    session, abortPost);
```

In the preceding example, **abortPost** is set to 0, indicating that the message will be sent to the recipients listed in an earlier call to **MailTransport5__BeginPost()**. **session** is defined by the session handle returned from an earlier call to **MailTransport5__BeginRetrieval()**. **connected** contains the courier connection number. A method for obtaining this connection number is shown in chapter 6. Bulk data transfer is not required, so **__BDTprocptr** is set to **NULL**.

MailTransport5__EndRetrieval()

The **MailTransport5__EndRetrieval()** function is used to end the current delivery slot retrieval sequence.

MailTransport5__EndRetrieval(__Connection, __BDTprocptr, session)

Example

```
(void)MailTransport5__EndRetrieval(connected, NULL,
    retsession);
```

In the preceding example, the current delivery slot retrieval sequence is terminated. **retsession** is defined by the session handle returned from an earlier call to **MailTransport5__BeginRetrieval()**. **connected** contains the courier connection number. A method for obtaining this connection number is shown in chapter 6. Bulk data transfer is not required, so **__BDTprocptr** is set to **NULL**.

MailTransport5__MailPoll()

The **MailTransport5__MailPoll()** function is used to determine if messages are present in a delivery mail slot.

MailTransport5__MailPoll (__Connection, __BDTprocptr, deliverySlot, credentials, verifier)

Example

```
MailTransport5__MailPollResults mailpollmtresult;
Authentication2__Credentialscredentials;
Authentication2__Verifierverifier;
char  *mailrecipientname;

mailrecipientname = "Jones:Sunnyvale:USA"
mailrecipientname = MakeSimpleCredsAndVerifier
    (mailrecipientname, 0, &credentials, &verifier);

mailpollmtresult = MailTransport5__MailPoll(connected,
    NULL, *CH__StringToName(mailrecipientname),
    credentials, verifier);
```

In the preceding example, the Boolean **mailpollmtresult** will indicate the presence of mail in the delivery slot or an empty delivery slot. A **TRUE** indicates there is mail ready for retrieval, a **FALSE** indicates that the delivery slot is empty. **CH__StringToName()** will translate the mail recipient's name into a three-part name so that it can be recognized by the Clearinghouse. **credentials** and **verifier** are the credentials and verifier returned by the call to the **MakeSimpleCredsAndVerifier()**. **connected** contains the courier connection number. A method for obtaining this connection number is shown in Chapter 6. Since bulk data transfer is not required, **__BDTprocptr** is set to **NULL**.

MailTransport5__RetrieveEnvelope()

The **MailTransport5__RetrieveEnvelope()** function is used to extract the header information from a delivery slot mail.

MailTransport5__RetrieveEnvelope(__Connection, __BDTprocptr, session)

Example

```
MailTransport5__RetrieveEnvelopeResults
    retrieveenveloperesult;

retrieveenveloperesult = MailTransport5__RetrieveEnvelope
    (connected, NULL, retsession);
```

In the preceding example, **retrieveenveloperesult** will contain two members: **empty** and **envelope**. The **empty** member will indicate whether the active delivery slot is empty or if it has envelopes available. **envelope** will be the envelope itself. **retsession** is defined by the session handle returned from an earlier call to **MailTransport5__BeginRetrieval()**. **connected** contains the courier connection number. A method for obtaining this connection number is shown in chapter 6. Bulk data transfer is not required, so **__BDTprocptr** is set to **NULL**.

A. Additional programs

This appendix provides several sample programs that may be helpful to the programmer using the XNS portion of the Document Interfaces Toolkit. These programs are not supported items, but are given here as a helpful hint to the programmer. To use these programs, you must type them onto your workstation, and compile them.

The first program, simpleauth.c, contains the procedure definition for **MakeSimpleCredsAndVerifier**, used in Chapter 7. The next four programs, attribute.c, getservice.c, gettype.c, and misc.c, may be used with calls to the Filing service. These contain various miscellaneous procedures that are identified within each program. Be sure to include these programs when you compile your XNS code. The last file, *papersize.h,* is a header file that contains paper size definitions. This header file may be used with programs interfacing to the Printing service.

simpleauth.c

```
/** simpleauth.c 1.3, last change 11/12/88 **/
static char sccsid[] = "@(#)simpleauth.c\t\t1.3";

#include <stdio.h>
#include <courier/Authenti1.h>
#include <courier/Authenti2.h>
#include <ctype.h>
#include <termio.h>

extern char   * CH_NameToString(/* chname */);
static void expand_name(/* char * name, char * buf,
        int buflen */);
static Authentication2_Clearinghouse_Name * getXNSuser (/*
        char * name, char * passwd, Cardinal * hpasswdp */);
static Cardinal hashpass (/* char * hpw */);
static void password_prompt (/* char * buf, int buflen */);
extern  char * strchr();
static Authentication2_Clearinghouse_Name * str_to_chname
    (/* char * name */);
static void user_prompt (/* char * buf, int buflen */);

static void
expand_name (name, buf, buflen)
    char   * name;
    char   * buf;
    int  buflen;
{
    char   * domain;
    int  len;
    if (strlen(name) >= buflen)
        return;
    strcpy(buf, name);
    if (domain = strchr(name, ':')) {
        if (strchr(&domain[1], ':'))
            return;
    } else {
        strcat(buf, ":");
        len = strlen(buf);
        cour_get_value("default domain", &buf[len],
            buflen-len);
    }
    strcat(buf, ":");
    len = strlen(buf);
    cour_get_value("default organization", &buf[len],
        buflen-len);
}

static Authentication2_Clearinghouse_Name *
getXNSuser (name, passwd, hpasswdp)
    char    * name;
    char    * passwd;
    Cardinal * hpasswdp;
{
    char    buf[85];
    static char    name_buf[85];
    Authentication2_Clearinghouse_Name * chname;

    if (!name) {
        cour_get_value("user name", buf, sizeof(buf));
        if (!buf[0]) {
            user_prompt(buf, sizeof(buf));
            if (!buf[0])
                return str_to_chname("::");
        }
        name = buf;
    }
    expand_name(name, name_buf, sizeof(name_buf));
    if (name_buf[0])
        cour_set_value("user name", name_buf);
    if (!passwd) {
        cour_get_value("user password", buf, sizeof(buf));
        if (!buf[0])
            password_prompt(buf, sizeof(buf));
        passwd = buf;
    }
    if (passwd[0])
        cour_set_value("user password", passwd);
    *hpasswdp = hashpass(passwd);
    return str_to_chname(name_buf);
}

static Cardinal
hashpass (hpw)
    char    * hpw;
{
    unsigned long    hash;
    char        ch;

    hash = 0;
    while ((ch = *hpw++) != '\0') {
        hash = (hash << 16) + (isupper(ch) ? tolower(ch) : ch);
        hash %= 65357;
    }
    return (Cardinal)hash;
}

char *
MakeAuth1SimpleCredsAndVerifier (name, passwd, credentials,
    verifier)
    char   * name;
    char   * passwd;
    Authentication1_Credentials* credentials;
    Authentication1_Verifier* verifier;
{
    Authentication1_ClearinghouseName    * chname;
    Cardinal   length;
    Unspecified    * data, * buf, * seq;
    Cardinal   c1;

    chname = (Authentication1_ClearinghouseName *)
        getXNSuser(name, passwd, &c1);
    if (credentials) {
        credentials->type =
          Authentication1_simpleCredentials;
        length =
          sizeof_Authentication1_ClearinghouseName
              (chname);
        data = Allocate(length);
        (void)externalize_Authentication1_ClearinghouseName
          (chname, data);
        seq = credentials->value.sequence = Allocate(length);
        credentials->value.length = length;
        buf = data;
        for ( ; length > 0; length--, seq++)
          buf += internalize_Unspecified(seq, buf);
        free(data);
    }
    if (verifier) {
        verifier->length = 1;
        verifier->sequence = Allocate(sizeof_Unspecified(0));
        verifier->sequence[0] = (Unspecified)c1;
    }
    return CH_NameToString(chname);
}

char *
MakeSimpleCredsAndVerifier (name, passwd, credentials,
    verifier)
    char   * name;
    char   * passwd;
    Authentication2_Credentials* credentials;
    Authentication2_Verifier* verifier;
```

```
{
    Authentication2_Clearinghouse_Name * chname;
    Cardinal  length;
    Unspecified   * data, * buf, * seq;
    Cardinal  c1;

    chname = getXNSuser(name, passwd, &c1);
    if (credentials) {
        credentials->type =
Authentication2_simpleCredentials;
        length =
          sizeof_Authentication2_Clearinghouse_Name
              (chname);
        data = Allocate(length);
        (void)externalize_Authentication2_Clearinghouse_Name
    (chname,data);
        seq = credentials->value.sequence = Allocate(length);
        credentials->value.length = length;
        buf = data;
        for ( ; length > 0; length--, seq + + )
          buf + = internalize_Unspecified(seq, buf);
        free(data);
    }
    if (verifier) {
        verifier->length = 1;
        verifier->sequence = Allocate(sizeof_Unspecified(0));
        verifier->sequence[0] = (Unspecified)c1;
    }
    return CH_NameToString(chname);
}

static void
password_prompt (buf, buflen)
    char  * buf;
    int  buflen;
{
register int     ch;
register char   * cp;
    int  lflag;
    struct termio  termio1;
    int  tty_fd;
    FILE   * tty_fp;

    buf[0] = '\0';
    if ((tty_fd = open("/dev/tty", 2)) = = -1)
        return;
    tty_fp = fdopen(tty_fd, "r");
    setbuf(tty_fp, (char *)0);
    ioctl(tty_fd, TCGETA, &termio1);
    lflag = termio1.c_lflag;
    termio1.c_lflag &= ~ECHO;
    ioctl(tty_fd, TCSETA, &termio1);
    fprintf(stderr, "Enter XNS password: ");
    fflush(stderr);
    for (cp = buf; (ch = getc(tty_fp)) ! = EOF; ) {
        if (ch = = '\n' || ch = = '\r')
            break;
        if (cp < &buf[buflen])
            *cp++ = ch;
    }
    *cp = '\0';
    fprintf(stderr, "\n");
    fflush(stderr);
    termio1.c_lflag = lflag;
    ioctl(tty_fd, TCSETA, &termio1);
    fclose(tty_fp);
}

static Authentication2_Clearinghouse_Name *
str_to_chname (name)
    char  * name;
{
static   Authentication2_Clearinghouse_Name result;

    if (name) {
        result.object = name;
        if (name = strchr(result.object, ':')) {
            *name++ = '\0';
            result.domain = name;
            if (name = strchr(name,':')) {
                *name++ = '\0';
                result.organization = name;
                return &result;
            }
        }
    }
    return (Authentication2_Clearinghouse_Name *)0;
}

static void
user_prompt (buf, buflen)
    char   * buf;
    int  buflen;
{
    char   * cp;
    FILE   * tty;

    buf[0] = '\0';
    if (tty = fopen("/dev/tty", "r+")) {
        setbuf(tty, (char *)NULL);
        fprintf(stderr, "Enter XNS username: ");
        fflush(stderr);
        fgets(buf, buflen, tty);
        if (cp = strchr(buf, '\n'))
            *cp = '\0';
        if (tty != stdin)
            fclose(tty);
    }
}
```

attribute.c

```
/** attribute.c 1 **/

static  char  sccsid[] = "@(#)attribute.c\t\t1.1";

#ifdef FILING4
#include "filingV4.h"
#include "clearingV2.h"
#endif
#ifdef FILING5
#include "filingV5.h"
#include "clearingV2.h"
#endif
#ifdef FILING6
#include "filingV6.h"
#include "clearingV3.h"
#endif
#ifdef FILINGSUBSET1
#include "filingsuV1.h"
#include "clearingV3.h"
#endif
#ifdef FPSUBSET3
#include "fpsubsetV3.h"
#include "clearingV2.h"
#endif

StringToAttr (str, attr)
    char  * str;
    FILING_Attribute* attr;
{
    Unspecified  buf[2049];
    Unspecified  * bp;
    Cardinal  len;

    bp = buf + sizeof_Cardinal(len);
    len = externalize_String(&str, bp);
    (void)externalize_Cardinal(&len, buf);
    internalize_Sequence_of_Unspecified(&(attr->value), buf);
}
char *
AttrToString (attr)
    FILING_Attribute* attr;
{
    Unspecified  buf[2049];
    Unspecified  * bp;
    Cardinal  len;
    char        * strval;

    externalize_Sequence_of_Unspecified(&(attr->value), buf);
    bp = buf;
    bp + = internalize_Cardinal(&len, bp);
    bp + = internalize_String(&strval, bp);
    return strval;
}
```

```
CLEARINGHOUSE_Name *
AttrToUser (attr)              /* J.H.A. */
    FILING_Attribute     * attr;
{
    Unspecified   buf[2049];
    Unspecified   * bp;
    Cardinal      len;
    CLEARINGHOUSE_Name    * user;

  externalize_Sequence_of_Unspecified(&(attr->value), buf);
    bp = buf;
    bp + = internalize_Cardinal(&len, bp);
    bp + = internalize_String(&user->organization, bp);
    bp + = internalize_String(&user->domain, bp);
    bp + = internalize_String(&user->object, bp);
    return user;
}

UserToAttr (id, attr)
    CLEARINGHOUSE_Name id;
    FILING_Attribute* attr;
{
    Unspecified   buf[2049];
    Unspecified   * bp;
    Cardinal  len;

    bp = buf + sizeof_Cardinal(len);
    len = CLEARINGHOUSE_externalize_Name(&id, bp);
    (void)externalize_Cardinal(&len, buf);
    internalize_Sequence_of_Unspecified(&(attr->value), buf);
}

LongCardinalToAttr (val, attr)
    LongCardinal val;
    FILING_Attribute* attr;
{
    Unspecified   buf[3];
    Unspecified   * bp;
    Cardinal  len;

    bp = buf + sizeof_Cardinal(len);
    len = externalize_LongCardinal(&val, bp);
    (void)externalize_Cardinal(&len, buf);
    internalize_Sequence_of_Unspecified(&(attr->value), buf);
}

LongCardinal
AttrToLongCardinal (attr)
    FILING_Attribute* attr;
{
    Unspecified   buf[2];
    LongCardinal result;

    (void)externalize_Unspecified(attr->value.sequence, buf);
    (void)externalize_Unspecified((attr->value.sequence) + 1,
        buf + 1);
    (void)internalize_LongCardinal(&result, buf);
    return result;
}

BooleanToAttr (val, attr)
    int val;
    FILING_Attribute* attr;
{
    Boolean       boolval;
    Unspecified   buf[3];
    Unspecified   * bp;
    Cardinal   len;

    boolval = (Boolean) val;
    bp = buf + sizeof_Cardinal(len);
    len = externalize_Boolean(&boolval, bp);
    (void)externalize_Cardinal(&len, buf);
    internalize_Sequence_of_Unspecified(&(attr->value), buf);
}

int
AttrToBoolean (attr)
    FILING_Attribute* attr;
{
    Unspecified   buf[1];
    Boolean       result;

    (void)externalize_Unspecified(attr->value.sequence, buf);
    (void)internalize_Boolean(&result, buf);
    return result;
}

CardinalToAttr (val, attr)
    Cardinal  val;
    FILING_Attribute* attr;
{
    Unspecified   buf[3];
    Unspecified   * bp;
    Cardinal   len;

    bp = buf + sizeof_Cardinal(len);
    len = externalize_Cardinal(&val, bp);
    (void)externalize_Cardinal(&len, buf);
    internalize_Sequence_of_Unspecified(&(attr->value), buf);
}

Cardinal
AttrToCardinal (attr)
    FILING_Attribute* attr;
{
    Unspecified   buf[2];
    Cardinal   result;

    (void)externalize_Unspecified(attr->value.sequence, buf);
    (void)internalize_Cardinal(&result, buf);
    return result;
}
```

```
FileIDToAttr (value, attr)
    Cardinal  value[];
    FILING_Attribute* attr;
{
    Unspecified  buf[6];
    Unspecified  * bp;
    Cardinal  len;

    bp = buf + sizeof_Cardinal(len);
    len = FILING_externalize_FileID(value, bp);
    (void)externalize_Cardinal(&len, buf);
    internalize_Sequence_of_Unspecified(&(attr->value), buf);
}

Unspecified *
AttrToFileID (attr)
    FILING_Attribute* attr;
{
    Unspecified  * bp;
    Unspecified  buf[6];

    bp = Allocate(FILING_sizeof_FileID(0));
    (void)FILING_externalize_FileID(attr->value.sequence, buf);
    (void)FILING_internalize_FileID(bp, buf);
    return bp;
}
```

getservice.c

```
/** getservice.c 1.1 **/

static  char  sccsid[] = "@(#)getservice.c\t\t1.1";

extern  char  * strchr();
extern  char  * strrchr();

getserviceandfile (name, srvcptr, fileptr)
    char  * name;
    char  ** srvcptr;
    char  ** fileptr;
{
    char  * sptr;
    char  * fptr;

    *srvcptr = (char *)0;
    /*
     * look for Xerox forms first:
     *   [host]filename
     */
    if (sptr = strchr(name, '[')) {
        if (fptr = strrchr(sptr, ']')) {
            *fptr = '\0';
            *srvcptr = sptr + 1;
            *fileptr = fptr + 1;
```

```
                return 1;
            }
            return 0;
        }
        /*
         *  (host)filename
         */
        if (sptr = strchr(name, '(')) {
            if (fptr = strchr(sptr, ')')) {
                *fptr = '\0';
                *srvcptr = sptr + 1;
                *fileptr = fptr + 1;
                return 1;
            }
            return 0;
        }
        /*
         * look for XNS style with trailing : delimiter
         * (assumes no : in file name, use alternate spec instead)
         * object:domain:organization:filename
         * domain & organization are optional
         */
        if (fptr = strrchr(name, ':')) {
            *fptr = '\0';
            *srvcptr = name;
            *fileptr = fptr + 1;
            return 1;
        }
        return 1;
}
```

gettype.c

```
/** gettype.c 1.1 **/

static  char  sccsid[] = "@(#)gettype.c\t\t1.1";

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <courier/filetypes.h>
#include <courier/courier.h>

#define  CHARS_TO_READ2048

/*
 * routine:
 *   get_type
 * input:
 *   pointer to pathname of file
 *   read first CHARS_TO_READ
 *   if it contains "RasterEncoding/", assume RES (return
 *       TYPE_VPCanvas)
 *   if it contains "Interpress/Xerox/", assume IP (return
 *       TYPE_Interpress)
 *  if a char is 0 or high order bit set, assume binary (return
        tUNSPECIFIED)
 *  else assume text (tText)
 *  (Note order of RES/IP checking is important since RES files
        contain
 *  IP string also)
 *
 *  returns:
 *  LongCardinal containing filing defined file type
 */

extern  LongCardinal GetTypeAttribute();
static  Boolean is_type();
extern  char   * strrchr();

typedef struct st_s {
    char        * st_string;
    LongCardinal st_type;
} ST, * STP;

static  ST doc_types[] = {
    { "text",        TYPE_A },
    { "binary",         TYPE_I },
    { "directory",      TYPE_Directory },
    { "vp doc",         TYPE_VP },
    { "interpress",         TYPE_Interpress },
    { "vp canvas",      TYPE_VPCanvas },
    { "vp dictionary", TYPE_VPDictionary },
    { "vp mailnote",   TYPE_VPMailNote },
    { "vp reference",  TYPE_VPReference },
    { "serialized",     TYPE_S },
    { "vp filedrawer", TYPE_VPDrawer },
    { "vp application",    TYPE_VPApplication },
    { "vp spreadsheet",    TYPE_VPSpreadsheet },
    { "vp book",        TYPE_VPBook },
    { "vp recordfile", TYPE_VPRecordsfile },
    { "vp calendar",   TYPE_VPCalendar },
    { "vp icons",       TYPE_VPIcons },
    { "printerfont",   TYPE_Font },
    { "860 doc",        TYPE_860 },
    { 0 }
};


LongCardinal
get_type (pathname)
    char  * pathname;
{
    int fd;
    char  buffer[CHARS_TO_READ];
    int count;
    char  * ptr;
    LongCardinal type;
    struct stat fs;

    if (stat(pathname, &fs) = = -1)
        return(TYPE_I);         /* if error, assume tUnspecified */
    /*
     * if not a regular file look for directory...
     * if a directory and in root, then assume file drawer
     */
    if ((fs.st_mode & S_IFMT) != S_IFREG) {
        if (fs.st_mode & S_IFDIR) {
            if (strrchr(pathname, '/') == pathname)
                return TYPE_VPDrawer;
            else
                return TYPE_Directory;
        } else
            return TYPE_I;
    }
    if ((fd = open(pathname, 0)) < 0)
        return TYPE_I;          /* if error, assume tUnspecified */
    if (count = read(fd, buffer, sizeof(buffer))) {
        if (is_type(RESHDR, buffer, count) == TRUE)
            type = TYPE_VPCanvas;
        else if (is_type(INTERPRESSHDR, buffer, count) ==
            TRUE)
            type = TYPE_Interpress;
        else if (is_type(VPHDR, buffer, count) == TRUE)
            type = GetTypeAttribute(fd);
        else {
            type = TYPE_A;          /* assume tAsciiText */
            for (ptr = buffer; ptr < &buffer[count - 1]; ptr++) {
                /* if 0 or high order bit */
                if (*ptr == 0 || (*ptr & 0200)) {
                    /* assume tUnspecified */
                    type = TYPE_I;
                    break;
                }
            }
        }
    } else
        type = TYPE_I;          /* if error, assume tUnspecified */
    close(fd);
    return type;
}

static Boolean
is_type (type_string, file_snip, file_snip_len)
    char  * type_string;
    char  * file_snip;
    int  file_snip_len;
{
    char  string[128];      /* 128 should be enough for all cases
*/
    char  * cp, * boundary;
    int  type_string_len;

    type_string_len = strlen(type_string);
    if (file_snip_len > sizeof(string))
        file_snip_len = sizeof(string);
    if (file_snip_len < type_string_len)
        return FALSE;
    memcpy(string, file_snip, file_snip_len);
```

```
	lowercasen(string, file_snip_len);
	boundary = &string[file_snip_len - type_string_len];
	for (cp = string; cp < boundary; cp++) {
		if (memcmp(cp, type_string, type_string_len) == 0)
			return TRUE;
	}
	return FALSE;
}

char *
typetostring (type)
	LongCardinal type;
{
register STP	st;
static	char	string[80];

	for (st = &doc_types[0]; st->st_string; st++) {
		if (type == st->st_type)
			return st->st_string;
	}
	sprintf(string, "%ld", (long)type);
	return string;
}

LongCardinal
stringtotype (string)
	char	*string;
{
extern	long	atol(/* char * str */);
register STP	st;

	if (!*string)
		return(TYPE_VPMailNote);
	lowercase(string);
	for (st = &doc_types[0]; st->st_string; st++) {
		if (strcmp(string, st->st_string) == 0)
			return st->st_type;
	}
	return (LongCardinal)atol(string);
}
```

misc.c

```
/** misc.c 1.1 **/

static	char	sccsid[] = "@(#)misc.c\t\t1.1";

#include <ctype.h>

/* change case to upper*/
char *
uppercase (string1)
	char	*string1;
{
register char	*string2 = string1;
```

```
    while (*string2) {
        if (islower(*string2))
            *string2 = *string2 - 040;
        string2 + + ;
    }
    return string1;
}

/* change counted string to upper */
char *
uppercasen (string1, count)
    char   * string1;
    int  count;
{
register char   * string2 = string1;

    while (count--) {
        if (islower(*string2))
            *string2 = *string2 - 040;
        string2 + + ;
    }
    return string1;
}

/* change case to lower */
char *
lowercase (string1)
    char   * string1;

{
register char   * string2 = string1;

    while (*string2) {
        if (isupper(*string2))
            *string2 = *string2 + 040;
        string2 + + ;
    }
    return string1;
}

/* change counted string to lower */
char *
lowercasen(string1, count)
    char   * string1;
    int  count;
{
register char   * string2 = string1;

    while (count--) {
        if (isupper(*string2))
            *string2= *string2 + 040;
        string2 + + ;
    }
    return string1;
}
```

papersize.h

```
/** papersize.h 1.2 **/

/* $Header: papersize.h,v 1.1 87/07/01 11:25:11 ed Exp $ */

/*
 * $Log:    papersize.h,v $
 *
 */

static struct {
    char * sizename;
    int sizevalue;
} papersizetable[] = {
    "usLetter", (int) usLetter, /* 1 */
    "usLegal", (int) usLegal, /* 2 */
    "a0", (int) a0, /* 3 */
    "a1", (int) a1, /* 4 */
    "a2", (int) a2, /* 5 */
    "a3", (int) a3, /* 6 */
    "a4", (int) a4, /* 7 */
    "a5", (int) a5, /* 8 */
    "a6", (int) a6, /* 9 */
    "a7", (int) a7, /* 10 */
    "a8", (int) a8, /* 11 */
    "a9", (int) a9, /* 12 */
    "isoB0", (int) isoB0, /* 13 */
    "isoB1", (int) isoB1, /* 14 */
    "isoB2", (int) isoB2, /* 15 */
    "isoB3", (int) isoB3, /* 16 */
    "isoB4", (int) isoB4, /* 17 */
    "isoB5", (int) isoB5, /* 18 */
    "isoB6", (int) isoB6, /* 19 */
    "isoB7", (int) isoB7, /* 20 */
    "isoB8", (int) isoB8, /* 21 */
    "isoB9", (int) isoB9, /* 22 */
    "isoB10", (int) isoB10, /* 23 */
    "jisB0", (int) jisB0, /* 24 */
    "jisB1", (int) jisB1, /* 25 */
    "jisB2", (int) jisB2, /* 26 */
    "jisB3", (int) jisB3, /* 27 */
    "jisB4", (int) jisB4, /* 28 */
    "jisB5", (int) jisB5, /* 29 */
    "jisB6", (int) jisB6, /* 30 */
    "jisB7", (int) jisB7, /* 31 */
    "jisB8", (int) jisB8, /* 32 */
    "jisB9", (int) jisB9, /* 33 */
    "jisB10", (int) jisB10, /* 34 */
    "a10", (int) a10, /* 35 */
    (char *) 0, 0
};
```

Index

NOTES

NOTES

NOTES

NOTES

Customer Comments

XEROX®

VP Series Reference Library
Document Interfaces Toolkit User Guide

Our goal is to improve the organization, ease of use, and accuracy of this library. Your comments and suggestions will help us tailor our manuals to better suit your needs.

Name:____________________ Company:____________________

Address:____________________ City:____________________

State:____________________ Zip:____________________

Please rate the following:

	Excellent	Good	Fair	Poor
1. Is the organization suitable for your needs?	☐	☐	☐	☐
2. Are you able to easily find the information you need?	☐	☐	☐	☐
3. Are the illustrations useful?	☐	☐	☐	☐
4. Overall, how would you rate the documentation?	☐	☐	☐	☐

Did you find any errors?
Page number / Error

How can we improve the documentation?

We appreciate your comments regarding our documentation. Thank you for taking the time to reply.

Staple or tape

Fold here

No Postage
Necessary
If Mailed
In the
United States

BUSINESS REPLY MAIL

First Class Permit No. 229 El Segundo, California

Postage will be paid by Addressee

Xerox Corporation
Attn: Product Education WS, N2-15
701 South Aviation Boulevard
El Segundo, California 90245

Fold here

Staple or tape

XEROX

Xerox Corporation
701 South Aviation Boulevard
El Segundo, CA 90245

610E22850